Arm® Compiler

Version 6.13

Reference Guide

101754_0613_00_en

Arm® Compiler

Reference Guide

Copyright © 2019 Arm Limited or its affiliates. All rights reserved.

Release Information

Document History

Issue	Date	Confidentiality	Change
0613-00	09 October 2019	Non-Confidential	Arm Compiler v6.13 Release.

Non-Confidential Proprietary Notice

This document is protected by copyright and other related rights and the practice 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. **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 implementations infringe any third party patents.

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, third party patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

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.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant 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.

If any of the provisions contained in these terms conflict with any of the provisions of any click through or signed written agreement covering this document with Arm, then the click through or signed written agreement prevails over and supersedes the conflicting provisions of these terms. 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 the Agreement shall prevail.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm’s trademark usage guidelines at *http://www.arm.com/company/policies/trademarks*.

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

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

Additional Notices

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

Confidentiality Status

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

Unrestricted Access is an Arm internal classification.

Product Status

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

Web Address

www.arm.com

Contents
Arm® Compiler Reference Guide

Preface

About this book ... 30

Part A Arm Compiler Tools Overview

Chapter A1 Overview of the Arm® Compiler tools

A1.1 Arm® Compiler tool command-line syntax ... A1-38
A1.2 Support level definitions ... A1-39

Part B armclang Reference

Chapter B1 armclang Command-line Options

B1.1 Summary of armclang command-line options ... B1-48
B1.2 -C (armclang) ... B1-54
B1.3 -c (armclang) ... B1-56
B1.4 -D ... B1-57
B1.5 -E ... B1-58
B1.6 -e ... B1-59
B1.7 -fbare-metal-pie ... B1-60
B1.8 -fbracket-depth=N ... B1-61
B1.9 -fcommon, -fno-common ... B1-62
B1.10 -fdata-sections, -fno-data-sections ... B1-63
B1.11 -ffast-math, -fno-fast-math ... B1-64

B1.12 -ffixed-rN B1-65
B1.13 -ffp-mode B1-67
B1.14 -ffunction-sections, -fno-function-sections B1-69
B1.15 -fident, -fno-ident B1-71
B1.16 @file B1-72
B1.17 -fldm-stm, -fno-ldm-stm B1-73
B1.18 -fno-builtin B1-74
B1.19 -fno-inline-functions B1-76
B1.20 -flto, -fno-lto B1-77
B1.21 -fexceptions, -fno-exceptions B1-78
B1.22 -fomit-frame-pointer, -fno-omit-frame-pointer B1-79
B1.23 -fpic, -fno-pic B1-80
B1.24 -fropi, -fno-ropi B1-81
B1.25 -fropi-lowering, -fno-ropi-lowering B1-82
B1.26 -frwpi, -fno-rwpi B1-83
B1.27 -frwpi-lowering, -fno-rwpi-lowering B1-84
B1.28 -fsanitize B1-85
B1.29 -fshort-enums, -fno-short-enums B1-88
B1.30 -fshort-wchar, -fno-short-wchar B1-90
B1.31 -fstack-protector, -fstack-protector-all, -fstack-protector-strong, -fno-stack-protector B1-91
B1.32 -fstrict-aliasing, -fno-strict-aliasing B1-93
B1.33 -fsysv, -fno-sysv B1-94
B1.34 -ftrapv B1-95
B1.35 -fvectorize, -fno-vectorize B1-96
B1.36 -fvisibility B1-97
B1.37 -fwrapv B1-98
B1.38 -g, -gdwarf-2, -gdwarf-3, -gdwarf-4 (armclang) B1-99
B1.39 -I B1-100
B1.40 -include B1-101
B1.41 -L B1-102
B1.42 -l B1-103
B1.43 -M, -MM B1-104
B1.44 -MD, -MMD B1-105
B1.45 -MF B1-106
B1.46 -MG B1-107
B1.47 -MP B1-108
B1.48 -MT B1-109
B1.49 -march B1-110
B1.50 -marm B1-115
B1.51 -masm B1-116
B1.52 -mbig-endian B1-118
B1.53 -mbranch-protection B1-119
B1.54 -mcmodel B1-122
B1.55 -mcmse B1-123
B1.56 -mcpu B1-125
B1.57 -mexecute-only B1-132
B1.58 -mfloat-abi B1-133
B1.59 -mfpu B1-134
B1.60 -mimplicit-it B1-136

B1.61 -mlittle-endian B1-137
B1.62 -mno-neg-immediates B1-138
B1.63 -moutline, -mno-outline B1-140
B1.64 -mpixolib B1-143
B1.65 -munaligned-access, -mno-unaligned-access B1-145
B1.66 -mthumb B1-146
B1.67 -nostdlib B1-147
B1.68 -nostdlibinc B1-148
B1.69 -o (armclang) B1-149
B1.70 -O (armclang) B1-150
B1.71 -pedantic B1-152
B1.72 -pedantic-errors B1-153
B1.73 -Rpass B1-154
B1.74 -S B1-156
B1.75 -save-temps B1-157
B1.76 -shared (armclang) B1-158
B1.77 -std B1-159
B1.78 --target B1-161
B1.79 -U B1-162
B1.80 -u (armclang) B1-163
B1.81 -v (armclang) B1-164
B1.82 --version (armclang) B1-165
B1.83 --version_number (armclang) B1-166
B1.84 --vsn (armclang) B1-167
B1.85 -W B1-168
B1.86 -Wl B1-169
B1.87 -Xlinker B1-170
B1.88 -x (armclang) B1-171
B1.89 -### B1-172

Chapter B2 Compiler-specific Keywords and Operators

B2.1 Compiler-specific keywords and operators B2-174
B2.2 __alignof__ B2-175
B2.3 __asm B2-177
B2.4 __declspec attributes B2-179
B2.5 __declspec(noinline) B2-180
B2.6 __declspec(noreturn) B2-181
B2.7 __declspec(nothrow) B2-182
B2.8 __inline B2-183
B2.9 __promise B2-184
B2.10 __unaligned B2-185
B2.11 Global named register variables B2-186

Chapter B3 Compiler-specific Function, Variable, and Type Attributes

B3.1 Function attributes B3-193
B3.2 __attribute__((always_inline)) function attribute B3-195
B3.3 __attribute__((cmse_nonsecure_call)) function attribute B3-196
B3.4 __attribute__((cmse_nonsecure_entry)) function attribute B3-197
B3.5 __attribute__((const)) function attribute B3-198
B3.6 __attribute__((constructor(priority))) function attribute B3-199

B3.7 __attribute__((format_arg(string-index))) function attribute B3-200
B3.8 __attribute__((interrupt("type"))) function attribute B3-201
B3.9 __attribute__((malloc)) function attribute B3-202
B3.10 __attribute__((naked)) function attribute B3-203
B3.11 __attribute__((noinline)) function attribute B3-204
B3.12 __attribute__((nonnull)) function attribute B3-205
B3.13 __attribute__((noreturn)) function attribute B3-206
B3.14 __attribute__((nothrow)) function attribute B3-207
B3.15 __attribute__((pcs("calling_convention"))) function attribute B3-208
B3.16 __attribute__((pure)) function attribute B3-209
B3.17 __attribute__((section("name"))) function attribute B3-210
B3.18 __attribute__((unused)) function attribute B3-211
B3.19 __attribute__((used)) function attribute B3-212
B3.20 __attribute__((value_in_regs)) function attribute B3-213
B3.21 __attribute__((visibility("visibility_type"))) function attribute B3-215
B3.22 __attribute__((weak)) function attribute B3-216
B3.23 __attribute__((weakref("target"))) function attribute B3-217
B3.24 Type attributes B3-218
B3.25 __attribute__((aligned)) type attribute B3-219
B3.26 __attribute__((packed)) type attribute B3-220
B3.27 __attribute__((transparent_union)) type attribute B3-221
B3.28 Variable attributes B3-222
B3.29 __attribute__((alias)) variable attribute B3-223
B3.30 __attribute__((aligned)) variable attribute B3-224
B3.31 __attribute__((deprecated)) variable attribute B3-225
B3.32 __attribute__((packed)) variable attribute B3-226
B3.33 __attribute__((section("name"))) variable attribute B3-227
B3.34 __attribute__((unused)) variable attribute B3-228
B3.35 __attribute__((used)) variable attribute B3-229
B3.36 __attribute__((visibility("visibility_type"))) variable attribute B3-230
B3.37 __attribute__((weak)) variable attribute B3-231
B3.38 __attribute__((weakref("target"))) variable attribute B3-232

Chapter B4 Compiler-specific Intrinsics

B4.1 __breakpoint intrinsic B4-234
B4.2 __current_pc intrinsic B4-235
B4.3 __current_sp intrinsic B4-236
B4.4 __disable_fiq intrinsic B4-237
B4.5 __disable_irq intrinsic B4-238
B4.6 __enable_fiq intrinsic B4-239
B4.7 __enable_irq intrinsic B4-240
B4.8 __force_stores intrinsic B4-241
B4.9 __memory_changed intrinsic B4-242
B4.10 __schedule_barrier intrinsic B4-243
B4.11 __semihost intrinsic B4-244
B4.12 __vfp_status intrinsic B4-246

Chapter B5 Compiler-specific Pragmas

B5.1 #pragma clang system_header B5-248
B5.2 #pragma clang diagnostic B5-249

B5.3 #pragma clang section B5-251
B5.4 #pragma once B5-253
B5.5 #pragma pack(...) B5-254
B5.6 #pragma unroll[(n)], #pragma unroll_completely B5-256
B5.7 #pragma weak symbol, #pragma weak symbol1 = symbol2 .... B5-257

Chapter B6 Other Compiler-specific Features

B6.1 ACLE support B6-260
B6.2 Predefined macros B6-261
B6.3 Inline functions B6-266
B6.4 Half-precision floating-point data types B6-267
B6.5 Half-precision floating-point number format B6-269
B6.6 Half-precision floating-point intrinsics B6-270
B6.7 Library support for _Float16 data type B6-271
B6.8 BFloat16 floating-point number format B6-272
B6.9 TT instruction intrinsics B6-273
B6.10 Non-secure function pointer intrinsics B6-276

Chapter B7 armclang Integrated Assembler

B7.1 Syntax of assembly files for integrated assembler B7-278
B7.2 Assembly expressions B7-280
B7.3 Alignment directives B7-285
B7.4 Data definition directives B7-287
B7.5 String definition directives B7-290
B7.6 Floating-point data definition directives B7-292
B7.7 Section directives B7-293
B7.8 Conditional assembly directives B7-299
B7.9 Macro directives B7-301
B7.10 Symbol binding directives B7-303
B7.11 Org directive B7-305
B7.12 AArch32 Target selection directives B7-306
B7.13 AArch64 Target selection directives B7-308
B7.14 Space-filling directives B7-309
B7.15 Type directive B7-310
B7.16 Integrated assembler support for the CSDB instruction B7-311

Chapter B8 armclang Inline Assembler

B8.1 Inline Assembly B8-314
B8.2 File-scope inline assembly B8-315
B8.3 Inline assembly statements within a function B8-316
B8.4 Inline assembly constraint strings B8-320
B8.5 Inline assembly template modifiers B8-325
B8.6 Forcing inline assembly operands into specific registers B8-328
B8.7 Symbol references and branches into and out of inline assembly B8-329
B8.8 Duplication of labels in inline assembly statements B8-330

Part C armlink Reference

Chapter C1 armlink Command-line Options

C1.1 --any_contingency C1-337

C1.2 --any_placement=algorithm C1-338
C1.3 --any_sort_order=order C1-340
C1.4 --api, --no_api C1-341
C1.5 --autoat, --no_autoat C1-342
C1.6 --bare_metal_pie C1-343
C1.7 --base_platform C1-344
C1.8 --bestdebug, --no_bestdebug C1-346
C1.9 --blx_arm_thumb, --no_blx_arm_thumb C1-347
C1.10 --blx_thumb_arm, --no_blx_thumb_arm C1-348
C1.11 --bpabi C1-349
C1.12 --branchnop, --no_branchnop C1-350
C1.13 --callgraph, --no_callgraph C1-351
C1.14 --callgraph_file=filename C1-353
C1.15 --callgraph_output=fmt C1-354
C1.16 --callgraph_subset=symbol[,symbol,...] C1-355
C1.17 --cgfile=type C1-356
C1.18 --cgsymbol=type C1-357
C1.19 --cgundefined=type C1-358
C1.20 --comment_section, --no_comment_section C1-359
C1.21 --cppinit, --no_cppinit C1-360
C1.22 --cpu=list (armlink) C1-361
C1.23 --cpu=name (armlink) C1-362
C1.24 --crosser_veneershare, --no_crosser_veneershare C1-365
C1.25 --datacompressor=opt C1-366
C1.26 --debug, --no_debug C1-367
C1.27 --diag_error=tag[,tag,…] (armlink) C1-368
C1.28 --diag_remark=tag[,tag,…] (armlink) C1-369
C1.29 --diag_style={arm|ide|gnu} (armlink) C1-370
C1.30 --diag_suppress=tag[,tag,…] (armlink) C1-371
C1.31 --diag_warning=tag[,tag,…] (armlink) C1-372
C1.32 --dll C1-373
C1.33 --dynamic_linker=name C1-374
C1.34 --eager_load_debug, --no_eager_load_debug C1-375
C1.35 --eh_frame_hdr C1-376
C1.36 --edit=file_list C1-377
C1.37 --emit_debug_overlay_relocs C1-378
C1.38 --emit_debug_overlay_section C1-379
C1.39 --emit_non_debug_relocs C1-380
C1.40 --emit_relocs C1-381
C1.41 --entry=location C1-382
C1.42 --errors=filename C1-383
C1.43 --exceptions, --no_exceptions C1-384
C1.44 --export_all, --no_export_all C1-385
C1.45 --export_dynamic, --no_export_dynamic C1-386
C1.46 --filtercomment, --no_filtercomment C1-387
C1.47 --fini=symbol C1-388
C1.48 --first=section_id C1-389
C1.49 --force_explicit_attr C1-390
C1.50 --force_so_throw, --no_force_so_throw C1-391
C1.51 --fpic C1-392

C1.52 --fpu=list (armlink) C1-393
C1.53 --fpu=name (armlink) C1-394
C1.54 --got=type C1-395
C1.55 --gnu_linker_defined_syms C1-396
C1.56 --help (armlink) C1-397
C1.57 --import_cmse_lib_in=filename C1-398
C1.58 --import_cmse_lib_out=filename C1-399
C1.59 --import_unresolved, --no_import_unresolved C1-400
C1.60 --info=topic[,topic,…] (armlink) C1-401
C1.61 --info_lib_prefix=opt C1-404
C1.62 --init=symbol C1-405
C1.63 --inline, --no_inline C1-406
C1.64 --inline_type=type C1-407
C1.65 --inlineveneer, --no_inlineveneer C1-408
C1.66 input-file-list (armlink) C1-409
C1.67 --keep=section_id (armlink) C1-410
C1.68 --keep_intermediate C1-412
C1.69 --largeregions, --no_largeregions C1-413
C1.70 --last=section_id C1-415
C1.71 --legacyalign, --no_legacyalign C1-416
C1.72 --libpath=pathlist C1-417
C1.73 --library=name C1-418
C1.74 --library_security=protection C1-419
C1.75 --library_type=lib C1-421
C1.76 --list=filename C1-422
C1.77 --list_mapping_symbols, --no_list_mapping_symbols C1-423
C1.78 --load_addr_map_info, --no_load_addr_map_info C1-424
C1.79 --locals, --no_locals C1-425
C1.80 --lto, --no_lto C1-426
C1.81 --lto_keep_all_symbols, --no_lto_keep_all_symbols C1-428
C1.82 --lto_intermediate_filename C1-429
C1.83 --lto_level C1-430
C1.84 --lto_relocation_model C1-432
C1.85 --mangled, --unmangled C1-433
C1.86 --map, --no_map C1-434
C1.87 --max_er_extension=size C1-435
C1.88 --max_veneer_passes=value C1-436
C1.89 --max_visibility=type C1-437
C1.90 --merge, --no_merge C1-438
C1.91 --merge_litpools, --no_merge_litpools C1-439
C1.92 --muldefweak, --no_muldefweak C1-440
C1.93 -o filename, --output=filename (armlink) C1-441
C1.94 --output_float_abi=option C1-442
C1.95 --overlay_veneers C1-443
C1.96 --override_visibility C1-444
C1.97 -Omax (armlink) C1-445
C1.98 --pad=num C1-446
C1.99 --paged C1-447
C1.100 --pagesize=pagesize C1-448
C1.101 --partial C1-449

C1.102 --pie C1-450
C1.103 --piveneer, --no_piveneer C1-451
C1.104 --pixolib C1-452
C1.105 --pltgot=type C1-454
C1.106 --pltgot_opts=mode C1-455
C1.107 --predefine="string" C1-456
C1.108 --preinit, --no_preinit C1-457
C1.109 --privacy (armlink) C1-458
C1.110 --ref_cpp_init, --no_ref_cpp_init C1-459
C1.111 --ref_pre_init, --no_ref_pre_init C1-460
C1.112 --reloc C1-461
C1.113 --remarks C1-462
C1.114 --remove, --no_remove C1-463
C1.115 --ro_base=address C1-464
C1.116 --ropi C1-465
C1.117 --rosplit C1-466
C1.118 --rw_base=address C1-467
C1.119 --rwpi C1-468
C1.120 --scanlib, --no_scanlib C1-469
C1.121 --scatter=filename C1-470
C1.122 --section_index_display=type C1-472
C1.123 --shared C1-473
C1.124 --show_cmdline (armlink) C1-474
C1.125 --show_full_path C1-475
C1.126 --show_parent_lib C1-476
C1.127 --show_sec_idx C1-477
C1.128 --soname=name C1-478
C1.129 --sort=algorithm C1-479
C1.130 --split C1-481
C1.131 --startup=symbol, --no_startup C1-482
C1.132 --stdlib C1-483
C1.133 --strict C1-484
C1.134 --strict_flags, --no_strict_flags C1-485
C1.135 --strict_ph, --no_strict_ph C1-486
C1.136 --strict_preserve8_require8 .... C1-487
C1.137 --strict_relocations, --no_strict_relocations C1-488
C1.138 --strict_symbols, --no_strict_symbols C1-489
C1.139 --strict_visibility, --no_strict_visibility C1-490
C1.140 --symbols, --no_symbols C1-491
C1.141 --symdefs=filename C1-492
C1.142 --symver_script=filename C1-493
C1.143 --symver_soname C1-494
C1.144 --sysv C1-495
C1.145 --tailreorder, --no_tailreorder C1-496
C1.146 --tiebreaker=option C1-497
C1.147 --unaligned_access, --no_unaligned_access C1-498
C1.148 --undefined=symbol C1-499
C1.149 --undefined_and_export=symbol C1-500
C1.150 --unresolved=symbol C1-501
C1.151 --use_definition_visibility C1-502

C1.152 --userlibpath=pathlist C1-503
C1.153 --veneerinject, --no_veneerinject C1-504
C1.154 --veneer_inject_type=type C1-505
C1.155 --veneer_pool_size=size C1-506
C1.156 --veneershare, --no_veneershare C1-507
C1.157 --verbose C1-508
C1.158 --version_number (armlink) C1-509
C1.159 --via=filename (armlink) C1-510
C1.160 --vsn (armlink) C1-511
C1.161 --xo_base=address C1-512
C1.162 --xref, --no_xref C1-513
C1.163 --xrefdbg, --no_xrefdbg C1-514
C1.164 --xref{from|to}=object(section) C1-515
C1.165 --zi_base=address C1-516

Chapter C2 Linking Models Supported by armlink

C2.1 Overview of linking models C2-518
C2.2 Bare-metal linking model overview C2-519
C2.3 Partial linking model overview C2-520
C2.4 Base Platform Application Binary Interface (BPABI) linking model overview C2-521
C2.5 Base Platform linking model overview C2-522
C2.6 SysV linking model overview C2-524
C2.7 Concepts common to both BPABI and SysV linking models C2-525

Chapter C3 Image Structure and Generation

C3.1 The structure of an Arm® ELF image C3-528
C3.2 Simple images C3-536
C3.3 Section placement with the linker C3-543
C3.4 Linker support for creating demand-paged files C3-546
C3.5 Linker reordering of execution regions containing T32 code C3-547
C3.6 Linker-generated veneers C3-548
C3.7 Command-line options used to control the generation of C++ exception tables C3-552
C3.8 Weak references and definitions C3-553
C3.9 How the linker performs library searching, selection, and scanning C3-555
C3.10 How the linker searches for the Arm® standard libraries C3-556
C3.11 Specifying user libraries when linking C3-557
C3.12 How the linker resolves references C3-558
C3.13 The strict family of linker options C3-559

Chapter C4 Linker Optimization Features

C4.1 Elimination of common section groups C4-562
C4.2 Elimination of unused sections C4-563
C4.3 Optimization with RW data compression C4-564
C4.4 Function inlining with the linker C4-567
C4.5 Factors that influence function inlining C4-568
C4.6 About branches that optimize to a NOP C4-570
C4.7 Linker reordering of tail calling sections C4-571
C4.8 Restrictions on reordering of tail calling sections C4-572
C4.9 Linker merging of comment sections C4-573
C4.10 Merging identical constants C4-574

Chapter C5 Accessing and Managing Symbols with armlink

C5.1 About mapping symbols C5-578
C5.2 Linker-defined symbols C5-579
C5.3 Region-related symbols C5-580
C5.4 Section-related symbols C5-585
C5.5 Access symbols in another image C5-587
C5.6 Edit the symbol tables with a steering file C5-590
C5.7 Use of $Super$$ and $Sub$$ to patch symbol definitions C5-593

Chapter C6 Scatter-loading Features

C6.1 The scatter-loading mechanism C6-596
C6.2 Root region and the initial entry point C6-602
C6.3 Example of how to explicitly place a named section with scatter-loading C6-617
C6.4 Placement of unassigned sections C6-619
C6.5 Placing veneers with a scatter file C6-630
C6.6 Placement of CMSE veneer sections for a Secure image C6-631
C6.7 Reserving an empty block of memory C6-633
C6.8 Placement of Arm® C and C++ library code C6-635
C6.9 Aligning regions to page boundaries C6-638
C6.10 Aligning execution regions and input sections C6-639
C6.11 Preprocessing a scatter file C6-640
C6.12 Example of using expression evaluation in a scatter file to avoid padding C6-642
C6.13 Equivalent scatter-loading descriptions for simple images C6-643
C6.14 How the linker resolves multiple matches when processing scatter files C6-650
C6.15 How the linker resolves path names when processing scatter files C6-652
C6.16 Scatter file to ELF mapping C6-653

Chapter C7 Scatter File Syntax

C7.1 BNF notation used in scatter-loading description syntax C7-656
C7.2 Syntax of a scatter file C7-657
C7.3 Load region descriptions C7-658
C7.4 Execution region descriptions C7-664
C7.5 Input section descriptions C7-672
C7.6 Expression evaluation in scatter files C7-677

Chapter C8 BPABI and SysV Shared Libraries and Executables

C8.1 About the Base Platform Application Binary Interface (BPABI) C8-686
C8.2 Platforms supported by the BPABI C8-687
C8.3 Features common to all BPABI models C8-688
C8.4 SysV linking model C8-691
C8.5 Bare metal and DLL-like memory models C8-697
C8.6 Symbol versioning C8-702

Chapter C9 Features of the Base Platform Linking Model

C9.1 Restrictions on the use of scatter files with the Base Platform model C9-706
C9.2 Scatter files for the Base Platform linking model C9-708
C9.3 Placement of PLT sequences with the Base Platform model C9-710

Chapter C10 Linker Steering File Command Reference

C10.1 EXPORT steering file command C10-712

C10.2 HIDE steering file command C10-713
C10.3 IMPORT steering file command C10-714
C10.4 RENAME steering file command C10-715
C10.5 REQUIRE steering file command C10-716
C10.6 RESOLVE steering file command C10-717
C10.7 SHOW steering file command C10-719

Part D fromelf Reference

Chapter D1 fromelf Command-line Options

D1.1 --base [[object_file::]load_region_ID=]num D1-725
D1.2 --bin D1-727
D1.3 --bincombined D1-728
D1.4 --bincombined_base=address D1-729
D1.5 --bincombined_padding=size,num D1-730
D1.6 --cad D1-731
D1.7 --cadcombined D1-733
D1.8 --compare=option[,option,…] D1-734
D1.9 --continue_on_error D1-736
D1.10 --cpu=list (fromelf) D1-737
D1.11 --cpu=name (fromelf) D1-738
D1.12 --datasymbols D1-741
D1.13 --debugonly D1-742
D1.14 --decode_build_attributes D1-743
D1.15 --diag_error=tag[,tag,…] (fromelf) D1-745
D1.16 --diag_remark=tag[,tag,…] (fromelf) D1-746
D1.17 --diag_style={arm|ide|gnu} (fromelf) D1-747
D1.18 --diag_suppress=tag[,tag,…] (fromelf) D1-748
D1.19 --diag_warning=tag[,tag,…] (fromelf) D1-749
D1.20 --disassemble D1-750
D1.21 --dump_build_attributes D1-751
D1.22 --elf D1-752
D1.23 --emit=option[,option,…] D1-753
D1.24 --expandarrays D1-755
D1.25 --extract_build_attributes D1-756
D1.26 --fieldoffsets D1-757
D1.27 --fpu=list (fromelf) D1-759
D1.28 --fpu=name (fromelf) D1-760
D1.29 --globalize=option[,option,…] D1-761
D1.30 --help (fromelf) D1-762
D1.31 --hide=option[,option,…] D1-763
D1.32 --hide_and_localize=option[,option,…] D1-764
D1.33 --i32 .... D1-765
D1.34 --i32combined D1-766
D1.35 --ignore_section=option[,option,…] D1-767
D1.36 --ignore_symbol=option[,option,…] D1-768
D1.37 --in_place D1-769
D1.38 --info=topic[,topic,…] (fromelf) D1-770
D1.39 input_file (fromelf) D1-771
D1.40 --interleave=option D1-773

D1.41 --linkview, --no_linkview D1-774
D1.42 --localize=option[,option,…] D1-775
D1.43 --m32 .... D1-776
D1.44 --m32combined D1-777
D1.45 --only=section_name D1-778
D1.46 --output=destination D1-779
D1.47 --privacy (fromelf) D1-780
D1.48 --qualify D1-781
D1.49 --relax_section=option[,option,…] D1-782
D1.50 --relax_symbol=option[,option,…] D1-783
D1.51 --rename=option[,option,…] D1-784
D1.52 --select=select_options D1-785
D1.53 --show=option[,option,…] D1-786
D1.54 --show_and_globalize=option[,option,…] D1-787
D1.55 --show_cmdline (fromelf) D1-788
D1.56 --source_directory=path D1-789
D1.57 --strip=option[,option,…] D1-790
D1.58 --symbolversions, --no_symbolversions D1-792
D1.59 --text D1-793
D1.60 --version_number (fromelf) D1-795
D1.61 --vhx D1-796
D1.62 --via=file (fromelf) D1-797
D1.63 --vsn (fromelf) D1-798
D1.64 -w D1-799
D1.65 --wide64bit D1-800
D1.66 --widthxbanks D1-801

Part E armar Reference

Chapter E1 armar Command-line Options

E1.1 archive E1-807
E1.2 -a pos_name E1-808
E1.3 -b pos_name E1-809
E1.4 -c (armar) E1-810
E1.5 -C (armar) E1-811
E1.6 --create E1-812
E1.7 -d E1-813
E1.8 --debug_symbols E1-814
E1.9 --diag_error=tag[,tag,…] (armar) E1-815
E1.10 --diag_remark=tag[,tag,…] (armar) E1-816
E1.11 --diag_style={arm|ide|gnu} (armar) E1-817
E1.12 --diag_suppress=tag[,tag,…] (armar) E1-818
E1.13 --diag_warning=tag[,tag,…] (armar) E1-819
E1.14 --entries E1-820
E1.15 file_list E1-821
E1.16 --help (armar) E1-822
E1.17 -i pos_name E1-823
E1.18 -m pos_name (armar) E1-824
E1.19 -n E1-825
E1.20 --new_files_only E1-826

E1.21 -p ... E1-827
E1.22 -r ... E1-828
E1.23 -s ... E1-829
E1.24 --show_cmdline (armar) ... E1-830
E1.25 --sizes ... E1-831
E1.26 -t ... E1-832
E1.27 -T ... E1-833
E1.28 -u (armar) ... E1-834
E1.29 -v (armar) ... E1-835
E1.30 --version_number (armar) ... E1-836
E1.31 --via=filename (armar) ... E1-837
E1.32 --vsn (armar) ... E1-838
E1.33 -x (armar) ... E1-839
E1.34 --zs ... E1-840
E1.35 --zt ... E1-841

Part F armasm Legacy Assembler Reference

Chapter F1 armasm Command-line Options

F1.1 --16 ... F1-847
F1.2 --32 ... F1-848
F1.3 --apcs=qualifier…qualifier ... F1-849
F1.4 --arm ... F1-851
F1.5 --arm_only ... F1-852
F1.6 --bi ... F1-853
F1.7 --bigend ... F1-854
F1.8 --brief_diagnostics, --no_brief_diagnostics ... F1-855
F1.9 --checkreglist ... F1-856
F1.10 --cpreproc ... F1-857
F1.11 --cpreproc_opts=option[,option,…] ... F1-858
F1.12 --cpu=list (armasm) ... F1-859
F1.13 --cpu=name (armasm) ... F1-860
F1.14 --debug ... F1-863
F1.15 --depend=dependfile ... F1-864
F1.16 --depend_format=string ... F1-865
F1.17 --diag_error=tag[,tag,…] (armasm) ... F1-866
F1.18 --diag_remark=tag[,tag,…] (armasm) ... F1-867
F1.19 --diag_style={arm|ide|gnu} (armasm) ... F1-868
F1.20 --diag_suppress=tag[,tag,…] (armasm) ... F1-869
F1.21 --diag_warning=tag[,tag,…] (armasm) ... F1-870
F1.22 --dllexport_all ... F1-871
F1.23 --dwarf2 ... F1-872
F1.24 --dwarf3 ... F1-873
F1.25 --errors=errorfile ... F1-874
F1.26 --exceptions, --no_exceptions ... F1-875
F1.27 --exceptions_unwind, --no_exceptions_unwind ... F1-876
F1.28 --execstack, --no_execstack ... F1-877
F1.29 --execute_only ... F1-878
F1.30 --fpmode=model ... F1-879
F1.31 --fpu=list (armasm) ... F1-880

F1.32 --fpu=name (armasm) F1-881
F1.33 -g (armasm) F1-882
F1.34 --help (armasm) F1-883
F1.35 -idir[,dir, …] F1-884
F1.36 --keep (armasm) F1-885
F1.37 --length=n F1-886
F1.38 --li F1-887
F1.39 --library_type=lib F1-888
F1.40 --list=file F1-889
F1.41 --list= F1-890
F1.42 --littleend F1-891
F1.43 -m (armasm) F1-892
F1.44 --maxcache=n F1-893
F1.45 --md F1-894
F1.46 --no_code_gen F1-895
F1.47 --no_esc F1-896
F1.48 --no_hide_all F1-897
F1.49 --no_regs F1-898
F1.50 --no_terse F1-899
F1.51 --no_warn F1-900
F1.52 -o filename (armasm) F1-901
F1.53 --pd F1-902
F1.54 --predefine "directive" F1-903
F1.55 --reduce_paths, --no_reduce_paths F1-904
F1.56 --regnames F1-905
F1.57 --report-if-not-wysiwyg F1-906
F1.58 --show_cmdline (armasm) F1-907
F1.59 --thumb F1-908
F1.60 --unaligned_access, --no_unaligned_access F1-909
F1.61 --unsafe F1-910
F1.62 --untyped_local_labels F1-911
F1.63 --version_number (armasm) F1-912
F1.64 --via=filename (armasm) F1-913
F1.65 --vsn (armasm) F1-914
F1.66 --width=n F1-915
F1.67 --xref F1-916

Chapter F2 Structure of armasm Assembly Language Modules
F2.1 Syntax of source lines in armasm syntax assembly language F2-918
F2.2 Literals F2-920
F2.3 ELF sections and the AREA directive F2-921
F2.4 An example armasm syntax assembly language module F2-922

Chapter F3 Writing A32/T32 Instructions in armasm Syntax Assembly Language
F3.1 About the Unified Assembler Language F3-927
F3.2 Syntax differences between UAL and A64 assembly language F3-928
F3.3 Register usage in subroutine calls F3-929
F3.4 Load immediate values F3-930
F3.5 Load immediate values using MOV and MVN F3-931
F3.6 Load immediate values using MOV32 .... F3-934

F3.7 Load immediate values using LDR Rd, =const F3-935
F3.8 Literal pools F3-936
F3.9 Load addresses into registers F3-937
F3.10 Load addresses to a register using ADR F3-938
F3.11 Load addresses to a register using ADRL F3-940
F3.12 Load addresses to a register using LDR Rd, =label F3-941
F3.13 Other ways to load and store registers F3-943
F3.14 Load and store multiple register instructions F3-944
F3.15 Load and store multiple register instructions in A32 and T32 .... F3-945
F3.16 Stack implementation using LDM and STM F3-946
F3.17 Stack operations for nested subroutines F3-948
F3.18 Block copy with LDM and STM F3-949
F3.19 Memory accesses F3-951
F3.20 The Read-Modify-Write operation F3-952
F3.21 Optional hash with immediate constants F3-953
F3.22 Use of macros F3-954
F3.23 Test-and-branch macro example F3-955
F3.24 Unsigned integer division macro example F3-956
F3.25 Instruction and directive relocations F3-958
F3.26 Symbol versions F3-960
F3.27 Frame directives F3-961
F3.28 Exception tables and Unwind tables F3-962

Chapter F4 Using armasm

F4.1 armasm command-line syntax F4-964
F4.2 Specify command-line options with an environment variable F4-965
F4.3 Using stdin to input source code to the assembler F4-966
F4.4 Built-in variables and constants F4-967
F4.5 Identifying versions of armasm in source code F4-971
F4.6 Diagnostic messages F4-972
F4.7 Interlocks diagnostics F4-973
F4.8 Automatic IT block generation in T32 code F4-974
F4.9 T32 branch target alignment F4-975
F4.10 T32 code size diagnostics F4-976
F4.11 A32 and T32 instruction portability diagnostics F4-977
F4.12 T32 instruction width diagnostics F4-978
F4.13 Two pass assembler diagnostics F4-979
F4.14 Using the C preprocessor F4-980
F4.15 Address alignment in A32/T32 code F4-982
F4.16 Address alignment in A64 code F4-983
F4.17 Instruction width selection in T32 code F4-984

Chapter F5 Symbols, Literals, Expressions, and Operators in armasm Assembly Language

F5.1 Symbol naming rules F5-987
F5.2 Variables F5-988
F5.3 Numeric constants F5-989
F5.4 Assembly time substitution of variables F5-990
F5.5 Register-relative and PC-relative expressions F5-991
F5.6 Labels F5-992

F5.7 Labels for PC-relative addresses F5-993
F5.8 Labels for register-relative addresses F5-994
F5.9 Labels for absolute addresses F5-995
F5.10 Numeric local labels F5-996
F5.11 Syntax of numeric local labels F5-997
F5.12 String expressions F5-998
F5.13 String literals F5-999
F5.14 Numeric expressions F5-1000
F5.15 Syntax of numeric literals F5-1001
F5.16 Syntax of floating-point literals F5-1002
F5.17 Logical expressions F5-1003
F5.18 Logical literals F5-1004
F5.19 Unary operators F5-1005
F5.20 Binary operators F5-1006
F5.21 Multiplicative operators F5-1007
F5.22 String manipulation operators F5-1008
F5.23 Shift operators F5-1009
F5.24 Addition, subtraction, and logical operators F5-1010
F5.25 Relational operators F5-1011
F5.26 Boolean operators F5-1012
F5.27 Operator precedence F5-1013
F5.28 Difference between operator precedence in assembly language and C F5-1014

Chapter F6 armasm Directives Reference

F6.1 Alphabetical list of directives armasm assembly language directives F6-1019
F6.2 About armasm assembly language control directives F6-1020
F6.3 About frame directives F6-1021
F6.4 Directives that can be omitted in pass 2 of the assembler F6-1022
F6.5 ALIAS F6-1024
F6.6 ALIGN F6-1025
F6.7 AREA F6-1027
F6.8 ARM or CODE32 directive F6-1031
F6.9 ASSERT F6-1032
F6.10 ATTR F6-1033
F6.11 CN F6-1034
F6.12 CODE16 directive F6-1035
F6.13 COMMON F6-1036
F6.14 CP F6-1037
F6.15 DATA F6-1038
F6.16 DCB F6-1039
F6.17 DCD and DCDU F6-1040
F6.18 DCDO F6-1041
F6.19 DCFD and DCFDU F6-1042
F6.20 DCFS and DCFSU F6-1043
F6.21 DCI F6-1044
F6.22 DCQ and DCQU F6-1045
F6.23 DCW and DCWU F6-1046
F6.24 END F6-1047
F6.25 ENDFUNC or ENDP F6-1048
F6.26 ENTRY F6-1049

F6.27 EQU F6-1050
F6.28 EXPORT or GLOBAL F6-1051
F6.29 EXPORTAS F6-1053
F6.30 FIELD F6-1054
F6.31 FRAME ADDRESS F6-1055
F6.32 FRAME POP F6-1056
F6.33 FRAME PUSH F6-1057
F6.34 FRAME REGISTER F6-1058
F6.35 FRAME RESTORE F6-1059
F6.36 FRAME RETURN ADDRESS F6-1060
F6.37 FRAME SAVE F6-1061
F6.38 FRAME STATE REMEMBER F6-1062
F6.39 FRAME STATE RESTORE F6-1063
F6.40 FRAME UNWIND ON F6-1064
F6.41 FRAME UNWIND OFF F6-1065
F6.42 FUNCTION or PROC F6-1066
F6.43 GBLA, GBLL, and GBLS F6-1067
F6.44 GET or INCLUDE F6-1068
F6.45 IF, ELSE, ENDIF, and ELIF F6-1069
F6.46 IMPORT and EXTERN F6-1071
F6.47 INCBIN F6-1073
F6.48 INFO F6-1074
F6.49 KEEP F6-1075
F6.50 LCLA, LCLL, and LCLS F6-1076
F6.51 LTORG F6-1077
F6.52 MACRO and MEND F6-1078
F6.53 MAP F6-1081
F6.54 MEXIT F6-1082
F6.55 NOFP F6-1083
F6.56 OPT F6-1084
F6.57 QN, DN, and SN F6-1086
F6.58 RELOC F6-1088
F6.59 REQUIRE F6-1089
F6.60 REQUIRE8 and PRESERVE8 .... F6-1090
F6.61 RLIST F6-1091
F6.62 RN F6-1092
F6.63 ROUT F6-1093
F6.64 SETA, SETL, and SETS F6-1094
F6.65 SPACE or FILL F6-1096
F6.66 THUMB directive F6-1097
F6.67 TTL and SUBT F6-1098
F6.68 WHILE and WEND F6-1099
F6.69 WN and XN F6-1100

Chapter F7 armasm-Specific A32 and T32 Instruction Set Features

F7.1 armasm support for the CSDB instruction F7-1102
F7.2 A32 and T32 pseudo-instruction summary F7-1103
F7.3 ADRL pseudo-instruction F7-1104
F7.4 CPY pseudo-instruction F7-1106
F7.5 LDR pseudo-instruction F7-1107

F7.6 MOV32 pseudo-instruction F7-1109
F7.7 NEG pseudo-instruction F7-1110
F7.8 UND pseudo-instruction F7-1111

Part G Appendixes

Appendix A Standard C Implementation Definition

A.1 Implementation definition Appx-A-1116
A.2 Translation Appx-A-1117
A.3 Translation limits Appx-A-1118
A.4 Environment Appx-A-1120
A.5 Identifiers Appx-A-1122
A.6 Characters Appx-A-1123
A.7 Integers Appx-A-1125
A.8 Floating-point Appx-A-1126
A.9 Arrays and pointers Appx-A-1127
A.10 Hints Appx-A-1128
A.11 Structures, unions, enumerations, and bitfields Appx-A-1129
A.12 Qualifiers Appx-A-1130
A.13 Preprocessing directives Appx-A-1131
A.14 Library functions Appx-A-1133
A.15 Architecture Appx-A-1138

Appendix B Standard C++ Implementation Definition

B.1 Implementation definition Appx-B-1146
B.2 General Appx-B-1147
B.3 Lexical conventions Appx-B-1148
B.4 Basic concepts Appx-B-1149
B.5 Standard conversions Appx-B-1150
B.6 Expressions Appx-B-1151
B.7 Declarations Appx-B-1153
B.8 Declarators Appx-B-1154
B.9 Templates Appx-B-1155
B.10 Exception handling Appx-B-1156
B.11 Preprocessing directives Appx-B-1157
B.12 Library introduction Appx-B-1158
B.13 Language support library Appx-B-1159
B.14 General utilities library Appx-B-1160
B.15 Strings library Appx-B-1161
B.16 Localization library Appx-B-1162
B.17 Containers library Appx-B-1163
B.18 Input/output library Appx-B-1164
B.19 Regular expressions library Appx-B-1165
B.20 Atomic operations library Appx-B-1166
B.21 Thread support library Appx-B-1167
B.22 Implementation quantities Appx-B-1168

Appendix C Via File Syntax

C.1 Overview of via files Appx-C-1172
C.2 Via file syntax rules Appx-C-1173

List of Figures
Arm® Compiler Reference Guide

Figure A1-1 Integration boundaries in Arm Compiler 6. .... A1-41
Figure B5-1 Nonpacked structure S B5-255
Figure B5-2 Packed structure SP B5-255
Figure B6-1 IEEE half-precision floating-point format B6-269
Figure B6-2 BFloat16 floating-point format B6-272
Figure C3-1 Relationship between sections, regions, and segments C3-529
Figure C3-2 Load and execution memory maps for an image without an XO section C3-531
Figure C3-3 Load and execution memory maps for an image with an XO section C3-531
Figure C3-4 Simple Type 1 image C3-537
Figure C3-5 Simple Type 2 image C3-539
Figure C3-6 Simple Type 3 image C3-541
Figure C6-1 Simple scatter-loaded memory map C6-599
Figure C6-2 Complex memory map C6-600
Figure C6-3 Memory map for fixed execution regions C6-604
Figure C6-4 .ANY contingency C6-627
Figure C6-5 Reserving a region for the stack C6-634
Figure C7-1 Components of a scatter file C7-657
Figure C7-2 Components of a load region description C7-658
Figure C7-3 Components of an execution region description C7-664
Figure C7-4 Components of an input section description C7-672
Figure C8-1 BPABI tool flow C8-686

List of Tables
Arm® Compiler Reference Guide

Table B1-1 armclang command-line options B1-48
Table B1-2 Floating-point library variants B1-64
Table B1-3 Floating-point library variant selection B1-68
Table B1-4 Translatable options B1-116
Table B1-5 Cryptographic Extensions B1-128
Table B1-6 Floating-point extensions B1-129
Table B1-7 Options for the Matrix Multiplication extension B1-129
Table B1-8 Options for the MVE extension B1-130
Table B1-9 Comparison of disassembly for -moutline with and without -O1 optimization B1-141
Table B1-10 Comparison of disassembly for -moutline with and without -O2 optimization and with func3 enabled B1-142
Table B1-11 Compiling without the -o option B1-149
Table B3-1 Function attributes that the compiler supports, and their equivalents B3-193
Table B4-1 Modifying the FPSCR flags B4-246
Table B6-1 Predefined macros B6-261
Table B6-2 Non-secure function pointer intrinsics B6-276
Table B7-1 Modifiers B7-280
Table B7-2 Unary operators B7-281
Table B7-3 Binary operators B7-281
Table B7-4 Binary logical operators B7-281
Table B7-5 Binary bitwise operators B7-281
Table B7-6 Binary comparison operators B7-282
Table B7-7 Relocation specifiers for AArch32 state B7-282

Table B7-8 Relocation specifiers for AArch64 state B7-283
Table B7-9 Data definition directives B7-287
Table B7-10 Expression types supported by the data definition directives B7-288
Table B7-11 Aliases for the data definition directives B7-288
Table B7-12 Escape characters for the string definition directives B7-290
Table B7-13 Aliases for the floating-point data definition directives B7-292
Table B7-14 Section flags B7-294
Table B7-15 Section Type B7-295
Table B7-16 Sections with implicit flags and default types B7-296
Table B7-17 .if condition modifiers B7-299
Table B7-18 Macro parameter qualifier B7-301
Table B8-1 Constraint modifiers B8-320
Table C1-1 Supported Arm architectures C1-362
Table C1-2 Data compressor algorithms C1-366
Table C1-3 GNU equivalent of input sections C1-396
Table C1-4 Link time optimization dependencies C1-426
Table C3-1 Comparing load and execution views C3-531
Table C3-2 Comparison of scatter file and equivalent command-line options C3-532
Table C4-1 Inlining small functions C4-568
Table C5-1 Image$$ execution region symbols C5-580
Table C5-2 Load$$ execution region symbols C5-581
Table C5-3 Load$$LR$$ load region symbols C5-582
Table C5-4 Image symbols C5-585
Table C5-5 Section-related symbols C5-586
Table C5-6 Steering file command summary C5-590
Table C6-1 Input section properties for placement of .ANY sections C6-622
Table C6-2 Input section properties for placement of sections with next_fit C6-624
Table C6-3 Input section properties and ordering for sections_a.o and sections_b.o C6-626
Table C6-4 Sort order for descending_size algorithm C6-626
Table C6-5 Sort order for cmdline algorithm C6-626
Table C7-1 BNF notation C7-656
Table C7-2 Execution address related functions C7-679
Table C7-3 Load address related functions C7-680
Table C8-1 Symbol visibility C8-689
Table C8-2 Turning on BPABI support C8-698
Table D1-1 Examples of using --base D1-725
Table D1-2 Supported Arm architectures D1-738
Table F1-1 Supported Arm architectures F1-860
Table F1-2 Severity of diagnostic messages F1-866
Table F1-3 Specifying a command-line option and an AREA directive for GNU-stack sections F1-877
Table F3-1 Syntax differences between UAL and A64 assembly language F3-928
Table F3-2 A32 state immediate values (8-bit) F3-931
Table F3-3 A32 state immediate values in MOV instructions F3-931
Table F3-4 32-bit T32 immediate values F3-932
Table F3-5 32-bit T32 immediate values in MOV instructions F3-932
Table F3-6 Stack-oriented suffixes and equivalent addressing mode suffixes F3-946
Table F3-7 Suffixes for load and store multiple instructions F3-946
Table F4-1 Built-in variables F4-967
Table F4-2 Built-in Boolean constants F4-968
Table F4-3 Predefined macros F4-968

Table F4-4 armclang equivalent command-line options F4-980
Table F5-1 Unary operators that return strings F5-1005
Table F5-2 Unary operators that return numeric or logical values F5-1005
Table F5-3 Multiplicative operators F5-1007
Table F5-4 String manipulation operators F5-1008
Table F5-5 Shift operators F5-1009
Table F5-6 Addition, subtraction, and logical operators F5-1010
Table F5-7 Relational operators F5-1011
Table F5-8 Boolean operators F5-1012
Table F5-9 Operator precedence in Arm assembly language F5-1014
Table F5-10 Operator precedence in C F5-1014
Table F6-1 List of directives F6-1019
Table F6-2 OPT directive settings F6-1084
Table F7-1 Summary of pseudo-instructions F7-1103
Table F7-2 Range and encoding of expr F7-1111
Table A-1 Translation limits Appx-A-1118

Preface

This preface introduces the *Arm® Compiler Reference Guide*.

It contains the following:
- *About this book* on page 30.

About this book

The Arm® Compiler Reference Guide provides reference information for the Arm Compiler toolchain. This document contains separate parts, that provide reference information for each tool in the Arm Compiler toolchain.

Using this book

This book is organized into the following chapters:

Part A Arm Compiler Tools Overview

Chapter A1 Overview of the Arm® Compiler tools

Arm Compiler 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.

Part B armclang Reference

Chapter B1 armclang Command-line Options

This chapter summarizes the supported options used with `armclang`.

Chapter B2 Compiler-specific Keywords and Operators

Summarizes the compiler-specific keywords and operators that are extensions to the C and C++ Standards.

Chapter B3 Compiler-specific Function, Variable, and Type Attributes

Summarizes the compiler-specific function, variable, and type attributes that are extensions to the C and C++ Standards.

Chapter B4 Compiler-specific Intrinsics

Summarizes the Arm compiler-specific intrinsics that are extensions to the C and C++ Standards.

Chapter B5 Compiler-specific Pragmas

Summarizes the Arm compiler-specific pragmas that are extensions to the C and C++ Standards.

Chapter B6 Other Compiler-specific Features

Summarizes compiler-specific features that are extensions to the C and C++ Standards, such as predefined macros.

Chapter B7 armclang Integrated Assembler

Provides information on integrated assembler features, such as the directives you can use when writing assembly language source files in the `armclang` integrated assembler syntax.

Chapter B8 armclang Inline Assembler

Provides reference information on writing inline assembly.

Part C armlink Reference

Chapter C1 armlink Command-line Options

Describes the command-line options supported by the Arm linker, `armlink`.

Chapter C2 Linking Models Supported by armlink

Describes the linking models supported by the Arm linker, `armlink`.

Chapter C3 Image Structure and Generation

Describes the image structure and the functionality available in the Arm linker, `armlink`, to generate images.

Chapter C4 Linker Optimization Features

Describes the optimization features available in the Arm linker, `armlink`.

Chapter C5 Accessing and Managing Symbols with armlink

Describes how to access and manage symbols with the Arm linker, `armlink`.

Chapter C6 Scatter-loading Features

Describes the scatter-loading features and how you use scatter files with the Arm linker, `armlink`, to create complex images.

Chapter C7 Scatter File Syntax

Describes the format of scatter files.

Chapter C8 BPABI and SysV Shared Libraries and Executables

Describes how the Arm linker, `armlink`, supports the *Base Platform Application Binary Interface* (BPABI) and *System V* (SysV) shared libraries and executables.

Chapter C9 Features of the Base Platform Linking Model

Describes features of the Base Platform linking model supported by the Arm linker, `armlink`.

Chapter C10 Linker Steering File Command Reference

Describes the steering file commands supported by the Arm linker, `armlink`.

Part D fromelf Reference

Chapter D1 fromelf Command-line Options

Describes the command-line options of the `fromelf` image converter provided with Arm Compiler.

Part E armar Reference

Chapter E1 armar Command-line Options

Describes the command-line options of the Arm librarian, `armar`.

Part F armasm Legacy Assembler Reference

Chapter F1 armasm Command-line Options

Describes the `armasm` command-line syntax and command-line options.

Chapter F2 Structure of armasm Assembly Language Modules

Describes the structure of `armasm` assembly language source files.

Chapter F3 Writing A32/T32 Instructions in armasm Syntax Assembly Language

Describes the use of a few basic A32 and T32 instructions and the use of macros in the armasm syntax assembly language.

Chapter F4 Using armasm

Describes how to use `armasm`.

Chapter F5 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.

Chapter F6 armasm Directives Reference

Describes the directives that are provided by the Arm assembler, `armasm`.

Chapter F7 armasm-Specific A32 and T32 Instruction Set Features

Describes the additional support that `armasm` provides for the Arm instruction set.

Part G Appendixes

Appendix A Standard C Implementation Definition

Provides information required by the ISO C standard for conforming C implementations.

Appendix B Standard C++ Implementation Definition

Provides information required by the ISO C++ Standard for conforming C++ implementations.

Appendix C Via File Syntax

Describes the syntax of via files accepted by the `armasm`, `armlink`, `fromelf`, and `armar` tools.

Glossary

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

See the *Arm® Glossary* for more information.

Typographic conventions

italic

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

bold

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

`monospace`

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

`monospace`

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

`monospace italic`

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

`monospace bold`

Denotes language keywords when used outside example code.

`<and>`

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

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

SMALL CAPITALS

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

Feedback

Feedback on this product

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

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

Feedback on content

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

- The title *Arm Compiler Reference Guide*.
- The number 101754_0613_00_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

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

Other information

- *Arm® Developer*.
- *Arm® Information Center*.
- *Arm® Technical Support Knowledge Articles*.
- *Technical Support*.
- *Arm® Glossary*.

Part A
Arm Compiler Tools Overview

Chapter A1

Overview of the Arm® Compiler tools

Arm Compiler 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.

It contains the following sections:

- *A1.1 Arm® Compiler tool command-line syntax* on page A1-38.
- *A1.2 Support level definitions* on page A1-39.

A1.1 Arm® Compiler tool command-line syntax

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

The command for invoking a tool is:

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

`tool_name options input-file-list`

For the `armar` tool:

`armar options archive [file_list]`

where:

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

`options`
The tool command-line options.

`input-file-list`

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

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

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

`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.

`file_list`
The list of files to be processed.

Related references

Chapter B1 armclang Command-line Options on page B1-45
Chapter F1 armasm Command-line Options on page F1-845
C1.66 input-file-list (armlink) on page C1-409
Chapter C1 armlink Command-line Options on page C1-333
Chapter D1 fromelf Command-line Options on page D1-723
D1.39 input_file (fromelf) on page D1-771
Chapter E1 armar Command-line Options on page E1-805
E1.1 archive on page E1-807
E1.15 file_list on page E1-821

A1.2 Support level definitions

This describes the levels of support for various Arm Compiler 6 features.

Arm Compiler 6 is built on Clang and LLVM technology. Therefore, it has more functionality than the set of product features described in the documentation. The following definitions clarify the levels of support and guarantees on functionality that are expected from these features.

Arm welcomes feedback regarding the use of all Arm Compiler 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*.

Identification in the documentation

All features that are documented in the Arm Compiler 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.

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 indicated 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 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.

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 indicated 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.

Community features

Arm Compiler 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 that are not listed in the documentation. These additional features are known as community features. For information on these community features, see the *documentation for the Clang/LLVM project*.

Where community features are referenced in the documentation, they are indicated 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 will 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 will 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 6 toolchain:

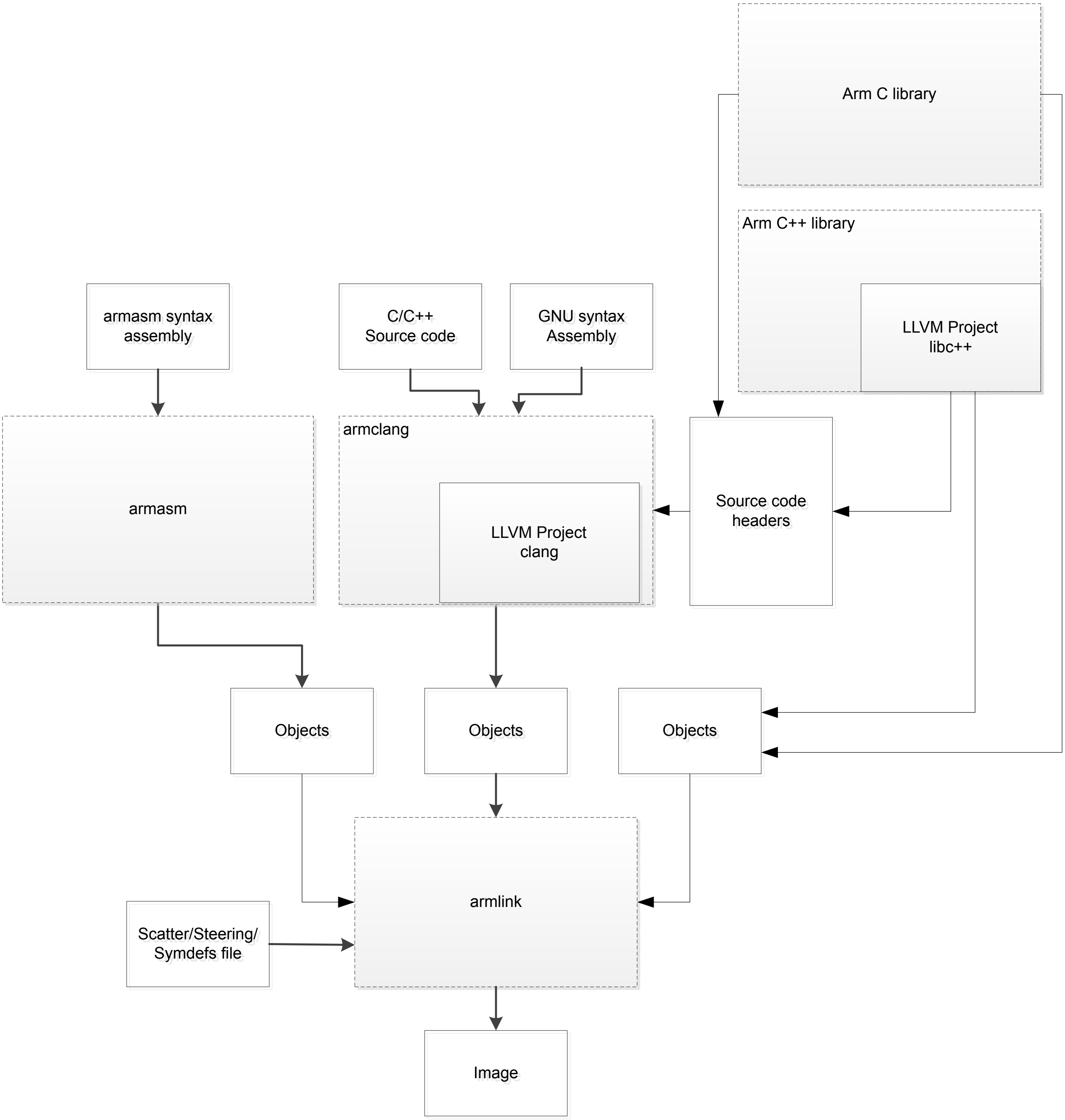


Figure A1-1 Integration boundaries in Arm Compiler 6.

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 6. See *Application Binary Interface (ABI) for the Arm® Architecture*. 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.
- 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. 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, refer to the Arm Compiler documentation and Release Notes.

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 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* on page A1-39.

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.
- 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.
- Compilation for targets that implement architectures older than Armv7 or Armv6-M is not supported.
- The **`long double`** data type is not supported for AArch64 state because of limitations in the current Arm C library.
- Complex numbers are not supported because of limitations in the current Arm C library.

Part B
armclang Reference

Chapter B1
armclang Command-line Options

This chapter summarizes the supported options used with `armclang`.

`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, *`http://llvm.org`*.

Note

Be aware of the following:
- Generated code might be different between two Arm Compiler releases.
- For a feature release, there might be significant code generation differences.

It contains the following sections:

- *B1.1 Summary of armclang command-line options* on page B1-48.
- *B1.2 -C (armclang)* on page B1-54.
- *B1.3 -c (armclang)* on page B1-56.
- *B1.4 -D* on page B1-57.
- *B1.5 -E* on page B1-58.
- *B1.6 -e* on page B1-59.
- *B1.7 -fbare-metal-pie* on page B1-60.
- *B1.8 -fbracket-depth=N* on page B1-61.
- *B1.9 -fcommon, -fno-common* on page B1-62.
- *B1.10 -fdata-sections, -fno-data-sections* on page B1-63.
- *B1.11 -ffast-math, -fno-fast-math* on page B1-64.
- *B1.12 -ffixed-rN* on page B1-65.
- *B1.13 -ffp-mode* on page B1-67.

- *B1.14 -ffunction-sections, -fno-function-sections* on page B1-69.
- *B1.15 -fident, -fno-ident* on page B1-71.
- *B1.16 @file* on page B1-72.
- *B1.17 -fldm-stm, -fno-ldm-stm* on page B1-73.
- *B1.18 -fno-builtin* on page B1-74.
- *B1.19 -fno-inline-functions* on page B1-76.
- *B1.20 -flto, -fno-lto* on page B1-77.
- *B1.21 -fexceptions, -fno-exceptions* on page B1-78.
- *B1.22 -fomit-frame-pointer, -fno-omit-frame-pointer* on page B1-79.
- *B1.23 -fpic, -fno-pic* on page B1-80.
- *B1.24 -fropi, -fno-ropi* on page B1-81.
- *B1.25 -fropi-lowering, -fno-ropi-lowering* on page B1-82.
- *B1.26 -frwpi, -fno-rwpi* on page B1-83.
- *B1.27 -frwpi-lowering, -fno-rwpi-lowering* on page B1-84.
- *B1.28 -fsanitize* on page B1-85.
- *B1.29 -fshort-enums, -fno-short-enums* on page B1-88.
- *B1.30 -fshort-wchar, -fno-short-wchar* on page B1-90.
- *B1.31 -fstack-protector, -fstack-protector-all, -fstack-protector-strong, -fno-stack-protector* on page B1-91.
- *B1.32 -fstrict-aliasing, -fno-strict-aliasing* on page B1-93.
- *B1.33 -fsysv, -fno-sysv* on page B1-94.
- *B1.34 -ftrapv* on page B1-95.
- *B1.35 -fvectorize, -fno-vectorize* on page B1-96.
- *B1.36 -fvisibility* on page B1-97.
- *B1.37 -fwrapv* on page B1-98.
- *B1.38 -g, -gdwarf-2, -gdwarf-3, -gdwarf-4 (armclang)* on page B1-99.
- *B1.39 -I* on page B1-100.
- *B1.40 -include* on page B1-101.
- *B1.41 -L* on page B1-102.
- *B1.42 -l* on page B1-103.
- *B1.43 -M, -MM* on page B1-104.
- *B1.44 -MD, -MMD* on page B1-105.
- *B1.45 -MF* on page B1-106.
- *B1.46 -MG* on page B1-107.
- *B1.47 -MP* on page B1-108.
- *B1.48 -MT* on page B1-109.
- *B1.49 -march* on page B1-110.
- *B1.50 -marm* on page B1-115.
- *B1.51 -masm* on page B1-116.
- *B1.52 -mbig-endian* on page B1-118.
- *B1.53 -mbranch-protection* on page B1-119.
- *B1.54 -mcmodel* on page B1-122.
- *B1.55 -mcmse* on page B1-123.
- *B1.56 -mcpu* on page B1-125.
- *B1.57 -mexecute-only* on page B1-132.
- *B1.58 -mfloat-abi* on page B1-133.
- *B1.59 -mfpu* on page B1-134.
- *B1.60 -mimplicit-it* on page B1-136.
- *B1.61 -mlittle-endian* on page B1-137.
- *B1.62 -mno-neg-immediates* on page B1-138.
- *B1.63 -moutline, -mno-outline* on page B1-140.
- *B1.64 -mpixolib* on page B1-143.
- *B1.65 -munaligned-access, -mno-unaligned-access* on page B1-145.
- *B1.66 -mthumb* on page B1-146.
- *B1.67 -nostdlib* on page B1-147.
- *B1.68 -nostdlibinc* on page B1-148.

- *B1.69 -o (armclang)* on page B1-149.
- *B1.70 -O (armclang)* on page B1-150.
- *B1.71 -pedantic* on page B1-152.
- *B1.72 -pedantic-errors* on page B1-153.
- *B1.73 -Rpass* on page B1-154.
- *B1.74 -S* on page B1-156.
- *B1.75 -save-temps* on page B1-157.
- *B1.76 -shared (armclang)* on page B1-158.
- *B1.77 -std* on page B1-159.
- *B1.78 --target* on page B1-161.
- *B1.79 -U* on page B1-162.
- *B1.80 -u (armclang)* on page B1-163.
- *B1.81 -v (armclang)* on page B1-164.
- *B1.82 --version (armclang)* on page B1-165.
- *B1.83 --version_number (armclang)* on page B1-166.
- *B1.84 --vsn (armclang)* on page B1-167.
- *B1.85 -W* on page B1-168.
- *B1.86 -Wl* on page B1-169.
- *B1.87 -Xlinker* on page B1-170.
- *B1.88 -x (armclang)* on page B1-171.
- *B1.89 -###* on page B1-172.

B1.1 Summary of armclang command-line options

This provides a summary of the `armclang` command-line options that Arm Compiler 6 supports.

Note

This topic includes descriptions of [ALPHA], [BETA], and [COMMUNITY] features. See *Support level definitions* on page A1-39.

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.

Table B1-1 armclang command-line options

Option	Description	Compilation or Assembly
`-C`	Keep comments in the preprocessed output.	Compilation and assembly.
`-c`	Only perform the compile step, do not invoke `armlink`.	Compilation and assembly.
`-D`	Defines a preprocessor macro.	Compilation and assembly.
`-E`	Only perform the preprocess step, do not compile or link.	Compilation and assembly.
`-e`	Specifies the unique initial entry point of the image.	Compilation and assembly.
`-fbare-metal-pie`	Generates position-independent code. AArch32 state only. This option is deprecated.	Compilation only.
`-fbracket-depth`	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.
`-fdata-sections`, `-fno-data-sections`	Enables or disables the generation of one ELF section for each variable in the source file.	Compilation only.

Table B1-1 armclang command-line options (continued)

Option	Description	Compilation or Assembly
`-ffast-math`, `-fno-fast-math`	Enables or disables the use of aggressive floating-point optimizations.	Compilation only.
`-ffixed-r`*`N`*	Prevents the compiler from using the specified core register, unless the use is for Arm ABI compliance.	Compilation only.
`-ffp-mode`	Specifies floating-point standard conformance.	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.
`@file`	Reads a list of command-line options from a file.	Compilation and assembly.
`-fldm-stm`, `-fno-ldm-stm`	Enable or disable the generation of LDM and STM instructions. AArch32 only.	Compilation only.
`-fno-inline-functions`	Disables the automatic inlining of functions at optimization levels `-O2` and `-O3`.	Compilation only.
`-flto`	Enables link time optimization, and outputs bitcode wrapped in an ELF file for link time optimization.	Compilation only.
`-fexceptions`, `-fno-exceptions`	Enables or disables the generation of code needed to support C++ exceptions.	Compilation only.
`-fomit-frame-pointer`, `-fno-omit-frame-pointer`	Enables or disables the storage of stack frame pointers during function calls.	Compilation only.
`-fno-builtin`	Disables special handling and optimizations of standard C library functions.	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 *Read-Only Position-Independent* (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 *Read-Write Position-Independent* (RWPI) code.	Compilation only.

Table B1-1 armclang command-line options (continued)

Option	Description	Compilation or Assembly
`-fsanitize` [ALPHA]	Selects the sanitizer option used in code generation.	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.

`-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.

`-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.
`-fvectorize`, `-fno-vectorize`	Enables or disables the generation of Advanced SIMD vector instructions directly from C or C++ code at optimization levels `-O1` and higher. Enables or disables the generation of MVE instructions directly from C or C++ code at optimization levels `-O1` and higher. The use of `-fvectorize` for MVE vectorization is a [BETA] support feature.	Compilation only.
`-ftrapv`	Instructs the compiler to generate traps for signed arithmetic overflow on addition, subtraction, and multiplication operations.	Compilation only.
`-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`	Adds debug tables for source-level debugging.	Compilation and assembly.
`-I`	Adds the specified directory to the list of places that are searched to find include files.	Compilation and assembly.
`-include`	Includes the source code of the specified file at the beginning of the compilation.	Compilation only.
`-L`	Specifies a list of paths that the linker searches for user libraries.	Compilation only.
`-l`	Add 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.

Table B1-1 armclang command-line options (continued)

Option	Description	Compilation or 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`	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.
`-march`	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`	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.
`-mbranch-protection`	Protects branches using Pointer Authentication and Branch Target Identification.	Compilation only.
`-mcmodel`	Selects the generated code model.	Compilation only.
`-mcmse`	Enables the generation of code for the Secure state of the Armv8-M Security Extensions.	Compilation only.
`-mcpu`	Targets a specific processor, generating optimized code for that specific processor.	Compilation and assembly.
`-mexecute-only`	Generates execute-only code, and prevents the compiler from generating any data accesses to code sections.	Compilation only.
`-mfloat-abi`	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`	Specifies the target FPU architecture, that is the floating-point hardware available on the target.	Compilation and assembly.
`-mimplicit-it`	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.

Table B1-1 armclang command-line options (continued)

Option	Description	Compilation or 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.
`-munaligned-access`, `-mno-unaligned-access`	Enables or disables unaligned accesses to data on Arm processors.	Compilation only.
`-mthumb`	Requests that the compiler targets the T32 instruction set.	Compilation only.
`-o`	Specifies the name of the output file.	Compilation and assembly.
`-O`	Specifies the level of optimization to use when compiling source files.	Compilation only.
`-pedantic`	Generate warnings if code violates strict ISO C and ISO C++.	Compilation only.
`-pedantic-errors`	Generate errors if code violates strict ISO C and ISO C++.	Compilation only.
`-Rpass` [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`	Specifies the language standard to compile for.	Compilation only.
`--target`	Generate code for the specified target triple.	Compilation and assembly.
`-U`	Removes any initial definition of the specified preprocessor macro.	Compilation only.
`-u`	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.
`--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.
`-W`	Controls diagnostics.	Compilation only.
`-Wl`	Specifies linker command-line options to pass to the linker when a link step is being performed after compilation.	Compilation only.

Table B1-1 armclang command-line options (continued)

Option	Description	Compilation or Assembly
`-Xlinker`	Specifies linker command-line options to pass to the linker when a link step is being performed after compilation.	Compilation only.
`-x`	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.

B1.2 -C (armclang)

Keeps comments in the preprocessed output.

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.

Usage

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

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]
```

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

Here is an example program, `foo.c`, which 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>LEVEL) return 1; // Comment C
  return 0 + THIS;
}
```

The preprocessor has kept the following comments:

- `// #define THIS 99`
- `// Comment A`
- `/* Comment B */`
- `// Comment C`

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

Related references

B1.5 -E on page B1-58

B1.3 -c (armclang)

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

Usage

Arm recommends 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. For example, the following 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
```

Note

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
```

B1.4 -D

Defines a macro *`name`*.

Syntax

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

Where:

`name`

Is the name of the macro to be defined.

`parm-list`

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

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

Note

Parentheses might require escaping on UNIX systems.

`=`*`def`*

Is an optional macro definition.

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

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

Usage

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

Example

Specifying this option:

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

is equivalent to defining the macro:

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

at the head of each source file.

Related references

B1.40 -include on page B1-101

B1.79 -U on page B1-162

B1.88 -x (armclang) on page B1-171

Related information

Preprocessing assembly code

B1.5 -E

Executes the preprocessor step only.

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

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

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

Examples

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
```

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
```

Related references

B1.2 -C (armclang) on page B1-54
B1.78 --target on page B1-161

B1.6 -e

Specifies the unique initial entry point of the image.

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 the *Arm® Compiler toolchain Linker Reference* for information about the `--entry` linker options.

Related references

C1.41 --entry=location on page C1-382

B1.7 -fbare-metal-pie

Generates position independent code.

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

Note

- This option is unsupported for AArch64 state.
- The bare-metal PIE feature is deprecated.

Related references

B1.24 -fropi, -fno-ropi on page B1-81

B1.26 -frwpi, -fno-rwpi on page B1-83

C1.51 --fpic on page C1-392

C1.102 --pie on page C1-450

C1.6 --bare_metal_pie on page C1-343

C1.111 --ref_pre_init, --no_ref_pre_init on page C1-460

Related information

Bare-metal Position Independent Executables

B1.8 -fbracket-depth=N

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

Syntax

`-fbracket-depth=N`

Usage

You can increase the depth limit `N`.

Default

The default depth limit is 256.

Related references

A.3 Translation limits on page Appx-A-1118

B1.9 -fcommon, -fno-common

Generates common zero-initialized values for tentative definitions.

Tentative definitions are declarations of variables with no storage class and no initializer.

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.

Default

The default is `-fno-common`.

B1.10 -fdata-sections, -fno-data-sections

Enables or disables the generation of one ELF section for each variable in the source file. The default is -`fdata-sections`.

Note

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

Example

```
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 sections:

```
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
...
```

If you compile this code with `-fno-data-sections`, you get:

```
    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
...
```

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.

Related references

B1.14 -ffunction-sections, -fno-function-sections on page B1-69

B3.33 __attribute__((section("name"))) variable attribute on page B3-227

B1.11 -ffast-math, -fno-fast-math

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

`-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. It also ensures that the floating-point code that the compiler generates is compliant with the IEEE Standard for Floating-Point Arithmetic (IEEE 754). Arm recommends that you use the alias option `-ffp-mode=full` instead of `-fno-fast-math`.

Note

Arm Compiler 6 uses neither `-ffast-math` nor `-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 *library variants* in *Arm® C and C++ Libraries and Floating-Point Support User Guide*.

Table B1-2 Floating-point library variants

`armclang` option	Floating-point library variant	Description
Default	`fz`	IEEE-compliant library with fixed rounding mode and support for certain IEEE exceptions, and flushing to zero.
`-ffast-math`	`fz`	Similar to the default behavior, but also performs aggressive floating-point optimizations and therefore it is not IEEE-compliant.
`-fno-fast-math`	`g`	IEEE-compliant library with configurable rounding mode and support for all IEEE exceptions, and flushing to zero.

Related references

B1.13 -ffp-mode on page B1-67

B1.12 -ffixed-rN

Prevents the compiler from using the specified core register, unless the use is required for Arm ABI compliance. You must use this option if you want to reserve registers for use as a global named register variable.

Default

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

Syntax

`-ffixed-rN`

Parameters

`N` specifies the register number, which can be any number from 5 to 11. This enables you to reserve core registers R5 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 `-fwrpi` or `-fwrpi-lowering`, then you must not use register R9 as a global named register variable.

Arm recommends that you do not use the following registers as global named register variables because the Arm ABI reserves them for use as a frame pointer if needed. You must carefully analyze your code, to avoid side effects, if you want to use these registers as global named register variables:

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

Code size

Declaring a core 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. In some cases, your program might not compile, for example if there are insufficient registers available to compute a particular expression.

Operation

`-ffixed-rN` reserves the specified core 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-rN` 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-r5` reserves register R5 so that the compiler cannot use R5 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-rN` option.

Examples

The following example demonstrates the effect of the `-ffixed-rN` option.

Source file `foo.c` contains the code below:

```
int foo(int a1, int a2, int a3, int a4, int a5, int a6)
{
   return  a1/a2 + a3/a4 + a5/a6;
}
```

Compile the above code without any `-ffixed-rN` option:

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

The generated assembly file, `foo.s`, saves the registers it needs to use, which are `{r4, r5, r6, r7, r11, lr}`:

```
foo:
        .fnstart
@ %bb.0:
        .save   {r4, r5, r6, r7, r11, lr}
        push    {r4, r5, r6, r7, r11, lr}
        .pad    #40
        sub     sp, sp, #40

        /* Code in between is hidden */

        add     sp, sp, #40
        pop     {r4, r5, r6, r7, r11, pc}
.Lfunc_end0:
```

To ensure that the compiler does not use registers R5 and R6, compile the same code in `foo.c` with the `-ffixed-r5` and `-ffixed-r6` options:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -O0 -ffixed-r5 -ffixed-r6 -S foo.c -o
foo.s
```

The generated assembly file, `foo.s`, saves the registers it needs to use, which are `{r4, r7, r8, r9, r11, lr}`. In this `foo.s`, the compiler uses registers R8 and R9 instead of R5 and R6:

```
foo:
        .fnstart
@ %bb.0:
        .save   {r4, r7, r8, r9, r11, lr}
        push    {r4, r7, r8, r9, r11, lr}
        .pad    #40
        sub     sp, sp, #40

        /* Code in between is hidden */

        add     sp, sp, #40
        pop     {r4, r7, r8, r9, r11, pc}
.Lfunc_end0:
```

Related references

B2.11 Global named register variables on page B2-186

B1.13 -ffp-mode

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

Syntax

`-ffp-mode=`*`model`*

Where *`model`* is one of the following:

`std`

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

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

- NaNs and infinities might not be produced in all circumstances defined by the IEEE model. When they are produced, they might not have the same sign.
- The sign of zero might not be that predicted by the IEEE model.
- Using NaNs in arithmetic operations with `-ffp-mode=std` causes undefined behavior.

`fast`

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

A number of 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)`.
- It is not guaranteed that the value of `errno` is compliant with the ISO C or C++ standard after math functions have been called. This enables the compiler to inline the VFP square root instructions in place of calls to `sqrt()` or `sqrtf()`.

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

`full`

All facilities, operations, and representations guaranteed by the IEEE Standard for Floating-Point Arithmetic (IEEE 754) are available in single and double-precision. Modes of operation can be selected dynamically at runtime.

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

Note

When using the `std` or `fast` modes, the binary representation of a floating-point number that cannot be represented exactly by its type can differ depending on whether it is evaluated by the compiler at compile time or generated at run time 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.

Table B1-3 Floating-point library variant selection

`armclang` option	Floating-point library variant	Description
`-ffp-mode=std`	`fz`	IEEE-compliant library with fixed rounding mode and support for certain IEEE exceptions, and flushing to zero.
`-ffp-mode=fast`	`fz`	Similar to the default behavior, but also performs aggressive floating-point optimizations and therefore it is not IEEE-compliant.
`-ffp-mode=full`	`g`	IEEE-compliant library with configurable rounding mode and support for all IEEE exceptions, and flushing to zero.

Default

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

B1.14 -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.

The output section for each function has the same name as the function that generates the section, but with a `.text.` prefix. To disable this, use `-fno-function-sections`.

Note

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

Default

The default is `-ffunction-sections`.

Restrictions

`-ffunction-sections` reduces the potential for sharing addresses, data, and string literals between functions. Consequently, it might increase code size slightly for some functions.

Example

```
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- main.c

...
    .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 concepts

C4.2 Elimination of unused sections on page C4-563

Related references

B3.17 __attribute__((section("name"))) function attribute on page B3-210

B1.10 -fdata-sections, -fno-data-sections on page B1-63

B1.15 -fident, -fno-ident

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

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.

Default

The default is `-fident`.

Syntax

`-fident`

Enables the emission of the compiler name and version information.

`-Qy`

Alias for `-fident`.

`-fno-ident`

Disables the emission of the compiler name and version information.

`-Qn`

Alias for `-fno-ident`.

B1.16 @file

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

Syntax

`@file`

Where *`file`* is the name of a file containing `armclang` options to include on the command line.

Usage

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
```

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
```

B1.17 -fldm-stm, -fno-ldm-stm

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

Usage

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.

Restrictions

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

Default

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

B1.18 -fno-builtin

Disables special handling and optimization of standard C library functions, for example for `printf()`, `strlen()`, and `malloc()`.

When compiling without `-fno-builtin`, the compiler can replace calls to certain standard C 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 re-implementations of the standard C library functions might not be used, and might be removed by the linker.

Default

`-fno-builtin` is disabled by default.

Example

This example shows the result of compiling the following program with and without `-fno-builtin`:

```
#include "stdio.h"

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

1. Compile without `-fno-builtin`:

```
armclang -c -O2 -g --target=arm-arm-none-eabi -mcpu=cortex-a9 -mfpu=none -nostdlib foo.c -
o foo.o
```

2. Run the following `fromelf` command to show the disassembled output:

```
fromelf --disassemble foo.o -o foo.s
...
||foo|| PROC
        ADR      r0,|L3.8|
        B        puts
|L3.8|
        DCD      0x6c6c6548
        DCD      0x0000006f
        ENDP
...
```

Note

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

3. Compile with `-fno-builtin`:

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

4. Run the following `fromelf` command to show the disassembled output:

```
fromelf --disassemble foo.o -o foo.s
...
||foo|| PROC
        ADR      r0,|L3.8|
        B        printf
|L3.8|
        DCD      0x6c6c6548
        DCD      0x00000a6f
        ENDP
...
```

Note

The compiler has not replaced the `printf()` function with the `puts()` function when using the `-fno-builtin` option.

Related references

B1.67 -nostdlib on page B1-147

B1.68 -nostdlibinc on page B1-148

Related information

Run-time ABI for the Arm Architecture

B1.19 -fno-inline-functions

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

The compiler attempts to automatically inline functions at optimization levels `-O2` and `-O3`. When these levels are used with `-fno-inline-functions`, automatic inlining is disabled.

When optimization levels `-O0` and `-O1` are used with `-fno-inline-functions`, no automatic inlining is attempted, and only functions that are tagged with `__attribute__((always_inline))` are inlined.

Related concepts

B6.3 Inline functions on page B6-266

Related references

B1.70 -O (armclang) on page B1-150

B1.20 -flto, -fno-lto

Enables or disables link time optimization. `-flto` outputs bitcode wrapped in an ELF file for link time optimization.

The primary use for files containing bitcode is for link time optimization. See *Optimizing across modules with link time optimization* in the *User Guide* for more information about link time optimization.

Usage

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

The `-flto` option passes the `--lto` option to `armlink` to enable link time optimization, unless the `-c` option is specified.

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

Note

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

Caution

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

Caution

Link Time Optimization performs aggressive optimizations by analyzing the dependencies between bitcode format objects. This can result in the removal of unused variables and functions in the source code.

Note

Link Time Optimization 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 and produces a warning.

Default

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

Related references

B1.3 -c (armclang) on page B1-56

C1.80 --lto, --no_lto on page C1-426

Related information

Optimizing across modules with link time optimization

Restrictions with link time optimization

B1.21 -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.

Usage

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

B1.22 -fomit-frame-pointer, -fno-omit-frame-pointer

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

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`.

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`, 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.
- The behavior of frame pointers in AArch32 state is not part of the ABI and therefore might change in the future. The behavior of frame pointers in AArch64 state is part of the ABI and is therefore unlikely to change.

Default

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

B1.23 -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`.

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 and BPABI 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.

B1.24 -fropi, -fno-ropi

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

Usage

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.

Default

The default is `-fno-ropi`.

Restrictions

The following restrictions apply:

- This option is not supported in AArch64 state.
- This option cannot be used with C++ code.
- This option is not compatible with `-fpic`, `-fpie`, or `-fbare-metal-pie` options.

Related references

B1.26 -frwpi, -fno-rwpi on page B1-83
B1.27 -frwpi-lowering, -fno-rwpi-lowering on page B1-84
B1.25 -fropi-lowering, -fno-ropi-lowering on page B1-82

B1.25 -fropi-lowering, -fno-ropi-lowering

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

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 enables these static initializations to work with 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`.

B1.26 -frwpi, -fno-rwpi

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

Usage

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.

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.

Default

The default is `-fno-rwpi`.

Related references

B1.24 -fropi, -fno-ropi on page B1-81

B1.25 -fropi-lowering, -fno-ropi-lowering on page B1-82

B1.27 -frwpi-lowering, -fno-rwpi-lowering on page B1-84

B1.27 -frwpi-lowering, -fno-rwpi-lowering

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

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 enables these static initializations to work with RWPI code.

Default

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

B1.28 -fsanitize

`-fsanitize` selects the sanitizer option that is used in code generation. It is an [ALPHA] feature.

Note

This topic describes an [ALPHA] feature. See *Support level definitions* on page A1-39.

Default

The default is no sanitizers are selected.

Syntax

`-fsanitize=option`

Parameters

`option` specifies the sanitizer option for code generation. The only supported option is `-fsanitize=memtag`.

Restrictions

Memory tagging stack protection (stack tagging) is available for the AArch64 state for architectures with the Memory Tagging Extension. The Memory Tagging Extension is optional in Armv8.5-A and later architectures. When compiling with `-fsanitize=memtag`, 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 *B1.56 -mcpu* on page B1-125.

Operation

Use `-fsanitize=memtag` to enable the generation of memory tagging code for protecting the memory allocations on the stack. When you enable memory tagging, the compiler checks that expressions that evaluate to addresses of objects on the stack are within the bounds of the object. If this cannot be guaranteed, the compiler generates code to ensure that the pointer and the object are tagged. When tagged pointers are dereferenced, the processor checks the tag on the pointer with the tag on the memory location being accessed. If the tags mismatch, the processor causes an exception and therefore tries to prevent the pointer from accessing any object that is different from the object whose address was taken.

For example, if a pointer to a variable on the stack is passed to another function, then the compiler might be unable to guarantee that this pointer is only used to access the same variable. In this situation, the compiler generates memory tagging code. The memory tagging instructions apply a unique tag to the pointer and to its corresponding allocation on the stack.

Note

- The ability of the compiler to determine whether a pointer access is bounded might be affected by optimizations. For example, if an optimization inlines a function, and as a result, if the compiler can guarantee that the pointer access is always safe, then the compiler might not generate memory tagging stack protection code. Therefore, the conditions for generating memory tagging stack protection code might not have a direct relationship to the source code.
- When using `-fsanitize=memtag`, there is a high probability that an unbounded pointer access to the stack causes a processor exception. This does not guarantee that all unbounded pointer accesses to the stack cause a processor exception.
- The [ALPHA] implementation of stack tagging does not protect variable-length allocations on the stack.

To ensure full memory tagging stack protection, you must also link your code with the library that provides stack protection with memory tagging. For more information, see *C1.74 --library_security=protection* on page C1-419.

`armlink` automatically selects the library with memory tagging stack protection if at least one object file is compiled with `-fsanitize=memtag` and at least one object file is compiled with pointer authentication, using `-mbranch-protection`. You can override the selected library by using the `armlink --library_security` option to specify the library that you want to use.

Note

- Use of `-fsanitize=memtag` to protect the stack increases the amount of memory that is allocated on the stack. This is because, the compiler has to allocate a separate 16-byte aligned block of memory on the stack for each variable whose stack allocation is protected by memory tagging.
- Code that is compiled with stack tagging can be safely linked together with code that is compiled without stack tagging. However, if any object file is compiled with `-fsanitize=memtag`, and if `setjmp`, `longjmp`, or C++ exceptions are present anywhere in the image, then you must use the `v8.5a` library to avoid stack tagging related memory fault at runtime.
- The `-fsanitize=memtag` option and the `-fstack-protector` options are independent and provide complementary stack protection. These options can be used together or in isolation.

Examples

The following example demonstrates the effect of the `-fsanitize=memtag` option.

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

```
extern void func2 (int* a);

void func1(void)
{
  int x=10;
  int y=20;

  func2(&x);
  func2(&y);
}
```

Compile `foo.c`, without memory tagging stack protection, using the following command line:

```
armclang --target=aarch64-arm-none-eabi -march=armv8.5-a+memtag -S -O1 foo.c -o
mem_no_protect.s
```

The generated assembly file `mem_no_protect.s` contains the following code:

```
func1:                                  // @func1
// %bb.0:                               // %entry
        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
```

Compile `foo.c`, with memory tagging stack protection, using the following command line:

```
armclang --target=aarch64-arm-none-eabi -march=armv8.5-a+memtag -S -O1 foo.c -
fsanitize=memtag -o mem_with_protect.s
```

The generated assembly file `mem_with_protect.s` contains the following code:

```
func1:                                  // @func1
// %bb.0:                               // %entry
        stp     x19, x30, [sp, #-16]!   // 16-byte Folded Spill
        sub     sp, sp, #32             // =32
        irg     x0, sp
        mov     w8, #10
```

```
        mov     w9, #20
        addg    x19, x0, #16, #1
        stg     x0, [x0]
        str     w8, [sp]
        stg     x19, [x19]
        str     w9, [sp, #16]
        bl      func2
        mov     x0, x19
        bl      func2
        add     x8, sp, xzr
        st2g    x8, [sp], #32
        ldp     x19, x30, [sp], #16     // 16-byte Folded Reload
        ret
```

When using the `-fsanitize=memtag` option:

- The compiler generates memory tagging instructions, for example `IRG`, `ADDG`, `STG`, and `ST2G`, to ensure that the pointers and the variables on the stack are tagged. For information on these instructions, see the *Arm® A64 Instruction Set Architecture: Arm®v8, for Arm®v8-A architecture profile Documentation*.
- The compiler uses an extra 32 bytes of memory on the stack for the variables in `foo.c`, whose addresses are taken.

B1.29 -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.

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.

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.

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 };

// 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
```

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.

Related information

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

B1.30 -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.

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`.

Default

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

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
```

B1.31 -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.

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.

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.

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[8];
    get_input(buffer);
    return buffer[0];
}
```

```
// get.c
#include <string.h>

void get_input(char *data)
{
    strcpy(data, "01234567");
}
```

When `main.c` and `get.c` are compiled with `-fstack-protector`, the array `buffer` is considered vulnerable and stack protection gets applied 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.
```

B1.32 -fstrict-aliasing, -fno-strict-aliasing

Instructs the compiler to apply the strictest aliasing rules available.

Usage

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

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, since 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
```

B1.33 -fsysv, -fno-sysv

Enables or disables the generation of code suitable for the SysV linking model.

Default

The default is `-fno-sysv`.

Syntax

`-fsysv`

`-fno-sysv`

Parameters

None.

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.

B1.34 -ftrapv

Instructs the compiler to generate traps for signed arithmetic overflow on addition, subtraction, and multiplication operations.

Default

`-ftrapv` is disabled by default.

Usage

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
```

B1.35 -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.

Note

This topic includes descriptions of [BETA] features. See *Support level definitions* on page A1-39.

The use of `-fvectorize` for MVE vectorization is a [BETA] support feature.

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 and MVE instructions. The compiler or linker might still introduce Advanced SIMD or MVE instructions, such as when linking libraries that contain these instructions.

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
```

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 references

B1.3 -c (armclang) on page B1-56

B1.70 -O (armclang) on page B1-150

B1.36 -fvisibility

Sets the default visibility of ELF symbols to the specified option.

Syntax

`-fvisibility=`*`visibility_type`*

Where *`visibility_type`* is one of the following:

`default`

Default visibility corresponds to external linkage.

`hidden`

The symbol is not placed into the dynamic symbol table, so no other executable or shared library can directly reference it. Indirect references are possible using function pointers.

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.

Note

You can override this option in code with the `__attribute__((visibility("`*`visibility_type`*`")))` attribute.

Default

The default type is `-fvisibility=hidden`.

Related references

B3.21 __attribute__((visibility("visibility_type"))) function attribute on page B3-215
B3.36 __attribute__((visibility("visibility_type"))) variable attribute on page B3-230

B1.37 -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.

Usage

Note

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
```

B1.38 -g, -gdwarf-2, -gdwarf-3, -gdwarf-4 (armclang)

Adds debug tables for source-level debugging.

Syntax

`-g`

`-gdwarf-`*`version`*

Where:

`version`

is the DWARF format to produce. Valid values are 2, 3, and 4.

The `-g` option is a synonym for `-gdwarf-4`.

Usage

The compiler produces debug information that is compatible with the specified DWARF standard.

Use a compatible debugger to load, run, and debug images. For example, Arm DS-5 Debugger is compatible with DWARF 4. Compile with the `-g` or `-gdwarf-4` options to debug with Arm DS-5 Debugger.

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 should only be used for backwards compatibility.

Default

By default, `armclang` does not produce debug information. When using `-g`, the default level is DWARF 4.

Examples

If you specify multiple options, the last option specified takes precedence. For example:

- `-gdwarf-3 -gdwarf-2` produces DWARF 2 debug, because `-gdwarf-2` overrides `-gdwarf-3`.
- `-g -gdwarf-2` produces DWARF 2 debug, because `-gdwarf-2` overrides the default DWARF level implied by `-g`.
- `-gdwarf-2 -g` produces DWARF 4 debug, because `-g` (a synonym for `-gdwarf-4`) overrides `-gdwarf-2`.

B1.39 -I

Adds the specified directory to the list of places that are searched to find include files.

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

Syntax

`-I`*`dir`*

Where:

`dir`

is a directory to search for included files.

Use multiple `-I` options to specify multiple search directories.

B1.40 -include

Includes the source code of the specified file at the beginning of the compilation.

Syntax

`-include filename`

Where *`filename`* is 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 references

B1.4 -D on page B1-57

B1.39 -I on page B1-100

B1.79 -U on page B1-162

B1.41 -L

Specifies a list of paths that the linker searches for user libraries.

Syntax

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

Where:

***`dir`*`[,`*`dir`*`,...]`**

is 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.

`armclang` translates this option to `--userlibpath` and passes it to `armlink`.

See *C1.152 --userlibpath=pathlist* on page C1-503 for information.

Note

The `-L` option has no effect when used with the `-c` option, that is when not linking.

Related references

C1.152 --userlibpath=pathlist on page C1-503

B1.42 -l

Add the specified library to the list of searched libraries.

Syntax

`-l` *`name`*

Where *`name`* is the name of the library.

`armclang` translates this option to `--library` and passes it to `armlink`.

See the *Arm® Compiler toolchain Linker Reference* for information about the `--library` linker option.

Note

The `-l` option has no effect when used with the `-c` option, that is when not linking.

Related references

C1.73 --library=name on page C1-418

B1.43 -M, -MM

Produces a list of makefile dependency rules suitable for use by a make utility.

`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.

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.

Example

You can redirect output to a file using standard UNIX and MS-DOS 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 references

B1.44 -MD, -MMD on page B1-105
B1.45 -MF on page B1-106
B1.48 -MT on page B1-109
B1.69 -o (armclang) on page B1-149

B1.44 -MD, -MMD

Compiles or assembles source files and produces a list of makefile dependency rules suitable for use by a make utility.

`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.

`-MD` lists both system header files and user header files.

`-MMD` lists only user header files.

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 references

B1.43 -M, -MM on page B1-104
B1.45 -MF on page B1-106
B1.48 -MT on page B1-109

B1.45 -MF

Specifies a filename for the makefile dependency rules produced by the `-M` and `-MD` options.

Syntax

`-MF` *`filename`*

Where:

`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.

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`.

Examples

This example sends makefile dependency rules to standard output, without compiling the source:

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

This example saves 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
```

This example compiles 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
```

This example compiles 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 references

B1.43 -M, -MM on page B1-104
B1.44 -MD, -MMD on page B1-105
B1.48 -MT on page B1-109

B1.46 -MG

Prints dependency lines for header files even if the header files are missing.

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

Note

The `-MG` option only has an effect when used with one of the following options: `-M` or `-MM`.

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");
    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
```

B1.47 -MP

Emits dummy dependency rules.

These rules work around make errors that are generated if you remove header files without a corresponding update to the makefile.

Note

The `-MP` option only has an effect when used in conjunction with the `-M`, `-MD`, `-MM`, or `-MMD` options.

Examples

This example sends dependency rules to standard output, without compiling the source.

`source.c` includes a header file:

```
#include <stdio.h>

int main(void){
    puts("Hello world\n");
    return 0;
}
```

This first example is compiled without the `-MP` option, and 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
```

This second example is compiled with the `-MP` option, and 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:
```

B1.48 -MT

Changes the target of the makefile dependency rule produced by dependency generating options.

Note

The `-MT` option only has an effect when used in conjunction with either the `-M`, `-MM`, `-MD`, or `-MMD` options.

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
```

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 references

B1.43 -M, -MM on page B1-104
B1.44 -MD, -MMD on page B1-105
B1.45 -MF on page B1-106

B1.49 -march

Targets an architecture profile, generating generic code that runs on any processor of that architecture.

Note

This topic includes descriptions of [ALPHA] features. See *Support level definitions* on page A1-39.

Syntax

To specify a target architecture, use:

`-march=`*`name`*

`-march=`*`name`*`[+[no]`*`feature`*`+…]` (for architectures with optional extensions)

Where:

`name`

Specifies the architecture.

To view a list of all the supported architectures, use:

`-march=list`

The following are valid `-march` values:

`armv8-a`
Armv8 application architecture profile. Valid with both `--target=aarch64-arm-none-eabi` and `--target=arm-arm-none-eabi`.

`armv8.1-a`
Armv8.1 application architecture profile. Valid with both `--target=aarch64-arm-none-eabi` and `--target=arm-arm-none-eabi`.

`armv8.2-a`
Armv8.2 application architecture profile. Valid with both `--target=aarch64-arm-none-eabi` and `--target=arm-arm-none-eabi`.

`armv8.3-a`
Armv8.3 application architecture profile. Valid with both `--target=aarch64-arm-none-eabi` and `--target=arm-arm-none-eabi`.

`armv8.4-a`
Armv8.4 application architecture profile. Valid with both `--target=aarch64-arm-none-eabi` and `--target=arm-arm-none-eabi`.

`armv8.5-a`
Armv8.5 application architecture profile. Valid with both `--target=aarch64-arm-none-eabi` and `--target=arm-arm-none-eabi`.

[ALPHA] `armv8.6-a`
Armv8.6 application architecture profile. Valid with both `--target=aarch64-arm-none-eabi` and `--target=arm-arm-none-eabi`.

`armv8-r`
Armv8 real-time architecture profile. Only valid with `--target=arm-arm-none-eabi`.

`armv8-m.base`
Armv8 microcontroller architecture profile without the Main Extension. Derived from the Armv6-M architecture. Only valid with `--target=arm-arm-none-eabi`.

`armv8-m.main`

Armv8 microcontroller architecture profile with the Main Extension. Derived from the Armv7-M architecture. Only valid with `--target=arm-arm-none-eabi`.

`armv8.1-m.main`

Armv8.1 microcontroller architecture profile with the Main Extension. Only valid with `--target=arm-arm-none-eabi`.

`armv7-a`

Armv7 application architecture profile. Only valid with `--target=arm-arm-none-eabi`.

`armv7-r`

Armv7 real-time architecture profile. Only valid with `--target=arm-arm-none-eabi`.

`armv7-m`

Armv7 microcontroller architecture profile. Only valid with `--target=arm-arm-none-eabi`.

`armv7e-m`

Armv7 microcontroller architecture profile with DSP extension. Only valid with `--target=arm-arm-none-eabi`.

`armv6-m`

Armv6 microcontroller architecture profile. Only valid with `--target=arm-arm-none-eabi`.

`feature`

Is an optional architecture feature that might be enabled or disabled by default depending on the architecture or processor.

Note

In general, if an architecture supports the optional feature, then this optional feature is enabled by default. For AArch32 state inputs only, you can use `fromelf --decode_build_attributes` to determine whether the optional feature is enabled.

`+`*`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.

For targets in AArch64 state, you can specify one or more of the following features if the architecture supports it:

- `aes` - Cryptographic Extension. See *Cryptographic Extensions* on page B1-128 for more information.
- `crc` - CRC extension.
- `crypto` - Cryptographic Extension. See *Cryptographic Extensions* on page B1-128 for more information.
- `dotprod` - Enables the `SDOT` and `UDOT` instructions. Supported in the Armv8.2 and later Application profile architectures, and is OPTIONAL in Armv8.2 and Armv8.3.
- `fp` - Floating-point extension. See *Floating-point extensions* on page B1-129 for more information.
- `fp16` - Armv8.2-A half-precision floating-point extension. See *Floating-point extensions* on page B1-129 for more information.
- [ALPHA] `bf16` - Armv8.6-A BFloat16 floating-point extension. See *Floating-point extensions* on page B1-129 for more information. This extension is optional in Armv8.2 and later Application profile architectures.
- `fp16fml` - Half-precision floating-point multiply with add or multiply with subtract extension. Supported in the Armv8.2 and later Application profile architectures, and is OPTIONAL in Armv8.2-A and Armv8.3-A. See *Floating-point extensions* on page B1-129 for more information.
- [ALPHA] `i8mm`, `f32mm`, `f64mm` - Armv8.6-A Matrix Multiply extension. This extension is optional in Armv8.2 and later Application profile architectures. See *Matrix Multiplication Extension* on page B1-129 for more information.
- `memtag` - Armv8.5-A memory tagging extension. See *B1.28 -fsanitize* on page B1-85.
- `profile` - Armv8.2-A statistical profiling extension.
- `ras` - Reliability, Availability, and Serviceability extension.
- `predres` - Enable instructions to prevent data prediction. See *Prevention of Speculative execution and data prediction* on page B1-130 for more information.
- `rcpc` - Release Consistent Processor Consistent extension. This extension applies to Armv8.2 and later Application profile architectures.
- `rng` - Armv8.5-A random number generation extension.
- `sb` - Enable the speculation barrier `SB` instruction. See *Prevention of Speculative execution and data prediction* on page B1-130 for more information.
- `ssbs` - Enable the Speculative Store Bypass Safe instructions. See *Prevention of Speculative execution and data prediction* on page B1-130 for more information.
- `sha2` - Cryptographic Extension. See *Cryptographic Extensions* on page B1-128 for more information.

- `sha3` - Cryptographic Extension. See *Cryptographic Extensions* on page B1-128 for more information.
- `simd` - Advanced SIMD extension.
- `sm4` - Cryptographic Extension. See *Cryptographic Extensions* on page B1-128 for more information.
- `sve` - Scalable Vector Extension. This extension applies to Armv8 and later Application profile architectures. See *Scalable Vector Extension* on page B1-130 for more information.
- `sve2` - Scalable Vector Extension 2. This extension is part of the early support for Future Architecture Technologies, and applies to Armv8 and later Application profile architectures. See *Scalable Vector Extension* on page B1-130 for more information.
- `tme` - Transactional Memory Extension. This extension is part of the early support for Future Architecture Technologies, and applies to Armv8 and later Application profile architectures. See *Transactional Memory Extension* on page B1-130 for more information.

For targets in AArch32 state, you can specify one or more of the following features if the architecture supports it:

- `crc` - CRC extension for architectures Armv8 and above.
- `dotprod` - Enables the `VSDOT` and `VUDOT` instructions. Supported in Armv8.2 and later Application profile architectures, and is OPTIONAL in Armv8.2 and Armv8.3.
- `dsp` - DSP extension for the Armv8-M.mainline architecture.
- `fp16` - Armv8.2-A half-precision floating-point extension. See *Floating-point extensions* on page B1-129 for more information.
- [ALPHA] `bf16` - Armv8.6-A BFloat16 floating-point extension. See *Floating-point extensions* on page B1-129 for more information. This extension is optional in Armv8.2 and later Application profile architectures.
- `fp16fml` - Half-precision floating-point multiply with add or multiply with subtract extension. Supported in the Armv8.2 and later Application profile architectures, and is OPTIONAL in Armv8.2-A and Armv8.3-A. See *Floating-point extensions* on page B1-129 for more information.
- [ALPHA] `i8mm` - Armv8.6-A Matrix Multiply extension. This extension is optional in Armv8.2 and later Application profile architectures. See *Matrix Multiplication Extension* on page B1-129 for more information.
- `mve` - MVE extension for the Armv8.1-M architecture profile. See *M-profile Vector Extension* on page B1-130 for more information.
- `ras` - Reliability, Availability, and Serviceability extension.
- `sb` - Enable the speculation barrier `SB` instruction. See *Prevention of Speculative execution and data prediction* on page B1-130 for more information.

Note

For targets in AArch32 state, you can use `-mfpu` to specify the support for floating-point, Advanced SIMD, and Cryptographic Extensions.

Note

There are no software floating-point libraries for targets in AArch64 state. At link time `armlink` links against AArch64 library code that can use floating-point and SIMD instructions and registers. This still applies if you compile the source with `-march=<name>+nofp+nosimd` to prevent the compiler from using floating-point and SIMD instructions and registers.

To prevent the use of any floating-point instruction or register, either re-implement the library functions or create your own library that does not use floating-point instructions or registers.

Default

For targets in AArch64 state (`--target=aarch64-arm-none-eabi`), unless you target a particular processor using `-mcpu` or a particular architecture using `-march`, the compiler defaults to `-march=armv8-a`, generating generic code for Armv8-A in AArch64 state.

For targets in AArch32 state (`--target=arm-arm-none-eabi`), there is no default. You must specify either `-march` (to target an architecture) or `-mcpu` (to target a processor).

Related references

B1.56 -mcpu on page B1-125
B1.50 -marm on page B1-115
B1.66 -mthumb on page B1-146
B1.78 --target on page B1-161
B6.4 Half-precision floating-point data types on page B6-267
B6.6 Half-precision floating-point intrinsics on page B6-270

B1.50 -marm

Requests that the compiler targets the A32 instruction set.

Most Armv7-A (and earlier) processors support two instruction sets. These are the A32 instruction set (formerly ARM), and the T32 instruction set (formerly Thumb). Armv8-A processors in AArch32 state continue to support these two instruction sets, but with additional instructions. The Armv8-A architecture additionally introduces the A64 instruction set, used in AArch64 state.
Different architectures support different instruction sets:

- Armv8-A processors in AArch64 state execute A64 instructions.
- Armv8-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. For example, the `-marm` option is not valid for targets in AArch64 state. The compiler ignores the `-marm` option and generates a warning when compiling for a target in AArch64 state.

Default

The default for all targets that support A32 instructions is `-marm`.

Related references

B1.66 -mthumb on page B1-146

B1.78 --target on page B1-161

B1.56 -mcpu on page B1-125

Related information

Specifying a target architecture, processor, and instruction set

B1.51 -masm

Enables Arm Development Studio to select the correct assembler for the input assembly source files.

Default

`-masm=gnu`

Syntax

`-masm=`*`assembler`*

Parameters

The value of *`assembler`* can be one of:

`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 B1-4 Translatable options

armclang option	armasm option
`-mcpu, -march`	`--cpu`
`-marm`	`--arm`
`-mthumb`	`--thumb`
`-fropi`	`--apcs=/ropi`
`-frwpi`	`--apcs=/rwpi`

Table B1-4 Translatable options (continued)

armclang option	armasm option

`-mfloat-abi=soft`
`-mfloat-abi=softfp`
`-mfloat-abi=hard`
	`--apcs=/softfp`
	`--apcs=/softfp`
	`--apcs=/hardfp`

`-mfpu`	`--fpu`
`-mbig-endian`	`--bigend`
`-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 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 6. Arm recommends 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 Arm recommends that you do not specify the `-masm` option, and instead invoke the correct assembler explicitly.

B1.52 -mbig-endian

Generates code suitable for an Arm processor using byte-invariant big-endian (BE-8) data.

Default

The default is `-mlittle-endian`.

Related references

B1.61 -mlittle-endian on page B1-137

B1.53 -mbranch-protection

Protects branches using Pointer Authentication and Branch Target Identification.

Default

The default is `-mbranch-protection=none`.

Syntax

`-mbranch-protection=`*`protection`*

Parameters

`protection` can specify the level or type of protection.

When specifying the level of protection, it can be one of:

none

This disables all types of branch protection.

standard

This enables all types of branch protection to their standard values. The standard protection is equivalent to `-mbranch-protection=bti+pac-ret`.

When specifying the type of protection, you can enable one or more types of protection by using the `+` separator:

bti

This enables branch protection using Branch Target Identification.

pac-ret

This enables branch protection using Pointer Authentication using key A. This protects functions that save the Link Register (LR) on the stack. This does not generate branch protection code for leaf functions that do not save the LR on the stack.

If you use the `pac-ret` type of protection, you can specify additional parameters to modify the pointer authentication protection using the `+` separator:

leaf

This enables pointer authentication on all leaf functions, including the leaf functions that do not save the Link Register on the stack.

b-key

This enables pointer authentication with Key B, rather than Key A.
Key A and Key B refer to secret values that are used for generating a signature for authenticating the return addresses.

Operation

Use `-mbranch-protection` to enable or disable branch protection for your code. Branch protection protects your code from Return Oriented Programming (ROP) and Jump Oriented Programming (JOP) attacks. To protect your code from ROP attacks, enable protection using Pointer Authentication. To protect your code from JOP attacks, you must enable protection using Pointer Authentication and Branch Target Identification.

When compiling with `pac-ret`, for Armv8.3-A or later architectures, the compiler uses pointer authentication instructions that are not available for earlier architectures. The resulting code cannot be run on earlier architectures. However, when compiling with `pac-ret` for architectures before Armv8.3-A, the compiler uses pointer authentication instructions from the hint space. These instructions do not provide the branch protection in architectures before Armv8.3-A, but these instructions do provide branch protection when run on Armv8.3-A or later. This is useful when creating libraries, with branch protection, that you want to run on any Armv8-A architecture.

When compiling with `bti`, the compiler generates BTI instructions. These BTI instructions provide branch protection on Armv8.5-A or later architectures. However, on earlier architectures, these instructions are part of the hint space, and therefore these instructions are effectively NOP instructions that do not provide the BTI branch protection.

If you enable branch protection `armlink` automatically selects the library with branch protection. You can override the selected library by using the `armlink --library_security` option to specify the library that you want to use.

Branch protection using pointer authentication and branch target identification are only available in AArch64.

For more information on pointer authentication, see *Pointer authentication in AArch64 state* in the *Arm Architecture Reference Manual for Armv8-A architecture profile*.

For more information on branch target identification, see *BTI* in the *A64 Instruction Set Architecture: Armv8, for Armv8-A architecture profile*.

Examples

This enables the standard branch protection using Branch Target Identification and Pointer Authentication:

```
armclang --target=aarch64-arm-none-eabi -march=armv8.5-a -mbranch-protection=standard -c
foo.c
```

This enables the branch protection using pointer authentication, but does not use branch target identification:

```
armclang --target=aarch64-arm-none-eabi -march=armv8.5-a -mbranch-protection=pac-ret -c foo.c
```

This enables the branch protection using pointer authentication using Key B, but does not use branch target identification:

```
armclang --target=aarch64-arm-none-eabi -march=armv8.5-a -mbranch-protection=pac-ret+b-key -
c foo.c
```

This enables the branch protection using pointer authentication, including protection for all leaf functions, and also uses branch target identification:

```
armclang --target=aarch64-arm-none-eabi -march=armv8.5-a -mbranch-protection=bti+pac-ret
+leaf -c foo.c
```

This enables branch protection using pointer authentication. However, since the specified architecture is Armv8-A, the compiler generates pointer authentication instructions that are from the encoding space of hint instructions. These instructions are effectively NOP instructions and do not provide branch protection on architectures before Armv8.3-A. However these instructions do provide branch protection when run on Armv8.3-A or later architectures.

```
armclang --target=aarch64-arm-none-eabi -march=armv8-a -mbranch-protection=pac-ret -c foo.c
```

This enables branch protection using branch target identification. The compiler generates BTI instructions that are effectively NOP instructions and do not provide branch protection on architectures before Armv8.5-A. However these instructions do provide branch protection when run on Armv8.5-A or later architectures.

```
armclang --target=aarch64-arm-none-eabi -march=armv8-a -mbranch-protection=bti -c foo.c
```

Related references

C1.74 --library_security=protection on page C1-419

Related information

Pointer authentication in AArch64 state

Branch Target Identification

B1.54 -mcmodel

Selects the generated code model.

Note

This topic includes descriptions of [ALPHA] features. See *Support level definitions* on page A1-39.

Default

The default is `-mcmodel=small`.

Syntax

`-mcmodel=`*`model`*

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

[ALPHA] Generate code for the large code model. The compiler makes no assumptions about addresses and sizes of sections.

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

[ALPHA] This example enables code generation for the large model:

```
armclang --target=aarch64-arm-none-eabi -march=armv8.5-a -mcmodel=large -c foo.c
```

This example generates code for the default (small) code model:

```
armclang --target=aarch64-arm-none-eabi -march=armv8.5-a -c foo.c
```

Related concepts

C3.6 Linker-generated veneers on page C3-548

B1.55 -mcmse

Enables the generation of code for the Secure state of the Armv8-M Security Extension. This option is required when creating a Secure image.

Note

The Armv8-M Security Extension is not supported when building *Read-Only Position-Independent* (ROPI) and *Read-Write Position-Independent* (RWPI) images.

Usage

Specifying `-mcmse` targets the Secure state of the Armv8-M Security Extension. When compiling with `-mcmse`, the following are available:

- The Test Target, `TT`, instruction.
- `TT` instruction intrinsics.
- Non-secure function pointer intrinsics.
- `__attribute__((cmse_nonsecure_call))` and `__attribute__((cmse_nonsecure_entry))` function attributes.

Note

- The value of the `__ARM_FEATURE_CMSE` predefined macro 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.
- 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 `-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.

Example

This example shows how to create a Secure image using an input import library, `oldimportlib.o`, and a scatter file, `secure.scat`:

```
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 references

B1.49 -march on page B1-110
B1.59 -mfpu on page B1-134
B1.78 --target on page B1-161
B3.3 __attribute__((cmse_nonsecure_call)) function attribute on page B3-196
B3.4 __attribute__((cmse_nonsecure_entry)) function attribute on page B3-197

B6.2 Predefined macros on page B6-261
B6.9 TT instruction intrinsics on page B6-273
B6.10 Non-secure function pointer intrinsics on page B6-276
C1.53 --fpu=name (armlink) on page C1-394
C1.57 --import_cmse_lib_in=filename on page C1-398
C1.58 --import_cmse_lib_out=filename on page C1-399
C1.121 --scatter=filename on page C1-470

Related information

Building Secure and Non-secure Images Using the Armv8-M Security Extension
TT, TTT, TTA, TTAT

B1.56 -mcpu

Enables code generation for a specific Arm processor.

Note

This topic includes descriptions of [ALPHA] and [BETA] features. See *Support level definitions* on page A1-39.

Syntax

To specify a target processor, use:

`-mcpu=`*`name`*

`-mcpu=`*`name`*`[+[no]`*`feature`*`+…]` (for architectures with optional extensions)

Where:

`name`

Specifies the processor.

To view a list of all the supported processors, use:

`-mcpu=list`

`feature`

Is an optional architecture feature that might be enabled or disabled by default depending on the architecture or processor.

Note

In general, if an architecture supports the optional feature, then this optional feature is enabled by default. For AArch32 state inputs only, you can use `fromelf --decode_build_attributes` to determine whether the optional feature is enabled.

`+`*`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.

For targets in AArch64 state, you can specify one or more of the following features if the architecture supports it:

- `aes` - Cryptographic Extension. See *Cryptographic Extensions* on page B1-128 for more information.
- `crc` - CRC extension.
- `crypto` - Cryptographic Extension. See *Cryptographic Extensions* on page B1-128 for more information.
- `dotprod` - Enables the `SDOT` and `UDOT` instructions. Supported in the Armv8.2 and later Application profile architectures, and is OPTIONAL in Armv8.2 and Armv8.3.
- `fp` - Floating-point extension. See *Floating-point extensions* on page B1-129 for more information.
- `fp16` - Armv8.2-A half-precision floating-point extension. See *Floating-point extensions* on page B1-129 for more information.
- [ALPHA] `bf16` - Armv8.6-A BFloat16 floating-point extension. See *Floating-point extensions* on page B1-129 for more information. This extension is optional in Armv8.2 and later Application profile architectures.
- `fp16fml` - Half-precision floating-point multiply with add or multiply with subtract extension. Supported in the Armv8.2 and later Application profile architectures, and is OPTIONAL in Armv8.2-A and Armv8.3-A. See *Floating-point extensions* on page B1-129 for more information.
- [ALPHA] `i8mm`, `f32mm`, `f64mm` - Armv8.6-A Matrix Multiply extension. This extension is optional in Armv8.2 and later Application profile architectures. See *Matrix Multiplication Extension* on page B1-129 for more information.
- `memtag` - Armv8.5-A memory tagging extension. See *B1.28 -fsanitize* on page B1-85.
- `profile` - Armv8.2-A statistical profiling extension.
- `ras` - Reliability, Availability, and Serviceability extension.
- `predres` - Enable instructions to prevent data prediction. See *Prevention of Speculative execution and data prediction* on page B1-130 for more information.
- `rcpc` - Release Consistent Processor Consistent extension. This extension applies to Armv8.2 and later Application profile architectures.
- `rng` - Armv8.5-A random number generation extension.
- `sb` - Enable the speculation barrier `SB` instruction. See *Prevention of Speculative execution and data prediction* on page B1-130 for more information.
- `ssbs` - Enable the Speculative Store Bypass Safe instructions. See *Prevention of Speculative execution and data prediction* on page B1-130 for more information.
- `sha2` - Cryptographic Extension. See *Cryptographic Extensions* on page B1-128 for more information.

- `sha3` - Cryptographic Extension. See *Cryptographic Extensions* on page B1-128 for more information.
- `simd` - Advanced SIMD extension.
- `sm4` - Cryptographic Extension. See *Cryptographic Extensions* on page B1-128 for more information.
- `sve` - Scalable Vector Extension. This extension applies to Armv8 and later Application profile architectures. See *Scalable Vector Extension* on page B1-130 for more information.
- `sve2` - Scalable Vector Extension 2. This extension is part of the early support for Future Architecture Technologies, and applies to Armv8 and later Application profile architectures. See *Scalable Vector Extension* on page B1-130 for more information.
- `tme` - Transactional Memory Extension. This extension is part of the early support for Future Architecture Technologies, and applies to Armv8 and later Application profile architectures. See *Transactional Memory Extension* on page B1-130 for more information.

For targets in AArch32 state, you can specify one or more of the following features if the architecture supports it:

- `crc` - CRC extension for architectures Armv8 and above.
- `dotprod` - Enables the `VSDOT` and `VUDOT` instructions. Supported in Armv8.2 and later Application profile architectures, and is OPTIONAL in Armv8.2 and Armv8.3.
- `dsp` - DSP extension for the Armv8-M.mainline architecture.
- `fp16` - Armv8.2-A half-precision floating-point extension. See *Floating-point extensions* on page B1-129 for more information.
- [ALPHA] `bf16` - Armv8.6-A BFloat16 floating-point extension. See *Floating-point extensions* on page B1-129 for more information. This extension is optional in Armv8.2 and later Application profile architectures.
- `fp16fml` - Half-precision floating-point multiply with add or multiply with subtract extension. Supported in the Armv8.2 and later Application profile architectures, and is OPTIONAL in Armv8.2-A and Armv8.3-A. See *Floating-point extensions* on page B1-129 for more information.
- [ALPHA] `i8mm` - Armv8.6-A Matrix Multiply extension. This extension is optional in Armv8.2 and later Application profile architectures. See *Matrix Multiplication Extension* on page B1-129 for more information.
- `mve` - MVE extension for the Armv8.1-M architecture profile. See *M-profile Vector Extension* on page B1-130 for more information.
- `ras` - Reliability, Availability, and Serviceability extension.
- `sb` - Enable the speculation barrier `SB` instruction. See *Prevention of Speculative execution and data prediction* on page B1-130 for more information.

Note

For targets in AArch32 state, you can use `-mfpu` to specify the support for floating-point, Advanced SIMD, and Cryptographic Extensions.

Note

To write code that generates instructions for these extensions, use the intrinsics which are described in the *Arm® C Language Extensions*.

Usage

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
```

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.

Note

There are no software floating-point libraries for targets in AArch64 state. When linking for targets in AArch64 state, `armlink` uses AArch64 libraries that contain Advanced SIMD and floating-point instructions and registers. The use of the AArch64 libraries applies even if you compile the source with `-mcpu=<name>+nofp+nosimd` to prevent the compiler from using Advanced SIMD and floating-point 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 re-implement the library functions or create your own library that does not use Advanced SIMD and floating-point instructions and registers.

Default

For targets in AArch64 state (`--target=aarch64-arm-none-eabi`), the compiler generates generic code for the Armv8-A architecture in AArch64 state by default.

For targets in AArch32 state (`--target=arm-arm-none-eabi`), there is no default. You must specify either `-march` (to target an architecture) or `-mcpu` (to target a processor).

To see the default floating-point configuration for your processor:

1. Compile with `-mcpu=`*`name`*` -S` to generate the assembler file.
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`.

Cryptographic Extensions

The following table shows which algorithms the cryptographic features include for AArch64 state in the different versions of the Armv8-A architecture:

Table B1-5 Cryptographic Extensions

Feature	Armv8.0-A	Armv8.1-A	Armv8.2-A	Armv8.3-A	Armv8.4-A and later architectures
`+crypto`	SHA1, SHA256, AES	SHA1, SHA256, AES	SHA1, SHA256, AES	SHA1, SHA256, AES	SHA1, SHA256, SHA512, SHA3, AES, SM3, SM4
`+aes`	AES	AES	AES	AES	AES
`+sha2`	SHA1, SHA256	SHA1, SHA256	SHA1, SHA256	SHA1, SHA256	SHA1, SHA256
`+sha3`	-	-	SHA1, SHA256, SHA512, SHA3	SHA1, SHA256, SHA512, SHA3	SHA1, SHA256, SHA512, SHA3
`+sm4`	-	-	SM3, 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.

For AArch32 state in Armv8-A and Armv8-R, if you specify an `-mfpu` option that includes the cryptographic extension, then the cryptographic extension supports the AES, SHA1, and SHA256 algorithms.

Floating-point extensions

The following table shows the floating-point instructions that are available when you use the floating-point features:

Table B1-6 Floating-point extensions

Feature	Armv8.0-A	Armv8.1-A	Armv8.2-A	Armv8.3-A	Armv8.4-A and later architectures
`+fp`	FP	FP	FP	FP	FP
`+fp16`	-	-	FP, FP16	FP, FP16	FP, FP16, FP16fml
`+fp16fml`	-	-	FP, FP16, FP16fml	FP, FP16, FP16fml	FP, FP16, FP16fml
[ALPHA] `+bf16`	-	-	BF16	BF16	BF16

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 are supported in the Armv8.2 and later Application profile architectures, and are OPTIONAL in Armv8.2-A and Armv8.3-A.

BF16 refers to the BFloat16 floating-point dot product, matrix multiplication, and conversion operations.

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.

Matrix Multiplication Extension

Matrix Multiplication Extension is a component of the Armv8.6-A architecture and is an optional extension for the Armv8.2-A to Armv8.5-A architectures.

The following table shows the options to enable the Matrix Multiplication extension. These options are [ALPHA] support.

Table B1-7 Options for the Matrix Multiplication extension

Feature	Description
[ALPHA] `+i8mm`	Enables matrix multiplication instructions for 8-bit integer operations. This also enables the `+simd` feature in AArch64 state, or Advanced SIMD in AArch32 state.
[ALPHA] `+f32mm`	Enables matrix multiplication instructions for 32-bit single-precision floating-point operations. This also enables the `+sve` feature in AArch64 state.
[ALPHA] `+f64mm`	Enables matrix multiplication instructions for 64-bit double-precision floating-point operations. This also enables the `+sve` feature in AArch64 state.

Arm Compiler enables:

- [ALPHA] Assembly of source code containing Matrix Multiplication instructions.
- [ALPHA] Disassembly of ELF object files containing Matrix Multiplication instructions.
- [ALPHA] Support for the ACLE defined Matrix Multiplication intrinsics.

M-profile Vector Extension

M-profile Vector Extension (MVE) is an optional extension for the Armv8.1-M architecture profile.

The following table shows the options to enable MVE.

Table B1-8 Options for the MVE extension

Feature	Description
`+mve`	Enables MVE instructions for integer operations.
`+mve.fp`	Enables MVE instructions for integer and single-precision floating-point operations.
`+mve.fp+fp.dp`	Enables MVE instructions for integer, single-precision, and double-precision floating-point operations.

Arm Compiler enables:
- Assembly of source code containing MVE instructions.
- Disassembly of ELF object files containing MVE instructions.
- Support for the ACLE defined MVE intrinsics.
- [BETA] Automatic vectorization of source code operations into MVE instructions.

Scalable Vector Extension

Scalable Vector Extension (SVE) is a SIMD instruction set for Armv8-A AArch64, that 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 enables:
- Assembly of source code containing SVE instructions.
- Disassembly of ELF object files containing SVE instructions.

SVE2 builds upon SVE to add many new data-processing instructions that bring the benefits of scalable long vectors to a wider class of applications. To enable SVE2, you must use the `+sve2` option.

Transactional Memory Extension

Transactional Memory Extension (TME) is an architecture extension that adds instructions to support lock-free atomic execution of critical sections.

Prevention of Speculative execution and data prediction

Instructions are available to prevent predictions that are based on information gathered from earlier execution within a particular execution context from affecting the later Speculative execution within that context. These instructions are optional for architectures Armv8.0-A to Armv8.4-A, and are mandatory for Armv8.5-A and later architectures. 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
  ```

Examples

- To list the processors that target the AArch64 state:

```
armclang --target=aarch64-arm-none-eabi -mcpu=list
```

- To target the AArch64 state of a Cortex®-A57 processor:

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a57 test.c
```

- To target the AArch32 state of a Cortex-A53 processor, generating A32 instructions:

```
armclang --target=arm-arm-none-eabi -mcpu=cortex-a53 -marm test.c
```

- To target the AArch32 state of a Cortex-A53 processor, generating T32 instructions:

```
armclang --target=arm-arm-none-eabi -mcpu=cortex-a53 -mthumb test.c
```

- To target the AArch32 state of an Arm Neoverse™ N1 processor, use:

```
armclang --target=arm-arm-none-eabi -mcpu=neoverse-n1 test.c
```

Related references

B1.50 -marm on page B1-115
B1.66 -mthumb on page B1-146
B1.78 --target on page B1-161
B1.59 -mfpu on page B1-134
B6.4 Half-precision floating-point data types on page B6-267
B6.6 Half-precision floating-point intrinsics on page B6-270

Related information

AArch32 -- Base Instructions (alphabetic order)
A64 -- Base Instructions (alphabetic order)
Specifying a target architecture, processor, and instruction set
Preventing the use of floating-point instructions and registers

B1.57 -mexecute-only

Generates execute-only code, and prevents the compiler from generating any data accesses to code sections.

To keep code and data in separate sections, the compiler disables literal pools and branch tables when using the `-mexecute-only` option.

Restrictions

Execute-only code must be T32 code.

Execute-only code is only supported for:

- Processors that support the Armv8-M architecture, with or without the Main Extension.
- Processors that support the Armv7-M architecture, such as the Cortex-M3.

If your application calls library functions, the library objects included in the image are not execute-only compliant. You must ensure these objects are not assigned to an execute-only memory region.

Note

Arm does not provide libraries that are built without literal pools. The libraries still use literal pools, even when you use the `-mexecute-only` option.

Note

Link Time Optimization 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 and produces a warning.

Related information

Building applications for execute-only memory

B1.58 -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.

Syntax

`-mfloat-abi=`*`value`*

Where *`value`* is 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.

Default

The default for `--target=arm-arm-none-eabi` is `softfp`.

Related references

B1.59 -mfpu on page B1-134

B1.59 -mfpu

Specifies the target FPU architecture, that is the floating-point hardware available on the target.

Syntax

To view a list of all the supported FPU architectures, use:

`-mfpu=list`

Note

`-mfpu=list` is rejected when targeting AArch64 state.

Alternatively, to specify a target FPU architecture, use:

`-mfpu=`*`name`*

Where *`name`* is one of the following:

`none`

Prevents 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 is similar to the `-mfloat-abi=soft` option.

`vfpv3`

Enable the Armv7 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. Disable the Advanced SIMD extension.

`vfpv3xd-fp16`

Enable the Armv7 VFPv3XD floating-point extension, 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.

`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.

`fpv5-d16`
Enable the Armv7 FPv5-D16 floating-point extension.

`fpv5-sp-d16`
Enable the Armv7 FPv5-SP-D16 floating-point extension.

`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.

The `-mfpu` option overrides the default FPU option implied by the target architecture.

Note

- The `-mfpu` option is ignored with AArch64 targets, for example `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.
- In Armv7, the Advanced SIMD extension was called the Arm Neon™ Advanced SIMD extension.

Note

There are no software floating-point libraries for targets in AArch64 state. When linking for targets in AArch64 state, `armlink` uses AArch64 libraries that contain Advanced SIMD and floating-point instructions and registers. The use of the AArch64 libraries applies even if you compile the source with `-mcpu=<name>+nofp+nosimd` to prevent the compiler from using Advanced SIMD and floating-point 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 re-implement the library functions or create your own library that does not use Advanced SIMD and floating-point instructions and registers.

Note

In AArch32 state, if you specify `-mfloat-abi=soft`, then specifying the `-mfpu` option does not have an effect.

Default

The default FPU option depends on the target processor.

Related references

B1.56 -mcpu on page B1-125

B1.58 -mfloat-abi on page B1-133

B1.78 --target on page B1-161

Related information

Specifying a target architecture, processor, and instruction set

Preventing the use of floating-point instructions and registers

B1.60 -mimplicit-it

Specifies the behavior of the integrated assembler if there are conditional instructions outside `IT` blocks.

`-mimplicit-it=`*`name`*

Where *`name`* is one of the following:

`never`

In A32 code, the integrated assembler gives a warning when there is a conditional instruction without an enclosing IT block. In T32 code, the integrated assembler gives 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 give 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 gives an error, when there is a conditional instruction without an enclosing IT block.

`thumb`

In A32 code, the integrated assembler gives 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 gives a warning if you use the `-mimplicit-it` option with A64 code.

Default

The default is `-mimplicit-it=arm`.

Related information

IT

B1.61 -mlittle-endian

Generates code suitable for an Arm processor using little-endian data.

Default

The default is `-mlittle-endian`.

Related references

B1.52 -mbig-endian on page B1-118

B1.62 -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.

Syntax

`-mno-neg-immediates`

Usage

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.

Default

By default, `armclang` substitutes invalid instructions with an alternative instruction if the substitution is a valid equivalent instruction that produces the same result by using the logical inverse or negative of the specified immediate value.

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
 ^
```

B1.63 -moutline, -mno-outline

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

`-moutline` is only supported in AArch64 state.

Default

If the optimization level is `-Oz`, the default is `-moutline`. Otherwise the default is `-mno-outline`.

Syntax

`-moutline`
Enables outlining.

`-mno-outline`
Disables outlining.

Parameters

None.

Restrictions

Inline assembly might prevent the outlining of functions.

Examples

This example uses the following C file to show the effects of `-moutline`:

```
// foo.c

int func3(int x)
{
    return x*x;
}

int func1(int x)
{
    int i = x;
    i = i * i;
    i+=1;
    //i=func3(i);
    i+=2;

    return i;
}

char func2(char x)
{
    int i = x;
    i = i * i;
    i+=1;
    //i=func3(i);
    i+=2;

    return i;
}
```

Compile using `-S` to output the disassembly to a file as follows:

```
armclang.exe --target=aarch64-arm-none-eabi -march=armv8.5-a -moutline foo.c -S -o foo.s
armclang.exe --target=aarch64-arm-none-eabi -march=armv8.5-a -moutline foo.c -S -O1 -o
foo_O1.s
```

```
armclang.exe --target=aarch64-arm-none-eabi -march=armv8.5-a -moutline foo.c -S -O2 -o
foo_O2.s
```

Enable the calls to `func3` in `foo.c`, recompile with `-O1` and `-O2` and `_func3` appended to the output assembly filenames. The following tables show comparisons of the output:

- With no optimization and with `-O1` and the calls to `func3` disabled.
- With no optimization and with `-O2` and the calls to `func3` enabled.

Compiler-generated comments have been removed for brevity.

Functions `func1` and `func2` are outlined at `-O1`, as shown in `foo_01.s`.

If you enable the calls to `func3` and recompile with `-O1`, then no outlining occurs as shown in `foo_01_func3.s`. This case is not shown in the tables.

If you enable the calls to `func3` and recompile with `-O2`, then outlining occurs as shown in `foo_02_func3.s`.

Table B1-9 Comparison of disassembly for -moutline with and without -O1 optimization

No optimization (foo.s)

```
func3:
// %bb.0:
  sub    sp, sp, #16
  str    w0, [sp, #12]
  ldr    w0, [sp, #12]
  ldr    w8, [sp, #12]
  mul    w0, w0, w8
  add    sp, sp, #16
  ret
.Lfunc_end0:

...

func1:
// %bb.0:
  sub    sp, sp, #16
  str    w0, [sp, #12]
  ldr    w0, [sp, #12]

...

  ldr    w0, [sp, #8]
  add    sp, sp, #16
  ret
.Lfunc_end1:

...

func2:
// %bb.0:
    sub    sp, sp, #16
    strb    w0, [sp, #15]
    ldrb    w0, [sp, #15]

...

    mov    w0, w8
    add    sp, sp, #16
    ret
.Lfunc_end2:
```

-O1 (foo_01.s)

```
func3:
// %bb.0:
    mul    w0, w0, w0
    ret
.Lfunc_end0:

...

func1:
// %bb.0:
  b    OUTLINED_FUNCTION_0
.Lfunc_end1:

...

func2:
// %bb.0:
  b    OUTLINED_FUNCTION_0

...

  .section    .text.OUTLINED_FUNCTION_0,"ax",@progb
its
  .p2align    2
  .type    OUTLINED_FUNCTION_0,@function
OUTLINED_FUNCTION_0:
  .cfi_sections .debug_frame
  .cfi_startproc
// %bb.0:
  orr    w8, wzr, #0x3
  madd    w0, w0, w0, w8
  ret
.Lfunc_end3:
  .size    OUTLINED_FUNCTION_0, .Lfunc_end3-
OUTLINED_FUNCTION_0
  .cfi_endproc
```

Table B1-10 Comparison of disassembly for -moutline with and without -O2 optimization and with func3 enabled

No optimization (foo_func3.s)

```
func3:
// %bb.0:
  sub    sp, sp, #16
  str    w0, [sp, #12]
  ldr    w0, [sp, #12]
  ldr    w8, [sp, #12]
  mul    w0, w0, w8
  add    sp, sp, #16
  ret
.Lfunc_end0:

...

func1:
// %bb.0:
  sub    sp, sp, #32
  str    x30, [sp, #16]
  str    w0, [sp, #12]

...

  ldr    x30, [sp, #16]
  add    sp, sp, #32
  ret
.Lfunc_end1:

...

func2:
// %bb.0:
    sub    sp, sp, #32
    str    x30, [sp, #16]
    strb    w0, [sp, #15]

...

    ldr    x30, [sp, #16]
    add    sp, sp, #32
    ret
.Lfunc_end2:
```

-O2 and func3 enabled (foo_02_func3.s)

```
func3:
// %bb.0:
  mul    w0, w0, w0
  ret
.Lfunc_end0:

...

func1:
// %bb.0:
  orr    w8, wzr, #0x1
  madd    w8, w0, w0, w8
  b    OUTLINED_FUNCTION_0
.Lfunc_end1:

...

func2:
// %bb.0:
  and    w8, w0, #0xff
  orr    w9, wzr, #0x1
  madd    w8, w8, w8, w9
  b    OUTLINED_FUNCTION_0
.Lfunc_end2:
  .size    func2, .Lfunc_end2-func2

  .section    .text.OUTLINED_FUNCTION_0,"ax",@progb
its
  .p2align    2
  .type    OUTLINED_FUNCTION_0,@function
OUTLINED_FUNCTION_0:
  .cfi_sections .debug_frame
  .cfi_startproc
// %bb.0:
  orr    w9, wzr, #0x2
  madd    w0, w8, w8, w9
  ret
.Lfunc_end3:
  .size    OUTLINED_FUNCTION_0, .Lfunc_end3-
OUTLINED_FUNCTION_0
  .cfi_endproc
```

Related references

B1.70 -O (armclang) on page B1-150

B1.74 -S on page B1-156

B1.64 -mpixolib

Generates a Position Independent eXecute Only (PIXO) library.

Default

`-mpixolib` is disabled by default.

Syntax

`-mpixolib`

Parameters

None.

Usage

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 --pixolib` command-line option. 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 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.

  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

Restrictions

———— **Note** ————

Generation of PIXO libraries is only supported for Armv7-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.

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 references

C1.104 --pixolib on page C1-452

C1.67 --keep=section_id (armlink) on page C1-410

C1.131 --startup=symbol, --no_startup on page C1-482

Related information

The Arm C Micro-library

B1.65 -munaligned-access, -mno-unaligned-access

Enables or disables unaligned accesses to data on Arm processors.

The compiler defines the `__ARM_FEATURE_UNALIGNED` macro when `-munaligned-access` is enabled.

The libraries include special versions of certain library functions designed to exploit unaligned accesses. When unaligned access support is enabled, using `-munaligned-access`, the compilation tools use these library functions to take advantage of unaligned accesses. When unaligned access support is disabled, using `-mno-unaligned-access`, these special versions are not used.

Default

`-munaligned-access` is the default for architectures that support unaligned accesses to data. This applies to all architectures supported by Arm Compiler 6, except Armv6-M, and Armv8-M without the Main Extension.

Usage

`-munaligned-access`

Use this option on processors that support unaligned accesses to data, to speed up accesses to packed structures.

——— **Note** ———

Compiling with this option generates an error for the following architectures:

- Armv6-M.
- Armv8-M without the Main Extension.

———————

`-mno-unaligned-access`

If unaligned access is disabled, any unaligned data that is wider than 8-bit is accessed one byte at a time. For example, fields wider than 8-bit, in packed data structures, are always accessed one byte at a time even if they are aligned.

Related references

B6.2 Predefined macros on page B6-261

Related information

Arm C Language Extensions 2.0

B1.66 -mthumb

Requests that the compiler targets the T32 instruction set.

Most Armv7-A (and earlier) processors support two instruction sets. These are the A32 instruction set (formerly ARM), and the T32 instruction set (formerly Thumb). Armv8-A processors in AArch32 state continue to support these two instruction sets, but with additional instructions. The Armv8-A architecture additionally introduces the A64 instruction set, used in AArch64 state.
Different architectures support different instruction sets:

- Armv8-A processors in AArch64 state execute A64 instructions.
- Armv8-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

- 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.

Default

The default for all targets that support A32 instructions is `-marm`.

Example

```
armclang -c --target=arm-arm-none-eabi -march=armv8-a -mthumb test.c
```

Related references

B1.50 -marm on page B1-115

B1.78 --target on page B1-161

B1.56 -mcpu on page B1-125

Related information

Specifying a target architecture, processor, and instruction set

Assembling armasm and GNU syntax assembly code

B1.67 -nostdlib

Tells the compiler to not use the Arm standard C and C++ libraries.

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.

The `-nostdlib` `armclang` option, passes the `--no_scanlib` linker option to `armlink`. Therefore you must specify the location of the libraries you want to use as input objects to `armlink`, or with the `--userlibpath` `armlink` option.

Note

If you want to use your own libraries instead of the Arm standard libraries or if you want to re-implement the standard library functions, then you must use the `-nostdlib` `armclang` option. Your libraries must be compliant with the ISO C library and with the AEABI specification.

Default

`-nostdlib` is disabled by default.

Example

```
#include "math.h"

double foo(double d)
{
    return sqrt(d + 1.0);
}
int main(int argc, char *argv[])
{
    return foo(argc);
}
```

Compiling this code with `-nostdlib` generates a call to `sqrt`, which is AEABI compliant.

```
armclang --target=arm-arm-none-eabi -mcpu=Cortex-A9 -O0 -S -o- file.c -mfloat-abi=hard -
nostdlib
```

Compiling this code without `-nostdlib` generates a call to `__hardfp_sqrt` (from the Arm standard library), which the C Standard and the AEABI do not define.

```
armclang --target=arm-arm-none-eabi -mcpu=Cortex-A9 -O0 -S -o- file.c -mfloat-abi=hard
```

Related references

B1.68 -nostdlibinc on page B1-148

Related information

Run-time ABI for the Arm Architecture

C Library ABI for the Arm Architecture

B1.68 -nostdlibinc

Tells the compiler to exclude the Arm standard C and C++ library header files.

Note

This option still searches the `lib/clang/*/include` directory.

If you want to disable the use of the Arm standard library, then use both the `-nostdlibinc` and `-nostdlib` `armclang` options.

Default

`-nostdlibinc` is disabled by default.

Example

```
#include "math.h"

double foo(double d)
{
    return sqrt(d + 1.0);
}
int main(int argc, char *argv[])
{
    return foo(argc);
}
```

Compiling this code without `-nostdlibinc` generates a call to `__hardfp_sqrt`, from the Arm standard library.

```
armclang --target=arm-arm-none-eabi -mcpu=Cortex-A9 -O0 -S -o- file.c -mfloat-abi=hard
```

Compiling this code with `-nostdlibinc` and `-nostdlib` generates an error because the compiler cannot include the standard library header file `math.h`.

```
armclang --target=arm-arm-none-eabi -mcpu=Cortex-A9 -O0 -S -o- file.c -mfloat-abi=hard -
nostdlibinc -nostdlib
```

Related references

B1.67 -nostdlib on page B1-147

B1.69 -o (armclang)

Specifies the name of the output file.

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.

Default

If you do not specify a `-o` option, the compiler names the output file according to the conventions described by the following table.

Table B1-11 Compiling without the -o option

Compiler option	Action	Usage 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

B1.70 -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`*

Where *`level`* is 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 significantly 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.

`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 all the optimizations from level `3` including those optimizations that are performed with the `-ffp-mode=fast` `armclang` option. This level also performs other aggressive optimizations that 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.

Caution

This option is not guaranteed to be fully standards-compliant for all code cases.

Caution

`-Omax` automatically enables the `armclang -flto` option and the generated object files are not suitable for creating static libraries. When `-flto` is enabled, you cannot build ROPI or RWPI images.

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.
- It is not possible to output meaningful disassembly 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.
- 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

Link Time Optimization 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 and produces a warning.

`s`

Performs optimizations to reduce code size, balancing code size against code speed.

`z`

Performs optimizations to minimize image size.

Related references

B1.20 -flto, -fno-lto on page B1-77

B1.24 -fropi, -fno-ropi on page B1-81

B1.26 -frwpi, -fno-rwpi on page B1-83

Related information

Restrictions with link time optimization

B1.71 -pedantic

Generate warnings if code violates strict ISO C and ISO C++.

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++.

Default

`-pedantic` is disabled by default.

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
```

Note

The `-pedantic` option is stricter than the `-Wpedantic` option.

B1.72 -pedantic-errors

Generate errors if code violates strict ISO C and ISO C++.

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.

Default

`-pedantic-errors` is disabled by default.

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
```

B1.73 -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* on page A1-39.

Syntax

`-Rpass={*|optimization}`

Parameters

Where:

`*`
Indicates that remarks for all optimizations that are performed are to be output.

`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

The following examples use the file:

```
// test.c

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

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[8];
  strcpy(buf, p);
  printf("Copied: %s\n", buf);
}

int main(void) {
  const char *t = "Hello World!";
  copy(t);
  printf("%s\n", t);
  return 0;
}
```

- To display the inlining remarks, specify:

```
armclang -c --target=arm-arm-none-eabi -march=armv8-a -O2 -Rpass=inline test.c
test.c:22:3: remark: copy inlined into main with (cost=-14980, threshold=337) [-
Rpass=inline]
  copy(t);
  ^
```

- To display the stack protection remarks, specify:

```
armclang -c --target=arm-arm-none-eabi -march=armv8-a -fstack-protector -O0 -Rpass=stack-
protector test.c
test.c:14:13: remark: Stack protection applied to function copy due to a stack allocated
buffer or struct containing a
      buffer [-Rpass=stack-protector]
static void copy(const char *p) {
            ^
```

Related references

B1.31 -fstack-protector, -fstack-protector-all, -fstack-protector-strong, -fno-stack-protector on page B1-91

B1.74 -S

Outputs the disassembly of the machine code that the compiler generates.

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 references

B1.69 -o (armclang) on page B1-149
B1.70 -O (armclang) on page B1-150
B1.20 -flto, -fno-lto on page B1-77

B1.75 -save-temps

Instructs the compiler to generate intermediate assembly files from the specified C/C++ file.

It is similar to disassembling object code that has been compiled from C/C++.

Example

```
armclang --target=aarch64-arm-none-eabi -save-temps -c hello.c
```

Executing this command outputs the following files, that are listed in the order they are created:

- `hello.i` (or `hello.ii` for C++): the C or C++ file after pre-processing.
- `hello.bc`: the llvm-ir bitcode file.
- `hello.s`: the assembly file.
- `hello.o`: the output object file.

Note

- Specifying `-c` means that the compilation process stops after the compilation step, and does not do any linking.
- Specifying `-S` means that the compilation process stops after the disassembly step, and does not create an object file.

Related references

B1.3 -c (armclang) on page B1-56

B1.74 -S on page B1-156

B1.76 -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`.

B1.77 -std

Specifies the language standard to compile for.

Note

This topic includes descriptions of [COMMUNITY] features. See *Support level definitions* on page A1-39.

Syntax

`-std=`*`name`*

Where:

`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 additional GNU extensions.

`c99`
C as defined by the 1999 C standard.

`gnu99`
C as defined by the 1999 C standard, with additional GNU extensions.

`c11 [COMMUNITY]`
C as defined by the 2011 C standard.

`gnu11 [COMMUNITY]`
C as defined by the 2011 C standard, with additional GNU extensions.

`c++98`
C++ as defined by the 1998 C++ standard.

`gnu++98`
C++ as defined by the 1998 C++ standard, with additional GNU extensions.

`c++03`
C++ as defined by the 2003 C++ standard.

`gnu++03`
C++ as defined by the 2003 C++ standard, with additional GNU extensions.

`c++11`
C++ as defined by the 2011 C++ standard.

`gnu++11`
C++ as defined by the 2011 C++ standard, with additional GNU extensions.

`c++14`
C++ as defined by the 2014 C++ standard.

`gnu++14`
C++ as defined by the 2014 C++ standard, with additional GNU extensions.

`c++17 [COMMUNITY]`
C++ as defined by the 2017 C++ standard.

`gnu++17 [COMMUNITY]`
C++ as defined by the 2017 C++ standard, with additional GNU extensions.

For C++ code, the default is `gnu++14`. For more information about C++ support, see *C++ Status* on the Clang web site.

For C code, the default is `gnu11`. For more information about C support, see *Language Compatibility* on the Clang web site.

Note

Use of `C11` library features is unsupported.

Related references

B1.88 -x (armclang) on page B1-171

Related information

Language Compatibility

C++ Status

Language Support Levels

B1.78 --target

Generate code for the specified target triple.

Syntax

`--target=`*`triple`*

Where:

`triple`

has the form *`architecture-vendor-OS-abi`*.

Supported target triples are as follows:

`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.

Default

The `--target` option is mandatory and has no default. You must always specify a target triple.

Related references

B1.50 -marm on page B1-115

B1.66 -mthumb on page B1-146

B1.56 -mcpu on page B1-125

B1.59 -mfpu on page B1-134

F1.13 --cpu=name (armasm) on page F1-860

C1.23 --cpu=name (armlink) on page C1-362

Related information

Specifying a target architecture, processor, and instruction set

B1.79 -U

Removes any initial definition of the specified macro.

Syntax

`-U`*`name`*

Where:

`name`

is 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.

Usage

Specifying `-U`*`name`* has the same effect as placing the text `#undef` *`name`* at the head of each source file.

Restrictions

The compiler defines and undefines macros in the following order:
1. Compiler predefined macros.
2. Macros defined explicitly, using `-D`*`name`*.
3. Macros explicitly undefined, using `-U`*`name`*.

Related references

B1.4 -D on page B1-57

B6.2 Predefined macros on page B6-261

B1.40 -include on page B1-101

B1.80 -u (armclang)

Prevents the removal of a specified symbol if it is undefined.

Syntax

`-u symbol`

Where `symbol` is the symbol to keep.

`armclang` translates this option to `--undefined` and passes it to `armlink`.

See *C1.148 --undefined=symbol* on page C1-499 for information about the `--undefined` linker option.

Related references
C1.148 --undefined=symbol on page C1-499

B1.81 -v (armclang)

Displays the commands that invoke the compiler and sub-tools, such as `armlink`, and executes those commands.

Usage

The `-v` compiler option produces diagnostic output showing exactly how the compiler and linker are invoked, displaying the options for each tool. The `-v` compiler option also displays 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.

Related references

B1.89 -### on page B1-172

B1.82 --version (armclang)

Displays the same information as `--vsn`.

Related references

B1.84 --vsn (armclang) on page B1-167

B1.83 --version_number (armclang)

Displays the version of armclang that you are using.

Usage

armclang 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 references

B6.2 Predefined macros on page B6-261

B1.84 --vsn (armclang) on page B1-167

B1.84 --vsn (armclang)

Displays the version information and the license details.

Note

`--vsn` is intended to report the version information for manual inspection. The `Component` line indicates the release of Arm Compiler 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`.

Example

Example output:

```
> armclang --vsn
Product: ARM Compiler N.n.p
Component: ARM Compiler N.n.p
Tool: armclang [tool_id]

Target: target_name
```

Related references

B1.82 --version (armclang) on page B1-165
B1.83 --version_number (armclang) on page B1-166

B1.85 -W

Controls diagnostics.

Syntax

`-W`*`name`*

Where common values for *`name`* include:

`-Wc11-extensions`
: Warns about any use of C11-specific features.

`-Werror`
: Turn warnings into errors.

`-Werror=foo`
: Turn warning `foo` into an error.

`-Wno-error=foo`
: Leave warning `foo` as a warning even if `-Werror` is specified.

`-Wfoo`
: Enable warning `foo`.

`-Wno-foo`
: Suppress warning `foo`.

`-Weverything`
: Enable all warnings.

`-Wpedantic`
: Generate warnings if code violates strict ISO C and ISO C++.

See *Controlling Errors and Warnings* in the *Clang Compiler User's Manual* for full details about controlling diagnostics with `armclang`.

Related information

Options for controlling diagnostics with armclang

B1.86 -Wl

Specifies linker command-line options to pass to the linker when a link step is being performed after compilation.

See the *Chapter C1 armlink Command-line Options* on page C1-333 for information about available linker options.

Syntax

`-Wl,opt,[opt[,...]]`

Where:

`opt`

is 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 references

B1.87 -Xlinker on page B1-170

Chapter C1 armlink Command-line Options on page C1-333

B1.87 -Xlinker

Specifies linker command-line options to pass to the linker when a link step is being performed after compilation.

See the *Chapter C1 armlink Command-line Options* on page C1-333 for information about available linker options.

Syntax

`-Xlinker` *`opt`*

Where:

`opt`

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

If you want to pass multiple options, use multiple `-Xlinker` 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 references

B1.86 -Wl on page B1-169

Chapter C1 armlink Command-line Options on page C1-333

B1.88 -x (armclang)

Specifies the language of source files.

Syntax

`-x` *`language`*

Where:

`language`

Specifies the language of subsequent source files, one of the following:

`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.

Usage

`-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.

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.

Related references

B1.4 -D on page B1-57

B1.77 -std on page B1-159

Related information

Preprocessing assembly code

B1.89 -###

Displays the commands that invoke the compiler and sub-tools, such as `armlink`, without executing those commands.

Usage

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 references

B1.81 -v (armclang) on page B1-164

Chapter B2
Compiler-specific Keywords and Operators

Summarizes the compiler-specific keywords and operators that are extensions to the C and C++ Standards.

It contains the following sections:

- *B2.1 Compiler-specific keywords and operators* on page B2-174.
- *B2.2 __alignof__* on page B2-175.
- *B2.3 __asm* on page B2-177.
- *B2.4 __declspec attributes* on page B2-179.
- *B2.5 __declspec(noinline)* on page B2-180.
- *B2.6 __declspec(noreturn)* on page B2-181.
- *B2.7 __declspec(nothrow)* on page B2-182.
- *B2.8 __inline* on page B2-183.
- *B2.9 __promise* on page B2-184.
- *B2.10 __unaligned* on page B2-185.
- *B2.11 Global named register variables* on page B2-186.

B2.1 Compiler-specific keywords and operators

The Arm 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 the Arm compiler are not documented.

Keyword extensions that the Arm compiler supports:

- `__alignof__`
- `__asm`
- `__declspec`
- `__inline`

Related references

B2.2 __alignof__ on page B2-175
B2.3 __asm on page B2-177
B2.4 __declspec attributes on page B2-179
B2.8 __inline on page B2-183

B2.2 __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 the Arm compiler supports.

———————————

Syntax

`__alignof__(type)`

`__alignof__(expr)`

Where:

`type`

is a type

`expr`

is an lvalue.

Return value

`__alignof__(type)` returns the alignment requirement for the type, 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

The following example displays the alignment requirements for a variety of data types, first directly from the data type, then from an lvalue of the corresponding data type:

```
#include <stdio.h>

int main(void)
{
  int       var_i;
  char      var_c;
  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));
}
```

Compiling with the following command produces the following output:

```
armclang --target=arm-arm-none-eabi -march=armv8-a alignof_test.c -o alignof.axf
```

```
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
```

B2.3 __asm

This keyword passes information to the `armclang` assembler.

The precise action of this keyword depends on its usage.

Usage

Inline assembly

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);
}
```

The general form of an `__asm` inline assembly statement is:

`__asm(`*`code`*` [: `*`output_operand_list`*` [: `*`input_operand_list`*` [: `*`clobbered_register_list`*`]]]);`

`code` is the assembly code. In our example, this is `"ADD %[result], %[input_i], %[input_j]"`.

`output_operand_list` is an optional list of output operands, separated by commas. Each operand consists of a symbolic name in square brackets, a constraint string, and a C expression in parentheses. In our example, there is a single output operand: `[result] "=r" (res)`.

`input_operand_list` is an optional list of input operands, separated by commas. Input operands use the same syntax as output operands. In our example there are two input operands: `[input_i] "r" (i), [input_j] "r" (j)`.

`clobbered_register_list` is an optional list of clobbered registers. In our example, this is omitted.

Embedded assembly

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))`* `on page B3-203`. 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`*`);`

Assembly labels

The `__asm` keyword can specify an assembly label for a C symbol. For example:

```
int count __asm__("count_v1"); // export count_v1, not count
```

Related references

B3.10 __attribute__((naked)) function attribute on page B3-203

B2.4 __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 available `__declspec` attributes are as follows:

- `__declspec(noinline)`
- `__declspec(noreturn)`
- `__declspec(nothrow)`

`__declspec` attributes are storage class modifiers. They do not affect the type of a function or variable.

Related references

B2.5 __declspec(noinline) on page B2-180
B2.6 __declspec(noreturn) on page B2-181
B2.7 __declspec(nothrow) on page B2-182
B3.11 __attribute__((noinline)) function attribute on page B3-204
B3.13 __attribute__((noreturn)) function attribute on page B3-206
B3.14 __attribute__((nothrow)) function attribute on page B3-207

B2.5 __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))`.

Example

```
/* Suppress inlining of foo() wherever foo() is called */
__declspec(noinline) int foo(void);
```

Related references

B3.11 __attribute__((noinline)) function attribute on page B3-204

B2.6 __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))`.

Usage

Use this attribute to reduce the cost of calling a function that never returns, such as `exit()`. If a `noreturn` function returns to its caller, the behavior is undefined.

Restrictions

The return address is not preserved when calling the `noreturn` function. This limits the ability of a debugger to display the call stack.

Example

```
__declspec(noreturn) void overflow(void); // never return on overflow
int negate(int x)
{
    if (x == 0x80000000) overflow();
    return -x;
}
```

Related references

B3.13 __attribute__((noreturn)) function attribute on page B3-206

B2.7 __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))`.

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`.

Usage

If the compiler knows that a function can never throw an exception, it might be able to generate smaller exception-handling tables for callers of that function.

Restrictions

If a call to a function results in a C++ exception being propagated from the callee into the caller, the behavior is undefined.

This modifier is ignored when not compiling with exceptions enabled.

Example

```
struct S
{
    ~S();
};
__declspec(nothrow) extern void f(void);
void g(void)
{
    S s;
    f();
}
```

Related references

B3.14 __attribute__((nothrow)) function attribute on page B3-207

Related information

Standard C++ library implementation definition

B2.8 __inline

The `__inline` keyword suggests to the compiler that it compiles a C or C++ function inline, if it is sensible to do so.

`__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 concepts

B6.3 Inline functions on page B6-266

B2.9 __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`*`)`

Where *`expr`* is an expression that evaluates to nonzero.

Usage

You must `#include <assert.h>` to use `__promise(`*`expr`*`)`.

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`*`)`.

B2.10 __unaligned

The `__unaligned` keyword is a type qualifier that tells the compiler to treat the pointer or variable as an unaligned pointer or variable.

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

```
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

```
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.
}
```

B2.11 Global named register variables

The compiler enables you to use the `register` storage class specifier to store global variables in core 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 core register to use to store the variable. The core register can be `R5` 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 `-fwrpi` or `-fwrpi-lowering`, then you must not use register R9 as a global named register variable.

Arm recommends that you do not use the following registers as global named register variables because the Arm ABI reserves them for use as a frame pointer if needed. You must carefully analyze your code, to avoid side effects, if you want to use these registers as global named register variables:

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

Code size

Declaring a core 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. In some cases, your program might not compile, for example if there are insufficient registers available to compute a particular expression.

Operation

Using global named register variables enables faster access to these variables than if they were 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 R5 as a global named register for an integer `foo`, you must use:

```
register int  foo  __asm("R5")
```

For this example, you must compile with the command-line option `-ffixed-r5`. For more information, see *B1.12 -ffixed-rN* on page B1-65.

The Arm standard library has not been built with any `-ffixed-rN` 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, static, or typedef 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-rN` option.

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

```
#include <stdio.h>

/* Function defined in another file that will be compiled with
   -ffixed-r5 -ffixed-r6. */
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("r5");
register int *bar __asm("r6");

__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);  // should print 1000
   printf("bar: %d\n", *bar); // should print 1
}
```

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

```
/* Arbitrary function that could normally result in the compiler using R5 and R6.
When compiling with -ffixed-r5 -ffixed-r6, the compiler should not use registers
R5 or R6 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 R5 and R6, and therefore both source files must be compiled with the relevant `-ffixed-rN` option:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -O2 -ffixed-r5 -ffixed-r6 -c main.c -o
main.o --save-temps
```

```
armclang --target=arm-arm-none-eabi -march=armv8-a -O2 -ffixed-r5 -ffixed-r6 -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 R5 and R6 in the `add_ratio()` function:

```
add_ratio:
        .fnstart
@ %bb.0:
        .save   {r4, r7, r11, lr}
        push    {r4, r7, r11, lr}
        ldr     r12, [sp, #20]
        sdiv    r4, r2, r3
        ldr     lr, [sp, #16]
        sdiv    r7, r0, r1
        add     r4, r4, r7
        cmp     r0, r1
        sdiv    r7, lr, r12
        cmpgt   r2, r3
        add     r4, r4, r7
        bgt     .LBB0_2
@ %bb.1:
        sdiv    r0, r4, lr
        sdiv    r0, r0, r12
        pop     {r4, r7, r11, pc}
.LBB0_2:
        mul     r0, r12, lr
        mul     r0, r0, r4
        pop     {r4, r7, r11, pc}
```

The file `main.s` has been generated from `main.c`, and uses registers R5 and R6 only for the code that directly uses these global named register variables:

```
initialise_named_registers:
        .fnstart
@ %bb.0:
        movw    r6, :lower16:other_location
        mov     r5, #1000
        movt    r6, :upper16:other_location
        ldr     r0, [r6]
        add     r0, r0, #1
        str     r0, [r6]
        mov     r0, #0
        bx      lr
```

```
main:
        .fnstart
@ %bb.0:
        .save   {r11, lr}
        push    {r11, lr}
        .pad    #8
        sub     sp, sp, #8
        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, r5
        bl      __2printf
        ldr     r1, [r6]
        adr     r0, .LCPI1_1
        bl      __2printf
```

```
        mov     r0, #0
        add     sp, sp, #8
        pop     {r11, pc}
        .p2align        2
```

Note

The Arm standard library code, such as the library implementations for the `printf()` function, might still use R5 and R6 because the standard library has not been built with any `-ffixed-rN` option.

Related references

B1.12 -ffixed-rN on page B1-65

Chapter B3
Compiler-specific Function, Variable, and Type Attributes

Summarizes the compiler-specific function, variable, and type attributes that are extensions to the C and C++ Standards.

It contains the following sections:

- *B3.1 Function attributes* on page B3-193.
- *B3.2 __attribute__((always_inline)) function attribute* on page B3-195.
- *B3.3 __attribute__((cmse_nonsecure_call)) function attribute* on page B3-196.
- *B3.4 __attribute__((cmse_nonsecure_entry)) function attribute* on page B3-197.
- *B3.5 __attribute__((const)) function attribute* on page B3-198.
- *B3.6 __attribute__((constructor(priority))) function attribute* on page B3-199.
- *B3.7 __attribute__((format_arg(string-index))) function attribute* on page B3-200.
- *B3.8 __attribute__((interrupt("type"))) function attribute* on page B3-201.
- *B3.9 __attribute__((malloc)) function attribute* on page B3-202.
- *B3.10 __attribute__((naked)) function attribute* on page B3-203.
- *B3.11 __attribute__((noinline)) function attribute* on page B3-204.
- *B3.12 __attribute__((nonnull)) function attribute* on page B3-205.
- *B3.13 __attribute__((noreturn)) function attribute* on page B3-206.
- *B3.14 __attribute__((nothrow)) function attribute* on page B3-207.
- *B3.15 __attribute__((pcs("calling_convention"))) function attribute* on page B3-208.
- *B3.16 __attribute__((pure)) function attribute* on page B3-209.
- *B3.17 __attribute__((section("name"))) function attribute* on page B3-210.
- *B3.18 __attribute__((unused)) function attribute* on page B3-211.
- *B3.19 __attribute__((used)) function attribute* on page B3-212.
- *B3.20 __attribute__((value_in_regs)) function attribute* on page B3-213.

- *B3.21 __attribute__((visibility("visibility_type"))) function attribute* on page B3-215.
- *B3.22 __attribute__((weak)) function attribute* on page B3-216.
- *B3.23 __attribute__((weakref("target"))) function attribute* on page B3-217.
- *B3.24 Type attributes* on page B3-218.
- *B3.25 __attribute__((aligned)) type attribute* on page B3-219.
- *B3.26 __attribute__((packed)) type attribute* on page B3-220.
- *B3.27 __attribute__((transparent_union)) type attribute* on page B3-221.
- *B3.28 Variable attributes* on page B3-222.
- *B3.29 __attribute__((alias)) variable attribute* on page B3-223.
- *B3.30 __attribute__((aligned)) variable attribute* on page B3-224.
- *B3.31 __attribute__((deprecated)) variable attribute* on page B3-225.
- *B3.32 __attribute__((packed)) variable attribute* on page B3-226.
- *B3.33 __attribute__((section("name"))) variable attribute* on page B3-227.
- *B3.34 __attribute__((unused)) variable attribute* on page B3-228.
- *B3.35 __attribute__((used)) variable attribute* on page B3-229.
- *B3.36 __attribute__((visibility("visibility_type"))) variable attribute* on page B3-230.
- *B3.37 __attribute__((weak)) variable attribute* on page B3-231.
- *B3.38 __attribute__((weakref("target"))) variable attribute* on page B3-232.

B3.1 Function attributes

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__, ...))
```

For example:

```
int my_function(int b) __attribute__((const));
static int my_variable __attribute__((__unused__));
```

The following table summarizes the available function attributes.

Table B3-1 Function attributes that the compiler supports, and their equivalents

Function attribute	Non-attribute equivalent
`__attribute__((alias))`	-
`__attribute__((always_inline))`	-
`__attribute__((cmse_nonsecure_call))`	-
`__attribute__((cmse_nonsecure_entry))`	-
`__attribute__((const))`	-
`__attribute__((constructor(`*`priority`*`)))`	-
`__attribute__((deprecated))`	-
`__attribute__((format_arg(`*`string-index`*`)))`	-
`__attribute__((interrupt("`*`type`*`")))`	-
`__attribute__((malloc))`	-
`__attribute__((naked))`	-
`__attribute__((noinline))`	`__declspec(noinline)`
`__attribute__((nomerge))`	-
`__attribute__((nonnull))`	-
`__attribute__((noreturn))`	`__declspec(noreturn)`
`__attribute__((nothrow))`	`__declspec(nothrow)`
`__attribute__((notailcall))`	-
`__attribute__((pcs("`*`calling_convention`*`")))`	-
`__attribute__((pure))`	-
`__attribute__((section("`*`name`*`")))`	-
`__attribute__((unused))`	-
`__attribute__((used))`	-
`__attribute__((value_in_regs))`	-
`__attribute__((visibility("`*`visibility_type`*`")))`	-

Table B3-1 Function attributes that the compiler supports, and their equivalents (continued)

Function attribute	Non-attribute equivalent
`__attribute__((weak))`	-
`__attribute__((weakref("target")))`	-

Usage

You can set these function attributes in the declaration, the definition, or both. For example:

```
void AddGlobals(void) __attribute__((always_inline));
__attribute__((always_inline)) void AddGlobals(void) {...}
```

When function attributes conflict, the compiler uses the safer or stronger one. For example, `__attribute__((used))` is safer than `__attribute__((unused))`, and `__attribute__((noinline))` is safer than `__attribute__((always_inline))`.

Related references

B3.2 __attribute__((always_inline)) function attribute on page B3-195
B3.5 __attribute__((const)) function attribute on page B3-198
B3.6 __attribute__((constructor(priority))) function attribute on page B3-199
B3.7 __attribute__((format_arg(string-index))) function attribute on page B3-200
B3.9 __attribute__((malloc)) function attribute on page B3-202
B3.12 __attribute__((nonnull)) function attribute on page B3-205
B3.10 __attribute__((naked)) function attribute on page B3-203
B3.15 __attribute__((pcs("calling_convention"))) function attribute on page B3-208
B3.11 __attribute__((noinline)) function attribute on page B3-204
B3.14 __attribute__((nothrow)) function attribute on page B3-207
B3.17 __attribute__((section("name"))) function attribute on page B3-210
B3.16 __attribute__((pure)) function attribute on page B3-209
B3.13 __attribute__((noreturn)) function attribute on page B3-206
B3.18 __attribute__((unused)) function attribute on page B3-211
B3.19 __attribute__((used)) function attribute on page B3-212
B3.21 __attribute__((visibility("visibility_type"))) function attribute on page B3-215
B3.22 __attribute__((weak)) function attribute on page B3-216
B3.23 __attribute__((weakref("target"))) function attribute on page B3-217
B2.2 __alignof__ on page B2-175
B2.3 __asm on page B2-177
B2.4 __declspec attributes on page B2-179

B3.2 __attribute__((always_inline)) function attribute

This function attribute indicates that a function must be inlined.

The compiler attempts to inline the function, regardless of the characteristics of the function.

In some circumstances, the compiler 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()` might not be inlined.

Example

```
static int max(int x, int y) __attribute__((always_inline));
static int max(int x, int y)
{
    return x > y ? x : y; // always inline if possible
}
```

Note

`__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
}
```

B3.3 __attribute__((cmse_nonsecure_call)) function attribute

Declares a non-secure function type

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

```
#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 references

B3.4 __attribute__((cmse_nonsecure_entry)) function attribute on page B3-197

B6.10 Non-secure function pointer intrinsics on page B6-276

Related information

Building Secure and Non-secure Images Using the Armv8-M Security Extension

B3.4 __attribute__((cmse_nonsecure_entry)) function attribute

Declares an entry function that can be called from Non-secure state or Secure state.

Syntax

C linkage:

```
void __attribute__((cmse_nonsecure_entry)) entry_func(int val)
```

C++ linkage:

```
extern "C" void __attribute__((cmse_nonsecure_entry)) entry_func(int val)
```

Note

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

```
#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 references

B3.3 __attribute__((cmse_nonsecure_call)) function attribute on page B3-196

B6.10 Non-secure function pointer intrinsics on page B6-276

Related information

Building Secure and Non-secure Images Using the Armv8-M Security Extension

B3.5 __attribute__((const)) function attribute

The `const` function attribute specifies that a function examines only its arguments, and has no effect except for the return value. That is, the function does not read or modify any global memory.

If a function is known to operate only on its arguments then it can be subject to common sub-expression elimination and loop optimizations.

This 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);
    }
}
```

B3.6 __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`*`)))`

Where *`priority`* is an optional integer value denoting the priority. A constructor with a low integer value runs before a constructor with a high integer value. A constructor with a priority runs before a constructor without a priority.

Priority values up to and including 100 are reserved for internal use. If you use these values, the compiler gives a warning.

Usage

You can use this attribute for start-up or initialization code.

Example

In the following example, the constructor functions are called before execution enters `main()`, in the order specified:

```
#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()
```

B3.7 __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`*`)))`

Where *`string-index`* specifies the argument that is the format string argument (starting from one).

Example

The following example declares two functions, `myFormatText1()` and `myFormatText2()`, that provide format strings to `printf()`.

The first function, `myFormatText1()`, does not specify the `format_arg` attribute. The compiler does not check the types of the `printf` arguments for consistency with the format string.

The second function, `myFormatText2()`, specifies the `format_arg` attribute. In the subsequent calls to `printf()`, the compiler checks that the types of the supplied arguments `a` and `b` are consistent with the format string argument to `myFormatText2()`. The compiler produces a warning when a `float` is provided where an `int` is expected.

```
#include <stdio.h>

// Function used by printf. No format type checking.
extern char *myFormatText1 (const char *);

// Function used by printf. Format type checking on argument 1.
extern char *myFormatText2 (const char *) __attribute__((format_arg(1)));


int main(void) {
  int a;
  float b;

  a = 5;
  b = 9.099999;

  printf(myFormatText1("Here is an integer: %d\n"), a); // No type checking. Types match 
anyway.
  printf(myFormatText1("Here is an integer: %d\n"), b); // No type checking. Type mismatch, 
but no warning

  printf(myFormatText2("Here is an integer: %d\n"), a); // Type checking. Types match.
  printf(myFormatText2("Here is an integer: %d\n"), b); // Type checking. Type mismatch 
results in warning
}
```

```
$ 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.
```

B3.8 __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")))`

Where *`type`* is one of the following:

- IRQ.
- FIQ.
- SWI.
- ABORT.
- UNDEF.

Usage

This attribute affects the code generation of a function as follows:

- If the function is AAPCS, the stack is realigned to 8 bytes on entry.
- For processors that are not based on the M-profile, 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.

Restrictions

When using `__attribute__((interrupt("type")))` functions:

- No arguments or return values can be used with the functions.
- The functions are incompatible with `-frwpi`.

Note

In Armv6-M, Armv7-M, and Armv8-M, the architectural exception handling mechanism preserves all processor registers, and a standard function return can cause an exception return. Therefore, specifying this attribute does not affect the behavior of the compiled output. However, Arm recommends using this attribute on exception handlers for clarity and easier software porting.

Note

- For architectures that support A32 and T32 instructions, functions 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.

B3.9 __attribute__((malloc)) function attribute

This function attribute indicates that the function can be treated like `malloc` and the compiler can perform the associated optimizations.

Example

```
void * foo(int b) __attribute__((malloc));
```

B3.10 __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.

The compiler does not generate prologue and epilogue sequences for functions with `__attribute__((naked))`.

The compiler only supports basic `__asm` statements in `__attribute__((naked))` functions. Using extended assembly, parameter references or mixing C code with `__asm` statements might not work reliably.

Example B3-1 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 references

B2.3 __asm on page B2-177

B3.11 __attribute__((noinline)) function attribute

This attribute suppresses the inlining of a function at the call points of the function.

Example

```
/* Suppress inlining of foo() wherever foo() is called */
int foo(void) __attribute__((noinline));
```

B3.12 __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, ...)]))`

Where `[(arg-index, ...)]` denotes an optional argument index list.

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));
```

B3.13 __attribute__((noreturn)) function attribute

This attribute asserts that a function never returns.

Usage

Use this attribute to reduce the cost of calling a function that never returns, such as exit(). If a noreturn function returns to its caller, the behavior is undefined.

Restrictions

The return address is not preserved when calling the `noreturn` function. This limits the ability of a debugger to display the call stack.

B3.14 __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.

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.

B3.15 __attribute__((pcs("calling_convention"))) function attribute

This function attribute specifies the calling convention on targets with hardware floating-point.

Syntax

`__attribute__((pcs("`*`calling_convention`*`")))`

Where *`calling_convention`* is one of the following:

`aapcs`

uses integer registers.

`aapcs-vfp`

uses floating-point registers.

Example

```
double foo (float) __attribute__((pcs("aapcs")));
```

B3.16 __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.

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;
}
```

The call to `bar` in this example might be eliminated because its result is not used.

B3.17 __attribute__((section("name"))) function attribute

The `section` function attribute enables you to place code in different sections of the image.

Example

In the following example, the function `foo` is placed into an RO section named `new_section` rather than `.text`.

```
int foo(void) __attribute__((section ("new_section")));
int foo(void)
{
  return 2;
}
```

Note

Section names must be unique. You must not use the same section name for different section types. If you use the same section name for different section types, then the compiler merges the sections into one and gives the section the type of whichever function or variable is first allocated to that section.

B3.18 __attribute__((unused)) function attribute

The `unused` function attribute prevents the compiler from generating warnings if the function is not referenced. This does not change the behavior of the unused function removal process.

Note

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)
{
  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 references

B3.34 __attribute__((unused)) variable attribute on page B3-228

B3.19 __attribute__((used)) function attribute

This function attribute informs the compiler that a static function is to be retained in the object file, even if it is unreferenced.

Functions marked with `__attribute__((used))` are tagged in the object file to avoid removal by linker unused section removal.

Note

Static variables can also be marked as used, 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 (;;);
}
```

Related concepts

C4.2 Elimination of unused sections on page C4-563

B3.20 __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]);`

Where:

`return-type`

is the type of the returned structure that conforms to certain restrictions as described in *Restrictions* on page B3-213.

Usage

Declaring a function `__attribute__((value_in_regs))` can be useful when calling functions that return more than one result.

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.
- 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.

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);
```

B3.21 __attribute__((visibility("visibility_type"))) function attribute

This function attribute affects the visibility of ELF symbols.

Syntax

`__attribute__((visibility("`*`visibility_type`*`")))`

Where *`visibility_type`* is one of the following:

`default`

Default visibility corresponds to external linkage.

`hidden`

The symbol is not placed into the dynamic symbol table, so no other executable or shared library can directly reference it. Indirect references are possible using function pointers.

`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.

Usage

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.

Default

If you do not specify visibility, then the default type is `default` for **`extern`** declarations and `hidden` for everything else.

Example

```
void __attribute__((visibility("protected"))) foo()
{
   ...
}
```

Related references

B1.36 -fvisibility on page B1-97

B3.36 __attribute__((visibility("visibility_type"))) variable attribute on page B3-230

B3.22 __attribute__((weak)) function attribute

Functions defined with `__attribute__((weak))` export their symbols weakly.

Functions declared with `__attribute__((weak))` and then defined without `__attribute__((weak))` behave as *weak* functions.

Example

```
extern int Function_Attributes_weak_0 (int b) __attribute__((weak));
```

B3.23 __attribute__((weakref("target"))) function attribute

This function attribute marks a function declaration as an alias that does not by itself require a function definition to be given for the target symbol.

Syntax

```
__attribute__((weakref("target")))
```

Where *`target`* is the target symbol.

Example

In the following example, `foo()` calls `y()` through a weak reference:

```
extern void y(void);
static void x(void) __attribute__((weakref("y")));
void foo (void)
{
  ...
  x();
  ...
}
```

Restrictions

This attribute can only be used on functions with static linkage.

B3.24 Type attributes

The `__attribute__` keyword enables you to specify special attributes of variables or structure fields, functions, and types.

The keyword format is either of the following:

```
__attribute__((attribute1, attribute2, ...))
__attribute__((__attribute1__, __attribute2__, ...))
```

For example:

```
typedef union { int i; float f; } U __attribute__((transparent_union));
```

The available type attributes are as follows:

- `__attribute__((aligned))`
- `__attribute__((packed))`
- `__attribute__((transparent_union))`

Related references

B3.25 __attribute__((aligned)) type attribute on page B3-219

B3.27 __attribute__((transparent_union)) type attribute on page B3-221

B3.26 __attribute__((packed)) type attribute on page B3-220

B3.25 __attribute__((aligned)) type attribute

The `aligned` type attribute specifies a minimum alignment for the type.

B3.26 __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.

Note

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 which might be unaligned. The compiler assumes that the pointer is aligned. Dereferencing such a pointer can be unsafe even when unaligned accesses are supported by the target, because certain instructions always require word-aligned addresses.

Note

If you take the address of a packed member, in most cases, the compiler 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 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 B3-2 Examples

```
struct __attribute__((packed)) foobar
{
  char x;
  short y;
};

short get_y(struct foobar *s)
{
    // Correct usage: the compiler will not use unaligned accesses
    // unless they are allowed.
    return s->y;
}

short get2_y(struct foobar *s)
{
    short *p = &s->y; // Incorrect usage: 'p' might be an unaligned pointer.
    return *p;  // This might cause an unaligned access.
}
```

Related references

B1.65 -munaligned-access, -mno-unaligned-access on page B1-145

B3.27 __attribute__((transparent_union)) type attribute

The `transparent_union` type attribute enables you to specify a *transparent union type*.

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.

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 */
}
```

B3.28 Variable attributes

The `__attribute__` keyword enables you to specify special attributes of variables or structure fields, functions, and types.

The keyword format is either of the following:

```
__attribute__((attribute1, attribute2, ...))
__attribute__((__attribute1__, __attribute2__, ...))
```

For example:

```
static int b __attribute__((__unused__));
```

The available variable attributes are as follows:

- `__attribute__((alias))`
- `__attribute__((aligned("x")))`
- `__attribute__((deprecated))`
- `__attribute__((packed))`
- `__attribute__((section("name")))`
- `__attribute__((unused))`
- `__attribute__((used))`
- `__attribute__((visibility("visibility_type")))`
- `__attribute__((weak))`
- `__attribute__((weakref("target")))`

Related references

B3.29 __attribute__((alias)) variable attribute on page B3-223
B3.30 __attribute__((aligned)) variable attribute on page B3-224
B3.31 __attribute__((deprecated)) variable attribute on page B3-225
B3.32 __attribute__((packed)) variable attribute on page B3-226
B3.33 __attribute__((section("name"))) variable attribute on page B3-227
B3.34 __attribute__((unused)) variable attribute on page B3-228
B3.35 __attribute__((used)) variable attribute on page B3-229
B3.36 __attribute__((visibility("visibility_type"))) variable attribute on page B3-230
B3.37 __attribute__((weak)) variable attribute on page B3-231
B3.38 __attribute__((weakref("target"))) variable attribute on page B3-232

B3.29 __attribute__((alias)) variable attribute

This variable attribute enables you to specify multiple aliases for a variable.

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. For example:

```
int oldname = 1;
extern int newname __attribute__((alias("oldname")));
```

This code compiles to:

```
        .type   oldname,%object         @ @oldname
        .data
        .globl  oldname
        .align  2
oldname:
        .long   1                       @ 0x1
        .size   oldname, 4
        ...
        .globl  newname
newname = oldname
```

Note

Function names can also be aliased using the corresponding function attribute `__attribute__((alias))`.

Syntax

`type newname __attribute__((alias("oldname")));`

Where:

oldname

is the name of the variable to be aliased

newname

is the new name of the aliased variable.

Example

```
#include <stdio.h>
int oldname = 1;
extern int newname __attribute__((alias("oldname"))); // declaration
void foo(void){
    printf("newname = %d\n", newname); // prints 1
}
```

B3.30 __attribute__((aligned)) variable attribute

The `aligned` variable attribute specifies a minimum alignment for the variable or structure field, measured in bytes.

Example

```
/* Aligns on 16-byte boundary */
int x __attribute__((aligned (16)));

/* In this case, the alignment used is the maximum alignment for a scalar data type. For
ARM, this is 8 bytes. */
short my_array[3] __attribute__((aligned));
```

B3.31 __attribute__((deprecated)) variable attribute

The `deprecated` variable attribute enables the declaration of a deprecated variable without any warnings or errors being issued by the compiler. However, any access to a `deprecated` variable creates a warning but still compiles.

The warning gives the location where the variable is used and the location where it is defined. This helps you to determine why a particular definition is deprecated.

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.
```

B3.32 __attribute__((packed)) variable attribute

You can specify the `packed` variable attribute on fields that are members of a structure or union. It specifies that a member field has the smallest possible alignment. That is, one byte for a variable field, and one bit for a bitfield, unless you specify a larger value with the `aligned` attribute.

Example

```
struct
{
    char a;
    int b __attribute__((packed));
} Variable_Attributes_packed_0;
```

Note

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 which might be unaligned. The compiler assumes that the pointer is aligned. Dereferencing such a pointer can be unsafe even when unaligned accesses are supported by the target, because certain instructions always require word-aligned addresses.

Note

If you take the address of a packed member, in most cases, the compiler generates a warning.

Related references

B3.30 __attribute__((aligned)) variable attribute on page B3-224

B3.33 __attribute__((section("name"))) variable attribute

The `section` attribute specifies that a variable must be placed in a particular data section.

Normally, the Arm compiler places the 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.

To place ZI data in a named section, the section must start with the prefix `.bss.`. Non-ZI data cannot be placed in a section named `.bss`.

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")));
```

Note

Section names must be unique. You must not use the same section name for different section types. If you use the same section name for different section types, then the compiler merges the sections into one and gives the section the type of whichever function or variable is first allocated to that section.

Related tasks

C6.2.8 Manually placing __at sections on page C6-614

B3.34 __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.

Note

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 references

B3.18 __attribute__((unused)) function attribute on page B3-211

B3.35 __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.

Data marked with `__attribute__((used))` is tagged in the object file to avoid removal by linker unused section removal.

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 concepts

C4.2 Elimination of unused sections on page C4-563

B3.36 __attribute__((visibility("visibility_type"))) variable attribute

This variable attribute affects the visibility of ELF symbols.

Syntax

`__attribute__((visibility("`*`visibility_type`*`")))`

Where *`visibility_type`* is one of the following:

`default`

Default visibility corresponds to external linkage.

`hidden`

The symbol is not placed into the dynamic symbol table, so no other executable or shared library can directly reference it. Indirect references are possible using function pointers.

`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.

Usage

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.

Default

If you do not specify visibility, then the default type is `default` for **`extern`** declarations and `hidden` for everything else.

Example

```
int __attribute__((visibility("hidden"))) foo = 1;   // hidden in object file
```

Related references

B1.36 -fvisibility on page B1-97

B3.21 __attribute__((visibility("visibility_type"))) function attribute on page B3-215

B3.37 __attribute__((weak)) variable attribute

Generates a weak symbol for a variable, rather than the default symbol.

```
extern int foo __attribute__((weak));
```

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.

B3.38 __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")))
```

Where *`target`* is the target symbol.

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;
  ...
}
```

Restrictions

This attribute can only be used on variables that are declared as `static`.

Chapter B4
Compiler-specific Intrinsics

Summarizes the Arm compiler-specific intrinsics that are extensions to the C and C++ Standards.

To use these intrinsics, your source file must contain `#include <arm_compat.h>`.

It contains the following sections:

- *B4.1 __breakpoint intrinsic* on page B4-234.
- *B4.2 __current_pc intrinsic* on page B4-235.
- *B4.3 __current_sp intrinsic* on page B4-236.
- *B4.4 __disable_fiq intrinsic* on page B4-237.
- *B4.5 __disable_irq intrinsic* on page B4-238.
- *B4.6 __enable_fiq intrinsic* on page B4-239.
- *B4.7 __enable_irq intrinsic* on page B4-240.
- *B4.8 __force_stores intrinsic* on page B4-241.
- *B4.9 __memory_changed intrinsic* on page B4-242.
- *B4.10 __schedule_barrier intrinsic* on page B4-243.
- *B4.11 __semihost intrinsic* on page B4-244.
- *B4.12 __vfp_status intrinsic* on page B4-246.

B4.1 __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.

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);
    ...
}
```

B4.2 __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.

B4.3 __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.

B4.4 __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();
}
```

B4.5 __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.

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`.

B4.6 __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)
```

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`.

B4.7 __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`.

B4.8 __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)
```

B4.9 __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.

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)
```

B4.10 __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)
```

B4.11 __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.

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 Armv7-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 emits `SVC` instructions rather than `HLT` instructions for semihosting calls. If you define this macro, `__USE_HLT_SEMIHOSTING`, then Arm Compiler 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);
}
```

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

B4.12 __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 B4-1 Modifying the FPSCR flags

mask bit	*flags* bit	Effect on FPSCR flag
0	0	Does not modify the flag
0	1	Toggles the flag
1	1	Sets the flag
1	0	Clears the flag

Note

If you want to read or modify only the exception flags in FPSCR, then Arm recommends that you use the standard C99 features in `<fenv.h>`.

Errors

The compiler generates an error if you attempt to use this intrinsic when compiling for a target that does not have VFP.

Chapter B5
Compiler-specific Pragmas

Summarizes the Arm compiler-specific pragmas that are extensions to the C and C++ Standards.

It contains the following sections:

- *B5.1 #pragma clang system_header* on page B5-248.
- *B5.2 #pragma clang diagnostic* on page B5-249.
- *B5.3 #pragma clang section* on page B5-251.
- *B5.4 #pragma once* on page B5-253.
- *B5.5 #pragma pack(...)* on page B5-254.
- *B5.6 #pragma unroll[(n)], #pragma unroll_completely* on page B5-256.
- *B5.7 #pragma weak symbol, #pragma weak symbol1 = symbol2* on page B5-257.

B5.1 #pragma clang system_header

Causes subsequent declarations in the current file to be marked as if they occur in a system header file.

This pragma suppresses the warning messages that the file produces, from the point after which it is declared.

B5.2 #pragma clang diagnostic

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.

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.

#pragma clang diagnostic ignored

```
#pragma clang diagnostic ignored "-Wname"
```

This pragma disables the diagnostic message specified by *`name`*.

#pragma clang diagnostic warning

```
#pragma clang diagnostic warning "-Wname"
```

This pragma sets the diagnostic message specified by *`name`* to warning severity.

#pragma clang diagnostic error

```
#pragma clang diagnostic error "-Wname"
```

This pragma sets the diagnostic message specified by *`name`* to error severity.

#pragma clang diagnostic fatal

```
#pragma clang diagnostic fatal "-Wname"
```

This pragma sets the diagnostic message specified by *`name`* to fatal error severity. Fatal error causes compilation to fail without processing the rest of the file.

#pragma clang diagnostic push, #pragma clang diagnostic pop

```
#pragma clang diagnostic push
#pragma clang diagnostic pop
```

`#pragma clang diagnostic push` saves the current pragma diagnostic state so that it can restored later.

`#pragma clang diagnostic pop` restores the diagnostic state that was previously saved using `#pragma clang diagnostic push`.

Examples of 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.

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`.

Related references

B1.85 -W on page B1-168

B5.3 #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`*`]`

Where:

`section_type_list`

specifies 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`*.

Valid section types are:

- `bss`.
- `data`.
- `rodata`.
- `text`.

Usage

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.

- 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-section` and `-fdata-section` 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-section` and `-fdata-section` command-line options, then each global variable is in a unique section.

Example

```
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;
}
#pragma clang section bss="" data="" text="" // Use default name for bss, data, and text
sections.
```

B5.4 #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.

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.

B5.5 #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 supports.

Syntax

`#pragma pack([`*`n`*`])`

`#pragma pack(push[,`*`n`*`])`

`#pragma pack(pop)`

Where:

n

Is the alignment in bytes, valid alignment values are `1`, `2`, `4`, and `8`. If omitted, sets the alignment to the one that was in effect when compilation started.

`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.

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)`.

Default

The default is the alignment that was in effect when compilation started.

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:

0	1	2	3
a	padding		
4	5	6	7
b	b	b	b

Figure B5-1 Nonpacked structure S

The layout of `SP` is:

0	1	2	3
a	x	b	b
4	5		
b	b		

Figure B5-2 Packed structure SP

———— **Note** ————

In this layout, `x` denotes one byte of padding.

`SP` is a 6-byte structure. There is no padding after `b`.

————————————

B5.6 #pragma unroll[(n)], #pragma unroll_completely

Instructs the compiler to unroll a loop by *n* iterations.

Syntax

`#pragma unroll`

`#pragma unroll_completely`

`#pragma unroll n`

`#pragma unroll(n)`

Where:

n

is an optional value indicating the number of iterations to unroll.

Default

If you do not specify a value for *n*, the compiler attempts to fully unroll the loop. The compiler can only fully unroll loops where it can determine the number of iterations.

`#pragma unroll_completely` will not unroll a loop if the number of iterations is not known at compile time.

Usage

This pragma only has an effect with optimization level `-O2` and higher.

When compiling with `-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.

`#pragma unroll[(n)]` can be used immediately before a **`for`** loop, a **`while`** loop, or a **`do`** ... **`while`** loop.

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.

B5.7 #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.

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)
{
    ...
}
```

Chapter B6
Other Compiler-specific Features

Summarizes compiler-specific features that are extensions to the C and C++ Standards, such as predefined macros.

It contains the following sections:

- *B6.1 ACLE support* on page B6-260.
- *B6.2 Predefined macros* on page B6-261.
- *B6.3 Inline functions* on page B6-266.
- *B6.4 Half-precision floating-point data types* on page B6-267.
- *B6.5 Half-precision floating-point number format* on page B6-269.
- *B6.6 Half-precision floating-point intrinsics* on page B6-270.
- *B6.7 Library support for _Float16 data type* on page B6-271.
- *B6.8 BFloat16 floating-point number format* on page B6-272.
- *B6.9 TT instruction intrinsics* on page B6-273.
- *B6.10 Non-secure function pointer intrinsics* on page B6-276.

B6.1 ACLE support

Arm Compiler 6 supports the Arm C Language Extensions (ACLE) 2.1 with a few exceptions.

Note

This topic includes descriptions of [ALPHA] and [BETA] features. See *Support level definitions* on page A1-39.

Arm Compiler 6 does not support:

- `__ARM_ALIGN_MAX_PWR` macro.
- `__ARM_ALIGN_MAX_STACK_PWR` macro.
- `__cls` intrinsic.
- `__clsl` intrinsic.
- `__clsll` intrinsic.
- `__saturation_occurred` intrinsic.
- `__set_saturation_occurred` intrinsic.
- `__ignore_saturation` intrinsic.
- Patchable constants.
- Floating-point data-processing intrinsics.

Arm Compiler 6 does not model the state of the Q (saturation) flag correctly in all situations.

Additional supported intrinsics

Arm Compiler 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.
- [ALPHA] Support for the ACLE defined BFloat16 floating-point scalar and vector intrinsics in AArch64 and AArch32 states.
- [ALPHA] 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 M-profile Vector Extension (MVE) intrinsics.
- Support for the ACLE defined Transactional Memory Extension (TME) intrinsics.
- Support for these additional floating-point intrinsics:
  - `__arm_rsrf`
  - `__arm_wsrf`
  - `__arm_rsrf64`
  - `__arm_wsrf64`

For more information on ACLE 2.1, see the *ACLE 2.1* specification.

For updates on the latest ACLE intrinsics, see the *Arm C Language Extensions*.

For more information on intrinsics that use the Advanced SIMD registers, see the *Neon Intrinsics Reference*.

For more information on the MVE intrinsics, see the *MVE Intrinsics Reference*.

Related references

B6.6 Half-precision floating-point intrinsics on page B6-270

Related information

List of Neon intrinsics

List of MVE intrinsics

Arm C Language Extensions

B6.2 Predefined macros

The Arm Compiler 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 the Arm compiler 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 2.0* (ACLE 2.0).

Note

`armclang` does not fully implement ACLE 2.0.

Table B6-1 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.
`__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'.
`__ARM_BIG_ENDIAN`	-	Set if compiling for a big-endian target.
`__ARM_FEATURE_CLZ`	`1`	Set to 1 if the `CLZ` (count leading zeroes) instruction is supported in hardware.

Table B6-1 Predefined macros (continued)

Name	Value	When defined

`__ARM_FEATURE_CMSE`	*num*	Indicates the availability of the Armv8-M Security Extension related extensions:
		`0`
		The Armv8-M `TT` instruction is not available.
		`1`
		The `TT` instruction is available. It is not part of Armv8-M Security Extension, but is closely related.
		`3`
		The Armv8-M Security Extension for secure executable files is available. This implies that the `TT` instruction is available.
		See *B6.9 TT instruction intrinsics* on page B6-273 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 in ACLE 2.0 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_NUMERIC_MAXMIN`	`1`	Set to 1 if the target supports floating-point maximum and minimum instructions. Only available when `__ARM_ARCH >= 8`.
`__ARM_FEATURE_QBIT`	`1`	Set for targets in AArch32 state only. Set to 1 if the Q (saturation) flag exists. **Note** This macro is deprecated in ACLE 2.0 for A-profile.

Table B6-1 Predefined macros (continued)

Name	Value	When defined
`__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 in ACLE 2.0 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 in ACLE 2.0 for A-profile, use Arm Neon intrinsics instead.
`__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.
`__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`.

Table B6-1 Predefined macros (continued)

Name	Value	When defined
`__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, or 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 6.3 update 1 is displayed as `6030154`, 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__`.
`__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.
`__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 version information, use the `__ARMCOMPILER_VERSION` macro.
`__INTMAX_TYPE__`	*`type`*	Always set. Defines the correct underlying type for the `intmax_t` **`typedef`**.

Table B6-1 Predefined macros (continued)

Name	Value	When defined
`__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`	Set to 1 when compiling with `-mfloat-abi=softfp` for targets in AArch32 state. Set to 0 otherwise.
`__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`**.

Related references

B1.83 --version_number (armclang) on page B1-166
B1.77 -std on page B1-159
B1.70 -O (armclang) on page B1-150
B1.78 --target on page B1-161
B1.50 -marm on page B1-115
B1.66 -mthumb on page B1-146

B6.3 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, through the use of `-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. Qualifying a function with the **`__inline__`** or **`inline`** keywords suggests to the compiler that it inlines that function, but the final decision rests with the compiler. Qualifying a function with **`__attribute__((always_inline))`** forces the compiler to inline the 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. For inlining, it means that when you suggest a function is inlined, the compiler expects to find another, non-qualified, version of the function elsewhere in the code, to use when it decides not to inline. If the compiler cannot find the non-qualified version, 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, non-qualified version of the function.
- Change the qualifier to **`static inline`**.
- Remove the **`inline`** keyword, because it is only acting as a suggestion.
- Compile your program using the GNU C90 dialect, using the `-std=gnu90` option.

Related references

B2.8 __inline on page B2-183
B1.77 -std on page B1-159
B3.2 __attribute__((always_inline)) function attribute on page B3-195

B6.4 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 6 supports two half-precision (16-bit) floating-point scalar data types:

- The IEEE 754-2008 `__fp16` data type, defined in the Arm C Language Extensions.
- The `_Float16` data type, defined in the C11 extension ISO/IEC TS 18661-3:2015

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.

Arm recommends 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 implicit argument conversion 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.

```
extern void ReadFloatValue(float f);

void ReadValues(void)
{
    // Half-precision floating-point value stored in the _Float16 data type.
    const _Float16 h = 1.0f16;


    // There is no implicit argument conversion between _Float16 and standard floating-point
data types.
    // Therefore, this call to the ReadFloatValue() function below is not a call to the
declared function extern void ReadFloatValue(float f).
    ReadFloatValue(h);

    // An explicit cast is required for promoting a _Float16 value to a single-precision
floating-point value.
    // Therefore, this call to the ReadFloatValue() function below is a call to the declared
function extern void ReadFloatValue(float f).
    ReadFloatValue((float)h);

    return;
}
```

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
```

Related references

B1.49 -march on page B1-110

B1.56 -mcpu on page B1-125

B6.7 Library support for _Float16 data type on page B6-271

Related information

Using Assembly and Intrinsics in C or C++ Code

C language extensions

List of intrinsics

Arm C Language Extensions

B6.5 Half-precision floating-point number format

Arm Compiler 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).

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 uses the half-precision binary floating-point format defined by IEEE 754r, a revision to the IEEE 754 standard:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	E					T									

Figure B6-1 IEEE half-precision floating-point format

Where:

```
   S (bit[15]):      Sign bit
   E (bits[14:10]):  Biased exponent
   T (bits[9:0]):    Mantissa.
```

The meanings of these fields 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

B6.6 Half-precision floating-point intrinsics

Arm Compiler 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* on page A1-39.

To see the half-precision floating-point intrinsics, you can search for `float16` from the list of intrinsics on *Arm Developer*.

`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};

    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 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 references

B1.49 -march on page B1-110

B1.56 -mcpu on page B1-125

Related information

Using Assembly and Intrinsics in C or C++ Code

C language extensions

List of intrinsics

Arm C Language Extensions

B6.7 Library support for _Float16 data type

The C standard library in Arm Compiler 6 does not support the `_Float16` data type.

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.

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 references

B1.49 -march on page B1-110

B1.56 -mcpu on page B1-125

Related information

C language extensions

List of intrinsics

Arm C Language Extensions

B6.8 BFloat16 floating-point number format

Arm Compiler [ALPHA] supports the floating-point `__bf16` type.

Note

This topic includes descriptions of [ALPHA] features. See *Support level definitions* on page A1-39.

BFloat16 is a floating-point format that occupies 16 bits. It is [ALPHA] 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.

BFloat16 floating-point format

Arm Compiler uses the BFloat16 binary floating-point format which is a truncated form of the IEEE 754 standard.

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>S</td><td colspan="8">E</td><td colspan="7">T</td></tr>
</table>

Figure B6-2 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.

B6.9 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 `__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.

When `__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 `__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 Armv8-M Security Extension that are available when `__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 `__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 `__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*.

In Secure state, the `TT` instruction returns the *Security Attribute Unit* (SAU) and *Implementation Defined Attribute 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.

(continued)

Macro	Value	Description
`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.

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.

Arm recommends 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 `__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)`

Arm recommends 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 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 references

B6.2 Predefined macros on page B6-261

B6.10 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 `__ARM_FEATURE_CMSE` is set to `3`:

Table B6-2 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. Arm recommends that the return type of this intrinsic is identical to the type of its argument.
`cmse_is_nsfptr(p)`	Returns non-zero if `p` has LSB unset, zero otherwise. The argument `p` can be any function pointer type.

Example

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 references

B3.3 __attribute__((cmse_nonsecure_call)) function attribute on page B3-196

B3.4 __attribute__((cmse_nonsecure_entry)) function attribute on page B3-197

Related information

Building Secure and Non-secure Images Using the Armv8-M Security Extension

Chapter B7
armclang Integrated Assembler

Provides information on integrated assembler features, such as the directives you can use when writing assembly language source files in the `armclang` integrated assembler syntax.

It contains the following sections:

- *B7.1 Syntax of assembly files for integrated assembler* on page B7-278.
- *B7.2 Assembly expressions* on page B7-280.
- *B7.3 Alignment directives* on page B7-285.
- *B7.4 Data definition directives* on page B7-287.
- *B7.5 String definition directives* on page B7-290.
- *B7.6 Floating-point data definition directives* on page B7-292.
- *B7.7 Section directives* on page B7-293.
- *B7.8 Conditional assembly directives* on page B7-299.
- *B7.9 Macro directives* on page B7-301.
- *B7.10 Symbol binding directives* on page B7-303.
- *B7.11 Org directive* on page B7-305.
- *B7.12 AArch32 Target selection directives* on page B7-306.
- *B7.13 AArch64 Target selection directives* on page B7-308.
- *B7.14 Space-filling directives* on page B7-309.
- *B7.15 Type directive* on page B7-310.
- *B7.16 Integrated assembler support for the CSDB instruction* on page B7-311.

B7.1 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

The optional `;` can be used to end the statement and start a new statement on the same line.

directive

The optional `;` can be used to end the statement and start a new statement on the same line.

macro_invocation

The optional `;` can be used to end the statement and start a new statement on the same line.

Comments

Comments are treated as equivalent to whitespace, their contents are ignored by the assembler.

There are two 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 when compiling C and C++ source, the end of the line cannot be escaped with `\` to continue the comment.

`@` 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 in C and C++ source, the comment always ends at the first `*/`, so comments cannot be nested. This style of comments can be used anywhere within an assembly statement where whitespace is valid.

Examples

```
// Instruction on it's 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
```

B7.2 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.

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 B7-1 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.

Table B7-1 Modifiers (continued)

Modifier	Meaning
`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:

Table B7-2 Unary operators

Unary operator	Meaning
`-`*`expr`*	Arithmetic negation of *`expr`*.
`+`*`expr`*	Arithmetic addition of *`expr`*.
`~`*`expr`*	Bitwise negation of *`expr`*.

Table B7-3 Binary operators

Binary operator	Meaning
`expr1` `-` *`expr2`*	Subtraction.
`expr1` `+` *`expr2`*	Addition.
`expr1` `*` *`expr2`*	Multiplication.
`expr1` `/` *`expr2`*	Division.
`expr1` `%` *`expr2`*	Modulo.

Table B7-4 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 B7-5 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 B7-6 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. `==`, `!=`, `<`, `>`, `<=`, `>=`
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 some instruction operands, a relocation specifier might be used to specify which bits of the expression should be used for the operand, and which type of relocation should be used.

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 B7-7 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.

These 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.

In AArch64 state, the following relocation specifiers are available:

Table B7-8 Relocation specifiers for AArch64 state

Relocation specifier	Relocation type	Bits to use	Overflow checked
`:lo12:`	Absolute	[11:0]	No
`: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.

Examples

```
    // Using an absolute expression in an instruction operand:
    orr r0, r0, #1<<23

    // Using an expression in the memory operand of an LDR instruction to
    // reference an offset from a symbol.
  func:
    ldr r0, #data+4 // Will load 2 into r0
    bx lr
  data:
    .word 1
    .word 2

    // Creating initialized data that contains the distance between two
    // labels:
  size:
    .word end - start
  start:
    .word 123
    .word 42
    .word 4523534
  end:

    // Load the base-relative address of 'sym' (used for 'RWPI'
    // position-independent code) into r0 using movw and movt:
    movw r0, #:lower16:sym(sbrel)
    movt r0, #:upper16:sym(sbrel)

    // Load the address of 'sym' from the GOT using ADRP and LDR (used for
    // position-independent code on AArch64):
    adrp x0, #:got:sym
    ldr x0, [x0, #:got_lo12:sym]

    // 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 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 - .
```

B7.3 Alignment directives

The alignment directives align the current location in the file to a specified boundary.

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]
```

```
.align exponent [, fill_value]
```

Description

num_bytes

This specifies the number of bytes that must be aligned to. This must be a power of 2.

exponent

This 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.

Operation

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. The width of *fill_value* can be controlled with the `w` and `l` suffixes, see below.
- 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.

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`

The `w` and `l` suffixes modify the width of the padding value that will be inserted.

- By default, the *fill_value* is a 1-byte value.
- If the `w` suffix is specified, the *fill_value* is a 2-byte value.
- If the `l` suffix is specified, the *fill_value* is a 4-byte value.

If either of these suffixes are specified, 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 will be increased.

Usage

Use the alignment directives to ensure that your data and code are aligned to appropriate boundaries. This 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.

Examples

Aligning a constant pool value to a 4-byte boundary in T32 code:

```
get_val:
  ldr r0, value
  adds r0, #1
  bx lr
  // The above code is 6 bytes in size.
  // Therefore the data defined by the .word directive below must be manually aligned
  // to a 4-byte boundary to be able to use the LDR instruction.
  .p2align 2
value:
  .word 42
```

Ensuring 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
```

Note

In both of the examples above, 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.

B7.4 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]…
```

```
.word  expr[, expr]…
```

```
.quad  expr[, expr]…
```

```
.octa  expr[, expr]…
```

Description

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 may or 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 may 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:
```

The number of bytes allocated by each directive is as follows:

Table B7-9 Data definition directives

Directive	Size in bytes
`.byte`	1
`.hword`	2
`.word`	4
`.quad`	8
`.octa`	16

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 B7-10 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 B7-11 Aliases for the data definition directives

Directive	Aliases
`.byte`	`.1byte`, `.dc.b`
`.hword`	`.2byte`, `.dc`, `.dc.w`, `.short`, `.value`
`.word`	`.4byte`, `.long`, `.int`, `.dc.l`, `.dc.a` (AArch32 only)
`.quad`	`.8byte`, `.xword` (AArch64 only), `.dc.a` (AArch64 only)

Examples

```
// 8-bit memory location, initialized to 42:
.byte 42

// 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

// 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

// 32-bit memory location, containing the offset between the current location in the file
```

```
and an externally defined symbol.
  .word extern_symbol - .
```

B7.5 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"
```

Description

`.ascii`

The `.ascii` directive does not append a null byte to the end of the string.

`.asciz`

The `.asciz` directive appends a null byte to the end of the string.

The `.string` directive is an alias for `.asciz`.

`string`

The following escape characters are accepted in the string literal:

Table B7-12 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

Examples

Using a null-terminated string in a constant pool:

```
      .text
    hello:
      adr r0, str_hello
      b printf
    str_hello:
      .asciz "Hello, world!\n"
```

Generating pascal-style strings (which are prefixed by a length byte, and have no null terminator), using a macro to avoid repeated code (see also *macros* on page B7-301 and *temporary numeric labels*).

```
      .macro pascal_string, str
      .byte 2f - 1f
    1:
      .ascii "\str"
    2:
      .endm
```

```
    .data
  hello:
    pascal_string "Hello"
  goodbye:
    pascal_string "Goodbye"
```

B7.6 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]…
```

Description

`.float`

The `.float` directive allocates 4 bytes of memory per argument, and stores the values in IEEE754 single-precision format.

`.double`

The `.double` directive allocates 8 bytes of memory per argument, and stores the values in IEEE754 double-precision format.

`value`

`value` is a floating-point literal.

Operation

If a floating-point value cannot be exactly represented by the storage format, it will be rounded to the nearest representable value using the `round to nearest, ties to even` rounding mode.

The following aliases for these directives are also accepted:

Table B7-13 Aliases for the floating-point data definition directives

Directive	Alias
`.float`	`.single`, `.dc.s`
`.double`	`.dc.d`

Examples

```
float_pi:
  .float 3.14159265359
double_pi:
  .double 3.14159265359
```

B7.7 Section directives

The section directives instruct the assembler to change the ELF section that code and data are emitted into.

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
```

Description

name

The *name* argument gives 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 will switch back to that section. Any code or data that is assembled will be appended to the end of that section. The unique-id argument can be used to override this behavior.

`flags`

The optional `flags` argument is a quoted string containing any of the following characters, which correspond to the sh_flags field in the ELF section header.

Table B7-14 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 is mergeable.
`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 ignored.

The flags can be specified as a numeric value, with the same encoding as the *sh_flags field* in the ELF section header. This cannot be combined with the flag characters listed above. 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 B7-15 Section Type

Argument	ELF type	Meaning
`%progbits`	`SHT_PROGBITS`	Section contains initialized data and/or instructions.
`%nobits`	`SHT_NOBITS`	Section consists only of zero-initialized data.
`%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 field* 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 should be a constant expression which evaluates to a positive integer. If a section has previously been created with the same name and unique ID, then the assembler will switch to the existing section, appending content to it. Otherwise, a new section is created. Sections without a unique ID specified will never be merged with sections that do have one. This 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 rest of the directives (`.text`, `.data`, `.rodata`, `.bss`) switch to one of the built-in sections.

If continuing a previous section, and the flags, type, or other arguments do not match the previous definition of the section, then the arguments of the current `.section` directive will have no effect on the section. Instead, the assembler uses the arguments from the previous `.section` directive. The assembler does not currently emit a diagnostic when this happens.

Default

Some section names and section name prefixes implicitly have some flags set. Additional flags can be set using the flags argument, but it is not possible to clear these implicit flags. The section names that have implicit flags are listed in the table here. 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 table here, the default section type is `%progbits`.

Table B7-16 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`

Table B7-16 Sections with implicit flags and default types (continued)

Section name	Implicit Flags	Default Type
`.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`

Examples

Splitting code and data into the built-in `.text` and `.data` sections. The linker can place these 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
```

Creating 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
```

Creating two sections with the same name:

```
  .section .data, "aw", %progbits, unique, 1
  .word 1
  .section .data, "aw", %progbits, unique, 2
  .word 2
```

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 further 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
```

B7.8 Conditional assembly directives

These directives allow you to conditionally assemble sequences of instructions and directives.

Syntax

```
.if[modifier] expression
    // ...
  [.elseif expression
    // ...]
  [.else
    // ...]
.endif
```

Operation

There are a number of different forms of the `.if` directive which 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 following forms if the `.if` directive are available, which check different conditions:

Table B7-17 .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.
`.ifc` *`string1 string2`*	Assembles the following code if the two strings are the same. The strings may 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 may 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.

Table B7-17 .if condition modifiers (continued)

.if condition modifier	Meaning
`.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.

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.

Examples

```
    // A macro to load an immediate value into a register. This 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:
    get_imm r0, 42
    get_imm r1, 0x12345678
    bx lr
```

B7.9 Macro directives

The `.macro` directive defines a new macro.

Syntax

```
.macro macro_name [, parameter_name]…
  // …
  [.exitm]
.endm
```

Description

`macro_name`

The name of the macro.

`parameter_name`

Inside the body of a macro, the parameters can be referred to by their name, prefixed with `\`. When the macro is instantiated, parameter references will be expanded to the value of the argument.

Parameters can be qualified in these ways:

Table B7-18 Macro parameter qualifier

Parameter qualifier	Meaning
`<name>:req`	This marks the parameter as required, it is an error to instantiate the macro with a blank value for this parameter.
`<name>:varag`	This parameter consumes all remaining arguments in the instantiation. If used, this must be the last parameter.
`<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 will be used.

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.

Once a macro is defined, it can be instantiated 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. This can be used to create unique label names, which will not interfere with other instantiations of the same macro.

The `.exitm` directive allows exiting a macro instantiation before reaching the end.

Examples

```
  // Macro for defining global variables, with the symbol binding, type and
  // size set appropriately. The 'value' parameter can be omitted, 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
```

B7.10 Symbol binding directives

These directives modify the ELF binding of one or more symbols.

Syntax

```
.global symbol[, symbol]…
```

```
.local symbol[, symbol]…
```

```
.weak symbol[, symbol]…
```

Description

`.global`

The `.global` directive sets the symbol binding to STB_GLOBAL. These symbols will be visible to all object files being linked, so a definition in one object file can satisfy a reference in another.

`.local`

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.

`.weak`

The `.weak` directive sets the symbol binding to STB_WEAK. These symbols behave similarly to global symbols, with these differences:

- If a reference to a symbol with weak binding is not satisfied (no definition of the symbol is found), this is not an error.
- If multiple definitions of a weak symbol are present, this is not an error. If a definition of the symbol with strong binding is present, that one will satisfy all references to the symbol, otherwise one of the weak references will be chosen.

Operation

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 will by default have global visibility.
- If a symbol is defined in the assembly file, it will by default have local visibility.

Note

`.local` and `.L` are different directives. Symbols starting with `.L` are not put into the symbol table.

Examples

```
    // This function has global binding, so 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 will search 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 will be relocated to point to it. If a
    // definition is not found (e.g. in a release build, which does not include
    // the debug code), the linker will point the bl instruction at the next
    // instruction, so it has no effect.
    .weak debug_trace
    bl debug_trace
```

B7.11 Org directive

The .org directive advances the location counter in the current section to new-location.

Syntax

```
.org new_location [, fill_value]
```

Description

new_location

The *new_location* argument 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.
- 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

This is 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 will be repeated in each skipped location.

Examples

```
    // 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
    // will be inserted. If the code is longer than that, the .org directive
    // will report 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...
```

B7.12 AArch32 Target selection directives

The AArch32 target selection directives specify code generation parameters for AArch32 targets.

Syntax

```
.arm
```

```
.thumb
```

```
.arch arch_name
```

```
.cpu cpu_name
```

```
.fpu fpu_name
```

```
.arch_extension extension_name
```

```
.eabi_attribute tag, value
```

Description

`.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 that the assembler is generating instructions for. The `arch_name` argument accepts the same names as the `-march` option, but does not accept the optional architecture extensions accepted by the command-line option.

`.cpu`

The `.cpu` directive changes the CPU that the assembler is generating instructions for. The `cpu_name` argument accepts the same names as the `-mcpu` option, but does not accept the optional architecture extensions accepted by the command-line option.

`.fpu`

The `.fpu` directive changes the FPU that the assembler is generating instructions for. The `fpu_name` argument accepts the same names as the `-mfpu` option.

`.arch_extension`

The `.arch_extension` enables or disables optional extensions to the architecture or CPU that the assembler is generating instructions for. It accepts the following optional extensions, which can be prefixed with `no` to disable them:

- `crc`
- `fp16`
- `ras`

`.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 have any effect on which instructions the assembler will accept. It is recommended that the `.arch`, `.cpu`, `.fpu` and `.arch_extension` directives are used where possible, as they also check that no instructions not valid 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.

`tag`

The tag argument specifies the tag that is to be set. This can either be the tag name or tag number, but not both.

`value`

The value argument specifies the value to set for the *`tag`*. The value can either be of integer or string type. The type must match exactly the type expected for that tag.

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
```

Examples

```
// Generate code for the Armv7-M architecture:
.arch armv7-m

// Generate code for the Cortex-R5, without an FPU:
.cpu cortex-r5
.fpu none

// Generate code for Armv8.2-A with the FP16 extension:
.arch armv8.2-a
.fpu neon-fp-armv8
.arch_extension fp16
```

B7.13 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]…
```

Description

`.arch`

The `.arch` directive changes the architecture that the assembler is generating instructions for.

The *`arch_name`* argument accepts the same names as the `-march` option, and accepts certain optional architecture extensions (*`extension`*) separated by `+`. The *`extension`* can be prefixed with `no` to disable it.

`.cpu`

The `.cpu` directive changes the CPU that the assembler is generating instructions for.

The *`cpu_name`* argument accepts the same names as the `-mcpu` option, and accepts certain optional architecture extensions (*`extension`*) separated by `+`. The *`extension`* can be prefixed with `no` to disable it.

`extension`

Optional architecture extensions. The accepted architecture extensions are:

- `crc`
- `crypto`
- `fp`
- `ras`
- `simd`

Examples

```
// Generate code for Armv8-A without a floating-point unit. The assembler
// will report an error if any instructions following this directive require
// the floating-point unit.
.arch armv8-a+nofp
```

B7.14 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]]
```

Description

`.space`

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.

`.fill`

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.

B7.15 Type directive

The `.type` directive sets the type of a symbol.

Syntax

```
.type symbol, %type
```

Description

`.type`

The `.type` directive sets the type of a symbol.

`symbol`

The symbol name to set the type for.

`%type`

The following types are accepted:

- `%function`
- `%object`
- `%tls_object`

Examples

```
  // 'func' is a function
  .type func, %function
func:
  bx lr

  // 'value' is a data object:
  .type value, %object
value:
  .word 42
```

B7.16 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

Chapter B8
armclang Inline Assembler

Provides reference information on writing inline assembly.

It contains the following sections:

- *B8.1 Inline Assembly* on page B8-314.
- *B8.2 File-scope inline assembly* on page B8-315.
- *B8.3 Inline assembly statements within a function* on page B8-316.
- *B8.4 Inline assembly constraint strings* on page B8-320.
- *B8.5 Inline assembly template modifiers* on page B8-325.
- *B8.6 Forcing inline assembly operands into specific registers* on page B8-328.
- *B8.7 Symbol references and branches into and out of inline assembly* on page B8-329.
- *B8.8 Duplication of labels in inline assembly statements* on page B8-330.

B8.1 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:

- 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 *Chapter B7 armclang Integrated Assembler* on page B7-277.

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 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.

For large blocks of assembly code where the overhead of calling between C or C++ and assembly is not significant, Arm recommends using a separate assembly file, which does not have these limitations.

Related references

B8.2 File-scope inline assembly on page B8-315

B8.3.2 Output and input operands on page B8-317

Related information

Optimizing across modules with link time optimization

B8.2 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.

Examples

```
// Simple file-scope inline assembly.
__asm(".global __use_realtime_heap");

// Multiple file-scope inline assembly statements in one block:
__asm("add_ints:\n"
     "  add r0, r0, r1\n"
     "  bx lr");

// 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
)");
```

B8.3 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 or directives that are available in the integrated assembler in the assembly string. 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 will be 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. It 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> ] ] ]
    );
```

This section contains the following subsections:
- *B8.3.1 Assembly string* on page B8-316.
- *B8.3.2 Output and input operands* on page B8-317.
- *B8.3.3 Clobber list* on page B8-318.
- *B8.3.4 volatile* on page B8-319.

B8.3.1 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.

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 *B8.5 Inline assembly template modifiers* on page B8-325.

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 *B8.8 Duplication of labels in inline assembly statements* on page B8-330.

B8.3.2 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>)
```

Where:

`<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 will be 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 *B8.4.1 Constraint modifiers* on page B8-320.
- Whether to store the value in a register or memory, or whether it is a compile-time constant. See *B8.4.2 Constraint codes* on page B8-320.

`<value>`
Is a C/C++ value that the operand corresponds to. For output operands, this value must be a writable value.

Example

```
foo.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 will be 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
          // should 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 will place 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 foo.c -o foo.s
```

The assembly language source file `foo.s` that is generated contains:

```
    .section    .text.saturating_add,"ax",%progbits
    .hidden saturating_add            @ -- Begin function saturating_add
    .globl  saturating_add
    .p2align    2
    .type   saturating_add,%function
    .code   32                        @ @saturating_add
saturating_add:
    .fnstart
@ %bb.0:                            @ %entry
    @APP
    qadd r0,r0,r1
    @NO_APP
    bx lr
.Lfunc_end0:
    .size saturating_add, .Lfunc_end0-saturating_add
    .cantunwind
    .fnend
```

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]`.

B8.3.3 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.

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");
}
```

B8.3.4 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.

The compiler assumes that any inline assembly statement with no output operands is volatile, even if the keyword is not present. However, Arm recommends that you still use it for clarity, and to avoid a behavior change if an output is added later.

Example

```
// Example where the volatile keyword is required. If the volatile keyword
// was omitted, this appears to still work. However, if the compiler were to
// inline it into a function that does not use the return value (old_table),
// then the inline assembly statement would appear 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 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;
}
```

B8.4 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.

This section contains the following subsections:
- *B8.4.1 Constraint modifiers* on page B8-320.
- *B8.4.2 Constraint codes* on page B8-320.
- *B8.4.3 Constraint codes common to AArch32 state and AArch64 state* on page B8-321.
- *B8.4.4 Constraint codes for AArch32 state* on page B8-321.
- *B8.4.5 Constraint codes for AArch64 state* on page B8-323.
- *B8.4.6 Using multiple alternative operand constraints* on page B8-324.

B8.4.1 Constraint modifiers

All output operands require a constraint modifier. There are currently no supported constraint modifiers for input operands.

Table B8-1 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.

B8.4.2 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 will be 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 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.

B8.4.3 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.

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. Arm recommends 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 *B8.5.2 Template modifiers for AArch32 state* on page B8-325.

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

Arm recommends 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.

B8.4.4 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 32-bit floating-point type.

The compiler can use S0-S31, D0-D15, or Q0-Q7.

`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 may 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 may be less than the number implied by the ranges above. This number may 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/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 *B8.5.2 Template modifiers for AArch32 state* on page B8-325.

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 (see *B8.4.6 Using multiple alternative operand constraints* on page B8-324) 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.

Constraint code	16-bit T32 instructions	32-bit T32 instructions	A32 instructions
`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.

B8.4.5 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-W31 instead, use the `w` template modifier.

`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.

Constants

`z`

A constant with value zero, printed as the zero register (XZR or WZR). Useful when combined with `r` (see *B8.4.6 Using multiple alternative operand constraints* on page B8-324) 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.

`K`

An immediate that is valid for 32-bit logical instructions. For example, AND, ORR, EOR.

`L`

An immediate that is valid for 64-bit logical instructions. For example, AND, ORR, 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 references

B8.5 Inline assembly template modifiers on page B8-325

B8.4.6 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

```
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 above, the call to add will be 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 is not usable 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);
}
```

B8.5 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. They 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.

This section contains the following subsections:
- *B8.5.1 Template modifiers common to AArch32 state and AArch64 state* on page B8-325.
- *B8.5.2 Template modifiers for AArch32 state* on page B8-325.
- *B8.5.3 Template modifiers for AArch64 state* on page B8-326.

B8.5.1 Template modifiers common to AArch32 state and AArch64 state

The following template modifiers are common to both AArch32 state and AArch64 state.

`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

```
// 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;
}
```

B8.5.2 Template modifiers for AArch32 state

The following template modifiers are specific to AArch32 state.

`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

```
// 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"
        : [new] "+&r" (old_val),
          [temp] "=&r" (temp)
        : [old] "r"   (new_val),
          [addr] "m"  (*addr));
    return old_val;
}
```

B8.5.3 Template modifiers for AArch64 state

The following template modifiers 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.

`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

```
// In AArch64 state, the 's' template modifiers cause 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;
}
```

B8.6 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 may 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

```
// 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;
}
```

B8.7 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.

It is valid to call a function from inside inline assembly, as that function will return control-flow back to the inline assembly code.

Arm does 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.

Arm recommends 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

```
static void foo(void) { /* ... */ }

// This function attempts to call the function foo from inside inline assembly.
// In some situations this may 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 bar() {
    __asm volatile(
          "bl foo"
        : /* no outputs */
        : /* no inputs */
        : "r0", "r1", "r2", "r3", "r12", "lr");
}

// This function is the same as above, except it passes a reference to foo into
// the inline assembly as an operand. This lets the compiler know about the
// reference, so the definition of foo will not be removed (unless, the
// definition of bar_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 (_ZL3foov) which would have to be used if writing the
// symbol name directly in the assembly string.
void bar_fixed() {
    __asm volatile(
          "bl %[foo]"
        : /* no outputs */
        : [foo] "i" (foo)
        : "r0", "r1", "r2", "r3", "r12", "lr");
}
```

B8.8 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. You can therefore 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

```
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.
    // R is a C++11 raw string literal. See the example in B8.2 File-scope inline assembly
 on page B8-315.
    __asm(R"(
        .Lloop%=:
            ldr %[tmp], %[src], #4
            str %[tmp], %[dst], #4
            subs %[len], #4
            bne .Lloop%=)"
        : [dst] "=&m" (*dst),
            [tmp] "=&r" (tmp),
            [len] "+r" (len)
        : [src] "m" (*src));
}
```

Part C

armlink Reference

Chapter C1
armlink Command-line Options

Describes the command-line options supported by the Arm linker, `armlink`.

It contains the following sections:

- *C1.1 --any_contingency* on page C1-337.
- *C1.2 --any_placement=algorithm* on page C1-338.
- *C1.3 --any_sort_order=order* on page C1-340.
- *C1.4 --api, --no_api* on page C1-341.
- *C1.5 --autoat, --no_autoat* on page C1-342.
- *C1.6 --bare_metal_pie* on page C1-343.
- *C1.7 --base_platform* on page C1-344.
- *C1.8 --bestdebug, --no_bestdebug* on page C1-346.
- *C1.9 --blx_arm_thumb, --no_blx_arm_thumb* on page C1-347.
- *C1.10 --blx_thumb_arm, --no_blx_thumb_arm* on page C1-348.
- *C1.11 --bpabi* on page C1-349.
- *C1.12 --branchnop, --no_branchnop* on page C1-350.
- *C1.13 --callgraph, --no_callgraph* on page C1-351.
- *C1.14 --callgraph_file=filename* on page C1-353.
- *C1.15 --callgraph_output=fmt* on page C1-354.
- *C1.16 --callgraph_subset=symbol[,symbol,...]* on page C1-355.
- *C1.17 --cgfile=type* on page C1-356.
- *C1.18 --cgsymbol=type* on page C1-357.
- *C1.19 --cgundefined=type* on page C1-358.
- *C1.20 --comment_section, --no_comment_section* on page C1-359.
- *C1.21 --cppinit, --no_cppinit* on page C1-360.
- *C1.22 --cpu=list (armlink)* on page C1-361.
- *C1.23 --cpu=name (armlink)* on page C1-362.

- *C1.24 --crosser_veneershare, --no_crosser_veneershare* on page C1-365.
- *C1.25 --datacompressor=opt* on page C1-366.
- *C1.26 --debug, --no_debug* on page C1-367.
- *C1.27 --diag_error=tag[,tag,…] (armlink)* on page C1-368.
- *C1.28 --diag_remark=tag[,tag,…] (armlink)* on page C1-369.
- *C1.29 --diag_style={arm|ide|gnu} (armlink)* on page C1-370.
- *C1.30 --diag_suppress=tag[,tag,…] (armlink)* on page C1-371.
- *C1.31 --diag_warning=tag[,tag,…] (armlink)* on page C1-372.
- *C1.32 --dll* on page C1-373.
- *C1.33 --dynamic_linker=name* on page C1-374.
- *C1.34 --eager_load_debug, --no_eager_load_debug* on page C1-375.
- *C1.35 --eh_frame_hdr* on page C1-376.
- *C1.36 --edit=file_list* on page C1-377.
- *C1.37 --emit_debug_overlay_relocs* on page C1-378.
- *C1.38 --emit_debug_overlay_section* on page C1-379.
- *C1.39 --emit_non_debug_relocs* on page C1-380.
- *C1.40 --emit_relocs* on page C1-381.
- *C1.41 --entry=location* on page C1-382.
- *C1.42 --errors=filename* on page C1-383.
- *C1.43 --exceptions, --no_exceptions* on page C1-384.
- *C1.44 --export_all, --no_export_all* on page C1-385.
- *C1.45 --export_dynamic, --no_export_dynamic* on page C1-386.
- *C1.46 --filtercomment, --no_filtercomment* on page C1-387.
- *C1.47 --fini=symbol* on page C1-388.
- *C1.48 --first=section_id* on page C1-389.
- *C1.49 --force_explicit_attr* on page C1-390.
- *C1.50 --force_so_throw, --no_force_so_throw* on page C1-391.
- *C1.51 --fpic* on page C1-392.
- *C1.52 --fpu=list (armlink)* on page C1-393.
- *C1.53 --fpu=name (armlink)* on page C1-394.
- *C1.54 --got=type* on page C1-395.
- *C1.55 --gnu_linker_defined_syms* on page C1-396.
- *C1.56 --help (armlink)* on page C1-397.
- *C1.57 --import_cmse_lib_in=filename* on page C1-398.
- *C1.58 --import_cmse_lib_out=filename* on page C1-399.
- *C1.59 --import_unresolved, --no_import_unresolved* on page C1-400.
- *C1.60 --info=topic[,topic,…] (armlink)* on page C1-401.
- *C1.61 --info_lib_prefix=opt* on page C1-404.
- *C1.62 --init=symbol* on page C1-405.
- *C1.63 --inline, --no_inline* on page C1-406.
- *C1.64 --inline_type=type* on page C1-407.
- *C1.65 --inlineveneer, --no_inlineveneer* on page C1-408.
- *C1.66 input-file-list (armlink)* on page C1-409.
- *C1.67 --keep=section_id (armlink)* on page C1-410.
- *C1.68 --keep_intermediate* on page C1-412.
- *C1.69 --largeregions, --no_largeregions* on page C1-413.
- *C1.70 --last=section_id* on page C1-415.
- *C1.71 --legacyalign, --no_legacyalign* on page C1-416.
- *C1.72 --libpath=pathlist* on page C1-417.
- *C1.73 --library=name* on page C1-418.
- *C1.74 --library_security=protection* on page C1-419.
- *C1.75 --library_type=lib* on page C1-421.
- *C1.76 --list=filename* on page C1-422.
- *C1.77 --list_mapping_symbols, --no_list_mapping_symbols* on page C1-423.
- *C1.78 --load_addr_map_info, --no_load_addr_map_info* on page C1-424.
- *C1.79 --locals, --no_locals* on page C1-425.

- *C1.80 --lto, --no_lto* on page C1-426.
- *C1.81 --lto_keep_all_symbols, --no_lto_keep_all_symbols* on page C1-428.
- *C1.82 --lto_intermediate_filename* on page C1-429.
- *C1.83 --lto_level* on page C1-430.
- *C1.84 --lto_relocation_model* on page C1-432.
- *C1.85 --mangled, --unmangled* on page C1-433.
- *C1.86 --map, --no_map* on page C1-434.
- *C1.87 --max_er_extension=size* on page C1-435.
- *C1.88 --max_veneer_passes=value* on page C1-436.
- *C1.89 --max_visibility=type* on page C1-437.
- *C1.90 --merge, --no_merge* on page C1-438.
- *C1.91 --merge_litpools, --no_merge_litpools* on page C1-439.
- *C1.92 --muldefweak, --no_muldefweak* on page C1-440.
- *C1.93 -o filename, --output=filename (armlink)* on page C1-441.
- *C1.94 --output_float_abi=option* on page C1-442.
- *C1.95 --overlay_veneers* on page C1-443.
- *C1.96 --override_visibility* on page C1-444.
- *C1.97 -Omax (armlink)* on page C1-445.
- *C1.98 --pad=num* on page C1-446.
- *C1.99 --paged* on page C1-447.
- *C1.100 --pagesize=pagesize* on page C1-448.
- *C1.101 --partial* on page C1-449.
- *C1.102 --pie* on page C1-450.
- *C1.103 --piveneer, --no_piveneer* on page C1-451.
- *C1.104 --pixolib* on page C1-452.
- *C1.105 --pltgot=type* on page C1-454.
- *C1.106 --pltgot_opts=mode* on page C1-455.
- *C1.107 --predefine="string"* on page C1-456.
- *C1.108 --preinit, --no_preinit* on page C1-457.
- *C1.109 --privacy (armlink)* on page C1-458.
- *C1.110 --ref_cpp_init, --no_ref_cpp_init* on page C1-459.
- *C1.111 --ref_pre_init, --no_ref_pre_init* on page C1-460.
- *C1.112 --reloc* on page C1-461.
- *C1.113 --remarks* on page C1-462.
- *C1.114 --remove, --no_remove* on page C1-463.
- *C1.115 --ro_base=address* on page C1-464.
- *C1.116 --ropi* on page C1-465.
- *C1.117 --rosplit* on page C1-466.
- *C1.118 --rw_base=address* on page C1-467.
- *C1.119 --rwpi* on page C1-468.
- *C1.120 --scanlib, --no_scanlib* on page C1-469.
- *C1.121 --scatter=filename* on page C1-470.
- *C1.122 --section_index_display=type* on page C1-472.
- *C1.123 --shared* on page C1-473.
- *C1.124 --show_cmdline (armlink)* on page C1-474.
- *C1.125 --show_full_path* on page C1-475.
- *C1.126 --show_parent_lib* on page C1-476.
- *C1.127 --show_sec_idx* on page C1-477.
- *C1.128 --soname=name* on page C1-478.
- *C1.129 --sort=algorithm* on page C1-479.
- *C1.130 --split* on page C1-481.
- *C1.131 --startup=symbol, --no_startup* on page C1-482.
- *C1.132 --stdlib* on page C1-483.
- *C1.133 --strict* on page C1-484.
- *C1.134 --strict_flags, --no_strict_flags* on page C1-485.
- *C1.135 --strict_ph, --no_strict_ph* on page C1-486.

- *C1.136 --strict_preserve8_require8* on page C1-487.
- *C1.137 --strict_relocations, --no_strict_relocations* on page C1-488.
- *C1.138 --strict_symbols, --no_strict_symbols* on page C1-489.
- *C1.139 --strict_visibility, --no_strict_visibility* on page C1-490.
- *C1.140 --symbols, --no_symbols* on page C1-491.
- *C1.141 --symdefs=filename* on page C1-492.
- *C1.142 --symver_script=filename* on page C1-493.
- *C1.143 --symver_soname* on page C1-494.
- *C1.144 --sysv* on page C1-495.
- *C1.145 --tailreorder, --no_tailreorder* on page C1-496.
- *C1.146 --tiebreaker=option* on page C1-497.
- *C1.147 --unaligned_access, --no_unaligned_access* on page C1-498.
- *C1.148 --undefined=symbol* on page C1-499.
- *C1.149 --undefined_and_export=symbol* on page C1-500.
- *C1.150 --unresolved=symbol* on page C1-501.
- *C1.151 --use_definition_visibility* on page C1-502.
- *C1.152 --userlibpath=pathlist* on page C1-503.
- *C1.153 --veneerinject, --no_veneerinject* on page C1-504.
- *C1.154 --veneer_inject_type=type* on page C1-505.
- *C1.155 --veneer_pool_size=size* on page C1-506.
- *C1.156 --veneershare, --no_veneershare* on page C1-507.
- *C1.157 --verbose* on page C1-508.
- *C1.158 --version_number (armlink)* on page C1-509.
- *C1.159 --via=filename (armlink)* on page C1-510.
- *C1.160 --vsn (armlink)* on page C1-511.
- *C1.161 --xo_base=address* on page C1-512.
- *C1.162 --xref, --no_xref* on page C1-513.
- *C1.163 --xrefdbg, --no_xrefdbg* on page C1-514.
- *C1.164 --xref{from|to}=object(section)* on page C1-515.
- *C1.165 --zi_base=address* on page C1-516.

C1.1 --any_contingency

Permits extra space in any execution regions containing `.ANY` sections for linker-generated content such as veneers and alignment padding.

Usage

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 concepts

C6.4.8 Behavior when .ANY sections overflow because of linker-generated content on page C6-627

Related references

C1.60 --info=topic[,topic,…] (armlink) on page C1-401
C1.3 --any_sort_order=order on page C1-340
C1.121 --scatter=filename on page C1-470
C7.5.2 Syntax of an input section description on page C7-672
C1.2 --any_placement=algorithm on page C1-338
C6.4 Placement of unassigned sections on page C6-619

C1.2 --any_placement=algorithm

Controls the placement of sections that are placed using the `.ANY` module selector.

Syntax

`--any_placement=`*`algorithm`*

where *`algorithm`* is 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.

Usage

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 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.

Default

The default option is `worst_fit`.

Related concepts

C6.4.5 Examples of using placement algorithms for .ANY sections on page C6-622

C6.4.6 Example of next_fit algorithm showing behavior of full regions, selectors, and priority on page C6-624

C6.4.8 Behavior when .ANY sections overflow because of linker-generated content on page C6-627

Related references

C1.3 --any_sort_order=order on page C1-340

C1.60 --info=topic[,topic,…] (armlink) on page C1-401

C1.121 --scatter=filename on page C1-470

C1.1 --any_contingency on page C1-337

C6.4 Placement of unassigned sections on page C6-619

C7.5.2 Syntax of an input section description on page C7-672

C1.3 --any_sort_order=order

Controls the sort order of input sections that are placed using the `.ANY` module selector.

Syntax

`--any_sort_order=order`

where *`order`* is 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.

Usage

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.

Default

The default option is `--any_sort_order=descending_size`.

Related concepts

C6.4.7 Examples of using sorting algorithms for .ANY sections on page C6-625

Related references

C1.60 --info=topic[,topic,…] (armlink) on page C1-401

C1.121 --scatter=filename on page C1-470

C1.1 --any_contingency on page C1-337

C6.4 Placement of unassigned sections on page C6-619

C1.4 --api, --no_api

Enables and disables API section sorting. API sections are the sections that are called the most within a region.

Usage

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.

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.

32MB

Execution region contains only A32 instructions.

16MB

Execution region contains 32-bit T32 instructions.

4MB

Execution region contains only 16-bit T32 instructions.

Related concepts

C3.6 Linker-generated veneers on page C3-548

Related references

C1.69 --largeregions, --no_largeregions on page C1-413

C1.5 --autoat, --no_autoat

Controls the automatic assignment of `__at` sections to execution regions.

`__at` sections are sections that must be placed at a specific address.

Usage

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.

Default

The default is `--autoat`.

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.

Related tasks

C6.2.5 Placing __at sections at a specific address on page C6-611

C6.2.7 Automatically placing __at sections on page C6-612

C6.2.8 Manually placing __at sections on page C6-614

Related references

C7.2 Syntax of a scatter file on page C7-657

C1.6 --bare_metal_pie

Specifies the bare-metal *Position Independent Executable* (PIE) linking model.

Note

Not supported for AArch64 state.

Note

The Bare-metal PIE feature is deprecated.

Default

The following default settings are automatically specified:

- `--fpic`.
- `--pie`.
- `--ref_pre_init`.

Related references

C1.51 --fpic on page C1-392
C1.102 --pie on page C1-450
C1.111 --ref_pre_init, --no_ref_pre_init on page C1-460

C1.7 --base_platform

Specifies the Base Platform linking model. It is a superset of the *Base Platform Application Binary Interface* (BPABI) model, `--bpabi` option.

Note

Not supported for AArch64 state.

Usage

When you specify `--base_platform`, the linker also acts as if you specified `--bpabi` with the following exceptions:

- The full choice of memory models is available, including scatter-loading:
  - `--dll`.
  - `--force_so_throw`, `--no_force_so_throw`.
  - `--pltgot=`*`type`*.
  - `--rosplit`.

  Note

  If you do not specify a scatter file with `--scatter`, then the standard BPABI memory model scatter file is used.

- The default value of the `--pltgot` option is different to that for `--bpabi`:
  - For `--base_platform`, the default is `--pltgot=none`.
  - For `--bpabi` the default is `--pltgot=direct`.
- If you specify `--pltgot_opts=crosslr` then calls to and from a load region marked `RELOC` go by way of the *Procedure Linkage Table* (PLT).

To place unresolved weak references in the dynamic symbol table, use the `IMPORT` steering file command.

Note

If you are linking with `--base_platform`, and the parent load region has the `RELOC` attribute, then all execution regions within that load region must have a `+`*`offset`* base address.

Related concepts

C9.2 Scatter files for the Base Platform linking model on page C9-708
C2.4 Base Platform Application Binary Interface (BPABI) linking model overview on page C2-521
C2.5 Base Platform linking model overview on page C2-522
C7.3.5 Inheritance rules for the RELOC address attribute on page C7-662

Related references

C1.11 --bpabi on page C1-349
C1.105 --pltgot=type on page C1-454
C1.106 --pltgot_opts=mode on page C1-455
C1.121 --scatter=filename on page C1-470
C1.114 --remove, --no_remove on page C1-463
C1.32 --dll on page C1-373
C1.50 --force_so_throw, --no_force_so_throw on page C1-391
C1.115 --ro_base=address on page C1-464
C1.117 --rosplit on page C1-466
C1.118 --rw_base=address on page C1-467

C1.119 --rwpi on page C1-468

C1.8 --bestdebug, --no_bestdebug

Selects between linking for smallest code and data size or for best debug illusion.

Usage

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.

Default

The default is `--no_bestdebug`. The smallest COMDAT groups are selected when linking, at the expense of a possibly slightly poorer debug illusion.

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
```

Related concepts

C4.1 Elimination of common section groups on page C4-562

C4.2 Elimination of unused sections on page C4-563

Related references

C1.93 -o filename, --output=filename (armlink) on page C1-441

C1.9 --blx_arm_thumb, --no_blx_arm_thumb

Enables the linker to use the `BLX` instruction for A32 to T32 state changes.

Usage

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 concepts

C3.6.3 Veneer types on page C3-549

Related references

C1.10 --blx_thumb_arm, --no_blx_thumb_arm on page C1-348

C1.10 --blx_thumb_arm, --no_blx_thumb_arm

Enables the linker to use the `BLX` instruction for T32 to A32 state changes.

Usage

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.

Related concepts

C3.6.3 Veneer types on page C3-549

Related references

C1.9 --blx_arm_thumb, --no_blx_arm_thumb on page C1-347

C1.11 --bpabi

Creates a *Base Platform Application Binary Interface* (BPABI) ELF file for passing to a platform-specific post-linker.

Note

Not supported for AArch64 state.

Usage

The BPABI model defines a standard-memory model that enables interoperability of BPABI-compliant files across toolchains. When you specify this option:

- *Procedure Linkage Table* (PLT) and *Global Offset Table* (GOT) generation is supported.
- The default value of the `--pltgot` option is `direct`.
- A *dynamically linked library* (DLL) placed on the command-line can define symbols.

Restrictions

The BPAPI model does not support scatter-loading. However, scatter-loading is supported in the Base Platform model.

Weak references in the dynamic symbol table are permitted only if the symbol is defined by a DLL placed on the command-line. You cannot place an unresolved weak reference in the dynamic symbol table with the `IMPORT` steering file command.

Related concepts

C2.4 Base Platform Application Binary Interface (BPABI) linking model overview on page C2-521

C2.5 Base Platform linking model overview on page C2-522

Related references

C1.7 --base_platform on page C1-344

C1.114 --remove, --no_remove on page C1-463

C1.32 --dll on page C1-373

C1.105 --pltgot=type on page C1-454

C1.123 --shared on page C1-473

C1.144 --sysv on page C1-495

Chapter C8 BPABI and SysV Shared Libraries and Executables on page C8-685

C1.12 --branchnop, --no_branchnop

Enables or disables the replacement of any branch with a relocation that resolves to the next instruction with a `NOP`.

Usage

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.

Default

The default is `--branchnop`.

Related concepts

C4.6 About branches that optimize to a NOP on page C4-570

Related references

C1.63 --inline, --no_inline on page C1-406

C1.145 --tailreorder, --no_tailreorder on page C1-496

C1.13 --callgraph, --no_callgraph

Creates a file containing a static callgraph of functions.

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.

Usage

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.

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.

Default

The default is `--no_callgraph`.

Related references

C1.14 --callgraph_file=filename on page C1-353
C1.15 --callgraph_output=fmt on page C1-354
C1.16 --callgraph_subset=symbol[,symbol,...] on page C1-355
C1.17 --cgfile=type on page C1-356
C1.18 --cgsymbol=type on page C1-357
C1.19 --cgundefined=type on page C1-358
C7.2 Syntax of a scatter file on page C7-657

C1.14 --callgraph_file=filename

Controls the output filename of the callgraph.

Syntax

`--callgraph_file=filename`

where *filename* is the callgraph filename.

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`.

Related references

C1.13 --callgraph, --no_callgraph on page C1-351
C1.15 --callgraph_output=fmt on page C1-354
C1.16 --callgraph_subset=symbol[,symbol,...] on page C1-355
C1.17 --cgfile=type on page C1-356
C1.18 --cgsymbol=type on page C1-357
C1.19 --cgundefined=type on page C1-358
C1.93 -o filename, --output=filename (armlink) on page C1-441

C1.15 --callgraph_output=fmt

Controls the output format of the callgraph.

Syntax

`--callgraph_output=`*`fmt`*

Where *`fmt`* can be one of the following:

`html`

Outputs the callgraph in HTML format.

`text`

Outputs the callgraph in plain text format.

Default

The default is `--callgraph_output=html`.

Related references

C1.13 --callgraph, --no_callgraph on page C1-351
C1.14 --callgraph_file=filename on page C1-353
C1.16 --callgraph_subset=symbol[,symbol,...] on page C1-355
C1.17 --cgfile=type on page C1-356
C1.18 --cgsymbol=type on page C1-357
C1.19 --cgundefined=type on page C1-358

C1.16 --callgraph_subset=symbol[,symbol,...]

Creates a file containing a static callgraph for one or more specified symbols.

Syntax

`--callgraph_subset=symbol[,symbol,…]`

where `symbol` is a comma-separated list of symbols.

Usage

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 references

C1.13 --callgraph, --no_callgraph on page C1-351
C1.14 --callgraph_file=filename on page C1-353
C1.15 --callgraph_output=fmt on page C1-354
C1.17 --cgfile=type on page C1-356
C1.18 --cgsymbol=type on page C1-357
C1.19 --cgundefined=type on page C1-358

C1.17 --cgfile=type

Controls the type of files to use for obtaining the symbols to be included in the callgraph.

Syntax

`--cgfile=`*`type`*

where *`type`* can be 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.

Default

The default is `--cgfile=all`.

Related references

C1.13 --callgraph, --no_callgraph on page C1-351
C1.14 --callgraph_file=filename on page C1-353
C1.15 --callgraph_output=fmt on page C1-354
C1.16 --callgraph_subset=symbol[,symbol,...] on page C1-355
C1.18 --cgsymbol=type on page C1-357
C1.19 --cgundefined=type on page C1-358

C1.18 --cgsymbol=type

Controls what symbols are included in the callgraph.

Syntax

`--cgsymbol=`*`type`*

Where *`type`* can be one of the following:

`all`

Includes both local and global symbols.

`locals`

Includes only local symbols.

`globals`

Includes only global symbols.

Default

The default is `--cgsymbol=all`.

Related references

C1.13 --callgraph, --no_callgraph on page C1-351
C1.14 --callgraph_file=filename on page C1-353
C1.15 --callgraph_output=fmt on page C1-354
C1.16 --callgraph_subset=symbol[,symbol,...] on page C1-355
C1.17 --cgfile=type on page C1-356
C1.19 --cgundefined=type on page C1-358

C1.19 --cgundefined=type

Controls what undefined references are included in the callgraph.

Syntax

`--cgundefined=type`

Where *`type`* can be 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.

Default

The default is `--cgundefined=all`.

Related references

C1.13 --callgraph, --no_callgraph on page C1-351
C1.14 --callgraph_file=filename on page C1-353
C1.15 --callgraph_output=fmt on page C1-354
C1.16 --callgraph_subset=symbol[,symbol,...] on page C1-355
C1.17 --cgfile=type on page C1-356
C1.18 --cgsymbol=type on page C1-357

C1.20 --comment_section, --no_comment_section

Controls the inclusion of a comment section `.comment` in the final image.

Usage

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.

Default

The default is `--comment_section`.

Related concepts

C4.9 Linker merging of comment sections on page C4-573

Related references

C1.46 --filtercomment, --no_filtercomment on page C1-387

C1.21 --cppinit, --no_cppinit

Enables the linker to use alternative C++ libraries with a different initialization symbol if required.

Syntax

`--cppinit=`*`symbol`*

Where *`symbol`* is the initialization symbol to use.

Usage

If you do not specify `--cppinit=`*`symbol`* then the default symbol `__cpp_initialize__aeabi_` is assumed.

`--no_cppinit` does not take a *`symbol`* argument.

Effect

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` means do not add a reference.

Related references

C1.110 --ref_cpp_init, --no_ref_cpp_init on page C1-459

C1.22 --cpu=list (armlink)

Lists the architecture and processor names that are supported by the `--cpu=name` option.

Syntax

`--cpu=list`

Related references

C1.23 --cpu=name (armlink) on page C1-362
C1.52 --fpu=list (armlink) on page C1-393
C1.53 --fpu=name (armlink) on page C1-394

C1.23 --cpu=name (armlink)

Enables code generation for the selected Arm processor or architecture.

If you do not include the `--cpu` option, `armlink` derives an architecture from the combination of the input objects.

If you include `--cpu=`*`name`*, `armlink`:
- Faults any input object that is not compatible with the cpu.
- For library selection, acts as if at least one input object is compiled with `--cpu=`*`name`*.

Note

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.

Syntax

`--cpu=`*`name`*

Where *`name`* is 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 C1-1 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 Security Extensions 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.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.

Table C1-1 Supported Arm architectures (continued)

Architecture name	Description
`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.
`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 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.

Usage

If you omit `--cpu`, the linker auto-detects the processor or architecture from the input object files.

Specify `--cpu=list` to list the supported processor and architecture names that you can use with `--cpu=name`.

The link phase fails if any of the component object files rely on features that are incompatible with the specified processor. 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.

Restrictions

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

Related references

C1.22 --cpu=list (armlink) on page C1-361
C1.52 --fpu=list (armlink) on page C1-393
C1.53 --fpu=name (armlink) on page C1-394

C1.24 --crosser_veneershare, --no_crosser_veneershare

Enables or disables veneer sharing across execution regions.

Usage

The default is `--crosser_veneershare`, and enables veneer sharing across execution regions.

`--no_crosser_veneershare` prohibits veneer sharing across execution regions.

Related references

C1.156 --veneershare, --no_veneershare on page C1-507

C1.25 --datacompressor=opt

Enables you to specify one of the supplied algorithms for RW data compression.

Note

Not supported for AArch64 state.

Syntax

`--datacompressor=opt`

Where *`opt`* is 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 C1-2 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.

Usage

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.

Default

The default is `--datacompressor=on`.

Related concepts

C4.3.3 How compression is applied on page C4-565
C4.3.4 Considerations when working with RW data compression on page C4-565
C4.3.1 How the linker chooses a compressor on page C4-564

C1.26 --debug, --no_debug

Controls the generation of debug information in the output file.

Usage

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 --expandarrays` and `--fieldoffsets` options on the image. The `fromelf --expandarrays` and `--fieldoffsets` functionality requires that the object or image file has debug information.

Default

The default is `--debug`.

Related references

D1.26 --fieldoffsets on page D1-757

C1.27 --diag_error=tag[,tag,…] (armlink)

Sets diagnostic messages that have a specific tag to Error severity.

Syntax

`--diag_error=tag[,tag,…]`

Where `tag` can be:

- A diagnostic message number to set to error severity. This is the four-digit number, *nnnn*, with the tool letter prefix, but without the letter suffix indicating the severity.
- `warning`, to treat all warnings as errors.

Related references

C1.28 --diag_remark=tag[,tag,...] (armlink) on page C1-369

C1.29 --diag_style={arm|ide|gnu} (armlink) on page C1-370

C1.30 --diag_suppress=tag[,tag,...] (armlink) on page C1-371

C1.31 --diag_warning=tag[,tag,...] (armlink) on page C1-372

C1.133 --strict on page C1-484

C1.28 --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,…]`

Where *`tag`* is a comma-separated list of diagnostic message numbers. This is the four-digit number, *`nnnn`*, with the tool letter prefix, but without the letter suffix indicating the severity.

Related references

C1.27 --diag_error=tag[,tag,…] (armlink) on page C1-368
C1.29 --diag_style={arm|ide|gnu} (armlink) on page C1-370
C1.30 --diag_suppress=tag[,tag,…] (armlink) on page C1-371
C1.31 --diag_warning=tag[,tag,…] (armlink) on page C1-372
C1.113 --remarks on page C1-462
C1.133 --strict on page C1-484

C1.29 --diag_style={arm|ide|gnu} (armlink)

Specifies the display style for diagnostic messages.

Syntax

`--diag_style=string`

Where `string` is 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`.

Default

The default is `--diag_style=arm`.

Usage

`--diag_style=gnu` matches the format reported by the GNU Compiler, `gcc`.

`--diag_style=ide` matches the format reported by Microsoft Visual Studio.

Related references

C1.27 --diag_error=tag[,tag,…] (armlink) on page C1-368
C1.28 --diag_remark=tag[,tag,…] (armlink) on page C1-369
C1.30 --diag_suppress=tag[,tag,…] (armlink) on page C1-371
C1.31 --diag_warning=tag[,tag,…] (armlink) on page C1-372
C1.113 --remarks on page C1-462
C1.133 --strict on page C1-484

C1.30 --diag_suppress=tag[,tag,…] (armlink)

Suppresses diagnostic messages that have a specific tag.

Syntax

`--diag_suppress=`*`tag[,tag,…]`*
Where *`tag`* can be:

- A diagnostic message number to be suppressed. This is the four-digit number, *nnnn*, with the tool letter prefix, but without the letter suffix indicating the severity.
- `error`, to suppress all errors that can be downgraded.
- `warning`, to suppress all warnings.

Example

To suppress the warning messages that have numbers `L6314W` and `L6305W`, use the following command:

```
armlink --diag_suppress=L6314,L6305 …
```

Related references

C1.27 --diag_error=tag[,tag,…] (armlink) on page C1-368
C1.28 --diag_remark=tag[,tag,…] (armlink) on page C1-369
C1.29 --diag_style={arm|ide|gnu} (armlink) on page C1-370
C1.31 --diag_warning=tag[,tag,…] (armlink) on page C1-372
C1.133 --strict on page C1-484
C1.113 --remarks on page C1-462

C1.31 --diag_warning=tag[,tag,…] (armlink)

Sets diagnostic messages that have a specific tag to Warning severity.

Syntax

`--diag_warning=tag[,tag,…]`

Where `tag` can be:

- A diagnostic message number to set to warning severity. This is the four-digit number, *nnnn*, with the tool letter prefix, but without the letter suffix indicating the severity.
- `error`, to set all errors that can be downgraded to warnings.

Related references

C1.27 --diag_error=tag[,tag,...] (armlink) on page C1-368
C1.28 --diag_remark=tag[,tag,...] (armlink) on page C1-369
C1.29 --diag_style={arm|ide|gnu} (armlink) on page C1-370
C1.30 --diag_suppress=tag[,tag,...] (armlink) on page C1-371
C1.113 --remarks on page C1-462

C1.32 --dll

Creates a *Base Platform Application Binary Interface* (BPABI) *dynamically linked library* (DLL).

Note

Not supported for AArch64 state.

Usage

The DLL is marked as a shared object in the ELF file header.

You must use `--bpabi` with `--dll` to produce a BPABI-compliant DLL.

You can also use `--dll` with `--base_platform`.

Note

By default, this option disables unused section elimination. Use the `--remove` option to re-enable unused sections when building a DLL.

Related references

C1.114 --remove, --no_remove on page C1-463
C1.11 --bpabi on page C1-349
C1.123 --shared on page C1-473
C1.144 --sysv on page C1-495
Chapter C8 BPABI and SysV Shared Libraries and Executables on page C8-685

C1.33 --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 references

C1.47 --fini=symbol on page C1-388
C1.62 --init=symbol on page C1-405
C1.73 --library=name on page C1-418
Chapter C8 BPABI and SysV Shared Libraries and Executables on page C8-685

C1.34 --eager_load_debug, --no_eager_load_debug

Manages how `armlink` loads debug section data.

Usage

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.

The default is `--eager_load_debug`.

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 references

C1.20 --comment_section, --no_comment_section on page C1-359

C1.35 --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.

Usage

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.

Default

The default is `--eh_frame_hdr`.

Restrictions

Valid only for AArch64 images.

Related information
Linux Foundation

C1.36 --edit=file_list

Enables you to specify steering files containing commands to edit the symbol tables in the output binary.

Syntax

`--edit=`*`file_list`*

Where *`file_list`* can be more than one steering file separated by a comma. Do not include a space after the comma.

Usage

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
```

Related concepts

C5.6.4 Hide and rename global symbols with a steering file on page C5-592

Related references

C5.6.2 Steering file command summary on page C5-590

Chapter C10 Linker Steering File Command Reference on page C10-711

C1.37 --emit_debug_overlay_relocs

Outputs only relocations of debug sections with respect to overlaid program sections to aid an overlay-aware debugger.

Note

Not supported for AArch64 state.

Related references

C1.38 --emit_debug_overlay_section on page C1-379
C1.40 --emit_relocs on page C1-381
C1.39 --emit_non_debug_relocs on page C1-380

Related information

Manual overlay support
ABI for the Arm Architecture: Support for Debugging Overlaid Programs

C1.38 --emit_debug_overlay_section

Emits a special debug overlay section during static linking.

Note

Not supported for AArch64 state.

Usage

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 references

C1.37 --emit_debug_overlay_relocs on page C1-378

C1.40 --emit_relocs on page C1-381

Related information

Automatic overlay support

Manual overlay support

ABI for the Arm Architecture: Support for Debugging Overlaid Programs

C1.39 --emit_non_debug_relocs

Retains only relocations from non-debug sections in an executable file.

Note

Not supported for AArch64 state.

Related references

C1.40 --emit_relocs on page C1-381

C1.40 --emit_relocs

Retains all relocations in the executable file. This results in larger executable files.

——— **Note** ———

Not supported for AArch64 state.

———————————

Usage

This is equivalent to the GNU ld `--emit-relocs` option.

Related references

C1.37 --emit_debug_overlay_relocs on page C1-378

C1.39 --emit_non_debug_relocs on page C1-380

Related information

ABI for the Arm Architecture: Support for Debugging Overlaid Programs

C1.41 --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`*
Where *`location`* is one of the following:

`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 within 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. The linker does this automatically if you specify a symbol. For example, if the entry code starts at address `0x8000` in T32 state you must use `--entry=0x8001`.

Usage

The image can contain multiple entry points. Multiple entry points might be specified with the `ENTRY` directive in assembler source files. 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.
- The execution region must be non-overlay, and must be a root execution region (load address == execution address).

Related references

C1.131 --startup=symbol, --no_startup on page C1-482

F6.26 ENTRY on page F6-1049

C1.42 --errors=filename

Redirects the diagnostics from the standard error stream to a specified file.

Syntax

`--errors=`*`filename`*

Usage

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.

Related references

C1.27 --diag_error=tag[,tag,…] (armlink) on page C1-368
C1.28 --diag_remark=tag[,tag,…] (armlink) on page C1-369
C1.29 --diag_style={arm|ide|gnu} (armlink) on page C1-370
C1.30 --diag_suppress=tag[,tag,…] (armlink) on page C1-371
C1.31 --diag_warning=tag[,tag,…] (armlink) on page C1-372
C1.113 --remarks on page C1-462

C1.43 --exceptions, --no_exceptions

Controls the generation of exception tables in the final image.

Usage

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.

Default

The default is `--exceptions`.

C1.44 --export_all, --no_export_all

Controls the export of all global, non-hidden symbols to the dynamic symbols table.

Usage

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.

`--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.

Default

The default is `--export_all` for building shared libraries and *dynamically linked libraries* (DLLs).

The default is `--no_export_all` for building applications.

Related references

C1.45 --export_dynamic, --no_export_dynamic on page C1-386

C1.45 --export_dynamic, --no_export_dynamic

Controls the export of dynamic symbols to the dynamic symbols table.

Note

Not supported for AArch64 state.

Usage

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.

Default

`--no_export_dynamic` is the default.

Related references

C1.44 --export_all, --no_export_all on page C1-385

C1.46 --filtercomment, --no_filtercomment

Controls whether or not the linker modifies the `.comment` section to assist merging.

Usage

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.

Note

`armlink` does not preprocess comment sections from `armclang`.

Default

The default is `--filtercomment`.

Related concepts

C4.9 Linker merging of comment sections on page C4-573

Related references

C1.20 --comment_section, --no_comment_section on page C1-359

C1.47 --fini=symbol

Specifies the symbol name to use to define the entry point for finalization code.

Syntax

`--fini=symbol`

Where `symbol` is the symbol name to use for the entry point to the finalization code.

Usage

The dynamic linker executes this code when it unloads the executable file or shared object.

Related references

C1.33 --dynamic_linker=name on page C1-374
C1.62 --init=symbol on page C1-405
C1.73 --library=name on page C1-418

C1.48 --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`*

Where *`section_id`* is 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.

Usage

The `--first` option cannot be used with `--scatter`. Instead, use the `+FIRST` attribute in a scatter file.

Related concepts

C3.3.2 Section placement with the FIRST and LAST attributes on page C3-544

C3.3 Section placement with the linker on page C3-543

Related references

C1.70 --last=section_id on page C1-415

C1.121 --scatter=filename on page C1-470

C1.49 --force_explicit_attr

Causes the linker to retry the CPU mapping using build attributes constructed when an architecture is specified with `--cpu`.

Usage

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 references

C1.23 --cpu=name (armlink) on page C1-362

C1.53 --fpu=name (armlink) on page C1-394

C1.50 --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`.

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.

C1.51 --fpic

Enables you to link *Position-Independent Code* (PIC), that is, code that has been 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.

Note

The Bare-metal PIE feature is deprecated.

Related concepts

Linker options for SysV models

Related references

C1.123 --shared on page C1-473

C1.144 --sysv on page C1-495

C1.6 --bare_metal_pie on page C1-343

C1.52 --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
```

Related references

C1.22 --cpu=list (armlink) on page C1-361
C1.23 --cpu=name (armlink) on page C1-362
C1.53 --fpu=name (armlink) on page C1-394

C1.53 --fpu=name (armlink)

Specifies the target FPU architecture.

Syntax

`--fpu=`*`name`*

Where *`name`* is 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.

Note

Software floating-point linkage is not supported for AArch64 state.

Usage

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.

Restrictions

Arm Neon support is disabled for `SoftVFP`.

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.

Related references

C1.22 --cpu=list (armlink) on page C1-361
C1.23 --cpu=name (armlink) on page C1-362
C1.52 --fpu=list (armlink) on page C1-393

C1.54 --got=type

Generates *Global Offset Tables* (GOTs) to resolve GOT relocations in bare metal images. `armlink` statically resolves the GOT relocations.

Syntax

`--got=`*`type`*

Where *`type`* is one of the following:

`none`

Disables GOT generation.

`local`

Creates a local offset table for each execution region.

Note

Not supported for AArch32 state.

`global`

Creates a single offset table for the whole image.

Default

The default for AArch32 state is `none`.

The default for AArch64 state is `local`.

C1.55 --gnu_linker_defined_syms

Enables support for the GNU equivalent of input section symbols.

Note

The `--gnu_linker_defined_syms` linker option is deprecated.

Usage

If you want GNU-style behavior when treating the Arm symbols `SectionName$$Base` and `SectionName$$Limit`, then specify `--gnu_linker_defined_syms`.

Table C1-3 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.

C1.56 --help (armlink)

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.

Related references

C1.158 --version_number (armlink) on page C1-509
C1.160 --vsn (armlink) on page C1-511
C1.124 --show_cmdline (armlink) on page C1-474

C1.57 --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`*

Where *`filename`* is the name of the import library file.

Usage

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 concepts

C3.6.6 Generation of secure gateway veneers on page C3-551

Related references

C1.58 --import_cmse_lib_out=filename on page C1-399

Related information

Building Secure and Non-secure Images Using Armv8-M Security Extensions

C1.58 --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`

Where *filename* is 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.

Related concepts

C3.6.6 Generation of secure gateway veneers on page C3-551

Related references

C1.57 --import_cmse_lib_in=filename on page C1-398

Related information

Building Secure and Non-secure Images Using Armv8-M Security Extensions

C1.59 --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
```

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 references

C1.123 --shared on page C1-473
C1.144 --sysv on page C1-495

C1.60 --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,…]`*

Where *`topic`* is 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.

`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.

`merge`

Lists the **`const`** strings that the linker merges. Each item lists the merged result, the strings being merged, and the associated object files.

`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. This option requires debug information generated by the compiler using the -g option.

`summarysizes`

Summarizes the code and data sizes of the image.

`summarystack`
Summarizes the stack usage of all global symbols. This option requires debug information generated by the compiler using the -g option.

`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.

`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.

Usage

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 concepts

C4.2 Elimination of unused sections on page C4-563
C4.3.4 Considerations when working with RW data compression on page C4-565
C4.3 Optimization with RW data compression on page C4-564
C4.3.1 How the linker chooses a compressor on page C4-564
C4.3.3 How compression is applied on page C4-565

Related references

C1.1 --any_contingency on page C1-337
C1.3 --any_sort_order=order on page C1-340
C1.13 --callgraph, --no_callgraph on page C1-351
C1.61 --info_lib_prefix=opt on page C1-404

C1.90 --merge, --no_merge on page C1-438
C1.154 --veneer_inject_type=type on page C1-505
C6.4 Placement of unassigned sections on page C6-619
C1.25 --datacompressor=opt on page C1-366
C1.63 --inline, --no_inline on page C1-406
C1.114 --remove, --no_remove on page C1-463
C1.68 --keep_intermediate on page C1-412
C1.145 --tailreorder, --no_tailreorder on page C1-496
C7.4.3 Execution region attributes on page C7-666

Related information

Options for getting information about linker-generated files

C1.61 --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`*

Where *`opt`* is the prefix of the required library.

Examples

- Displaying a list of libraries without the filter:

  ```
  armlink --info=libraries test.o
  ```

  Produces a list of libraries, for example:

  ```
  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:

  ```
  armlink --info=libraries --info_lib_prefix=c test.o
  ```

  Produces a list of libraries with the specified prefix, for example:

  ```
  install_directory\lib\armlib\c_4.l
  ```

Related references

C1.60 --info=topic[,topic,…] (armlink) on page C1-401

C1.62 --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`

Where `symbol` is the symbol name you want to use to define the location of the initialization code.

Related references

C1.33 --dynamic_linker=name on page C1-374
C1.47 --fini=symbol on page C1-388
C1.73 --library=name on page C1-418

C1.63 --inline, --no_inline

Enables or disables branch inlining to optimize small function calls in your image.

Note

Not supported for AArch64 state.

Default

The default is `--no_inline`.

Note

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.

Related concepts

C4.4 Function inlining with the linker on page C4-567

Related references

C1.12 --branchnop, --no_branchnop on page C1-350

C1.64 --inline_type=type on page C1-407

C1.145 --tailreorder, --no_tailreorder on page C1-496

C1.64 --inline_type=type

Inlines functions from all objects, Arm standard libraries only, or turns off inlining completely.

Syntax

`--inline_type=`*`type`*

Where *`type`* is 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.

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 references

C1.63 --inline, --no_inline on page C1-406

C1.145 --tailreorder, --no_tailreorder on page C1-496

C1.65 --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`.

Related concepts

C3.6.3 Veneer types on page C3-549
C3.6 Linker-generated veneers on page C3-548
C3.6.2 Veneer sharing on page C3-548
C3.6.4 Generation of position independent to absolute veneers on page C3-550
C3.6.5 Reuse of veneers when scatter-loading on page C3-550

Related references

C1.103 --piveneer, --no_piveneer on page C1-451
C1.156 --veneershare, --no_veneershare on page C1-507

C1.66 input-file-list (armlink)

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

Usage

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.

- 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 concepts

C5.5 Access symbols in another image on page C5-587

C3.9 How the linker performs library searching, selection, and scanning on page C3-555

Related references

C1.120 --scanlib, --no_scanlib on page C1-469

C1.132 --stdlib on page C1-483

C1.67 --keep=section_id (armlink)

Specifies input sections that must not be removed by unused section elimination.

Syntax

`--keep=`*`section_id`*

Where *`section_id`* is one of the following:

`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`.

Usage

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.

———— **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_id`* 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()`.

Related concepts

C3.9 How the linker performs library searching, selection, and scanning on page C3-555

C3.1 The structure of an Arm® ELF image on page C3-528

C1.68 --keep_intermediate

Specifies whether the linker preserves the ELF intermediate object file produced by the link time optimizer.

Syntax

`--keep_intermediate=`*`option`*

Where *`option`* is:

`lto`

Preserve an intermediate ELF 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.

Related references

C1.80 --lto, --no_lto on page C1-426

Related information

Optimizing across modules with link time optimization

C1.69 --largeregions, --no_largeregions

Controls the sorting order of sections in large execution regions to minimize the distance between sections that call each other.

Usage

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`.
- Veneer injection, `--veneerinject`.

The following command lines are equivalent:

```
armlink --largeregions --no_api --no_veneerinject --sort=Lexical
armlink --no_largeregions
```

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.

Related concepts

C3.6 Linker-generated veneers on page C3-548

C3.6.2 Veneer sharing on page C3-548

C3.6.3 Veneer types on page C3-549

C3.6.4 Generation of position independent to absolute veneers on page C3-550

Related references

C1.4 --api, --no_api on page C1-341

C1.129 --sort=algorithm on page C1-479

C1.154 --veneer_inject_type=type on page C1-505

C1.153 --veneerinject, --no_veneerinject on page C1-504

C1.70 --last=section_id

Places the selected input section last in its execution region.

Syntax

`--last=`*`section_id`*

Where *`section_id`* is 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.

Usage

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 concepts

C3.3.2 Section placement with the FIRST and LAST attributes on page C3-544

C3.3 Section placement with the linker on page C3-543

Related references

C1.48 --first=section_id on page C1-389

C1.121 --scatter=filename on page C1-470

C1.71 --legacyalign, --no_legacyalign

Controls how padding is inserted into the image.

Note

The `--legacyalign` and `--no_legacyalign` linker options are deprecated.

Usage

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.

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.

Default

The default is `--no_legacyalign`,

Related concepts

C3.3 Section placement with the linker on page C3-543

C6.12 Example of using expression evaluation in a scatter file to avoid padding on page C6-642

Related references

C7.3.3 Load region attributes on page C7-659

C7.4.3 Execution region attributes on page C7-666

C1.72 --libpath=pathlist

Specifies a list of paths that the linker uses to search for the Arm standard C and C++ libraries.

Syntax

`--libpath=`*`pathlist`*

Where *`pathlist`* is 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`*.

Note

This option does not affect searches for user libraries. Use `--userlibpath` instead for user libraries.

Related concepts

C3.9 How the linker performs library searching, selection, and scanning on page C3-555

Related references

C1.152 --userlibpath=pathlist on page C1-503

Related information

Toolchain environment variables

C1.73 --library=name

Enables the linker to search a static library without you having specifying the full library filename on the command-line.

———— **Note** ————

Not supported in the Keil® Microcontroller Development Kit (Keil MDK).

————————————

Syntax

`--library=`*`name`*

Links with the static library, `lib`*`name`*`.a`.

Usage

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 references

C1.51 --fpic on page C1-392

C1.123 --shared on page C1-473

C1.74 --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* on page A1-39.

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 and Pointer Authentication on function returns.

v8.5a [ALPHA]

Selects the `v8.5a` library, which provides memory tagging protection of the stack 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 Branch Target Identification and Pointer Authentication, and does not provide memory tagging stack protection.

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 `-mmemtag-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.

Usage

Use `--library_security` to override the automatic selection of protected libraries for branch protection and memory tagging stack protection (stack tagging).

Branch protection protects your code from Return Oriented Programming (ROP) and Jump Oriented Programming (JOP) attacks. Branch protection using pointer authentication and branch target identification are only available in AArch64 state.

Memory tagging stack protection protects accesses to variables on the stack whose addresses are taken. Memory tagging protection is available for the AArch64 state for architectures with the memory tagging extension.

Note

- Selecting the `v8.5a` library does not automatically imply memory tagging protection of the heap. To enable memory tagging protection of the heap, you must define the 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 stack tagging can be safely linked together with code that is compiled without stack tagging. However, if any object file is compiled with `-fsanitize=memtag`, and if `setjmp`, `longjmp`, or C++ exceptions are present anywhere in the image, then you must use the `v8.5a` library to avoid stack tagging related memory fault at runtime.

Examples

This uses the `v8.3a` library with branch protection using Branch Target Identification and Pointer Authentication:

```
armlink --cpu=8.3-A.64 --library_security=v8.3a foo.o
```

This uses the standard C library without any branch protection using Branch Target Identification and Pointer Authentication:

```
armlink --cpu=8.3-A.64 --library_security=none foo.o
```

This uses the `v8.5a` library with memory tagging stack protection, and branch protection using Branch Target Identification and Pointer Authentication:

```
armlink --library_security=v8.5a foo.o
```

Related references

C1.75 --library_type=lib on page C1-421

B1.53 -mbranch-protection on page B1-119

C1.75 --library_type=lib

Selects the library to be used at link time.

Syntax

`--library_type=lib`

Where *`lib`* can be one of:

`standardlib`

Specifies that the full Arm Compiler runtime libraries are selected at link time. This is the default.

`microlib`

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

Note

`microlib` is not supported for AArch64 state.

Usage

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`.

Related information

Building an application with microlib

C1.76 --list=filename

Redirects diagnostic output to a file.

Syntax

`--list=`*`filename`*

Where *`filename`* is the file to use to save the diagnostic output. *`filename`* can include a path

Usage

Redirects the diagnostics output by the `--info`, `--map`, `--symbols`, `--verbose`, `--xref`, `--xreffrom`, and `--xrefto` options to *`file`*.

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 references

C1.86 --map, --no_map on page C1-434
C1.157 --verbose on page C1-508
C1.162 --xref, --no_xref on page C1-513
C1.163 --xrefdbg, --no_xrefdbg on page C1-514
C1.164 --xref{from|to}=object(section) on page C1-515
C1.60 --info=topic[,topic,...] (armlink) on page C1-401
C1.140 --symbols, --no_symbols on page C1-491

C1.77 --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 (Armv7-A), data, and A64 code.

Default

The default is `--no_list_mapping_symbols`.

Related concepts

C5.1 About mapping symbols on page C5-578

Related references

C1.140 --symbols, --no_symbols on page C1-491

Related information

ELF for the Arm Architecture

C1.78 --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.

Usage

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

Default

The default is `--no_load_addr_map_info`.

Restrictions

You must use `--map` with this option.

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 references

C1.86 --map, --no_map on page C1-434

C1.79 --locals, --no_locals

Adds local symbols or removes local symbols depending on whether an image or partial object is being output.

Usage

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.

Default

The default is `--locals`.

Related references

C1.109 --privacy (armlink) on page C1-458

D1.47 --privacy (fromelf) on page D1-780

D1.57 --strip=option[,option,…] on page D1-790

C1.80 --lto, --no_lto

Enables link time optimization.

Caution

Link Time Optimization performs aggressive optimizations by analyzing the dependencies between bitcode format objects. This 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 `.llvmbc` section.

With the `--no_lto` option, `armlink` gives an error message if it encounters any `.llvmbc` sections.

Default

The default is `--no_lto`.

Dependencies

Link time optimization requires the dependent library `libLTO`.

Table C1-4 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` executables and the `libLTO` library must come from the same Arm Compiler installation. Any use of `libLTO` other than that supplied with Arm Compiler is unsupported.

Note

Link Time Optimization 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 and produces a warning.

Related references

C1.60 --info=topic[,topic,…] (armlink) on page C1-401
C1.68 --keep_intermediate on page C1-412
C1.81 --lto_keep_all_symbols, --no_lto_keep_all_symbols on page C1-428
C1.82 --lto_intermediate_filename on page C1-429
C1.84 --lto_relocation_model on page C1-432
C1.83 --lto_level on page C1-430
C1.97 -Omax (armlink) on page C1-445
B1.20 -flto, -fno-lto on page B1-77

Related information

Optimizing across modules with link time optimization

C1.81 --lto_keep_all_symbols, --no_lto_keep_all_symbols

Specifies whether link time optimization removes unreferenced global symbols.

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.

Default

The default is `--no_lto_keep_all_symbols`.

Related references

C1.67 --keep=section_id (armlink) on page C1-410

C1.80 --lto, --no_lto on page C1-426

Related information

Optimizing across modules with link time optimization

C1.82 --lto_intermediate_filename

Specifies the name of the ELF object file produced by the link time optimizer.

Syntax

`--lto_intermediate_filename=`*`filename`*

Where *`filename`* is the filename the link time optimizer uses for the ELF object file it produces.

Usage

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.

Default

The default is a temporary filename.

Related references

C1.68 --keep_intermediate on page C1-412

C1.80 --lto, --no_lto on page C1-426

Related information

Optimizing across modules with link time optimization

C1.83 --lto_level

Sets the optimization level for the link time optimization feature.

Syntax

`--lto_level=O`*`level`*

Where *`level`* is 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 significantly 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.

`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 all the optimizations from level `3` including those optimizations that are performed with the `-ffp-mode=fast` `armclang` option. This level also performs other aggressive optimizations that 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.

Caution

This option is not guaranteed to be fully standards-compliant for all code cases.

Note

- Code-size, build-time, and the debug view can each be adversely affected when using this option.
- Arm 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.

Default

If you do not specify `O`*`level`*, the linker assumes `O2`.

Related references

C1.80 --lto, --no_lto on page C1-426

C1.97 -Omax (armlink) on page C1-445

B1.70 -O (armclang) on page B1-150

Related information

Optimizing across modules with link time optimization

C1.84 --lto_relocation_model

Specifies whether the link time optimizer produces absolute or position independent code.

Syntax

`--lto_relocation_model=`*`model`*

Where model is one of the following:

`static`

The link time optimizer produces absolute code.

`pic`

The link time optimizer produces code that uses GOT relative position independent code.

The `--lto_relocation_model=pic` option requires the `armlink` `--bare_metal_pie` option.

Note

The Bare-metal PIE feature is deprecated.

Default

The default is `--lto_relocation_model=static`.

Related references

C1.6 --bare_metal_pie on page C1-343

C1.80 --lto, --no_lto on page C1-426

Related information

Optimizing across modules with link time optimization

C1.85 --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.

Usage

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.

Default

The default is `--unmangled`.

Related references

C1.140 --symbols, --no_symbols on page C1-491
C1.162 --xref, --no_xref on page C1-513
C1.163 --xrefdbg, --no_xrefdbg on page C1-514
C1.164 --xref{from|to}=object(section) on page C1-515

C1.86 --map, --no_map

Enables or disables the printing of a memory map.

Usage

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`*.

Default

The default is `--no_map`.

Related references

C1.78 --load_addr_map_info, --no_load_addr_map_info on page C1-424
C1.76 --list=filename on page C1-422
C1.122 --section_index_display=type on page C1-472

C1.87 --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.

Syntax

```
--max_er_extension=size
```

Where *`size`* is the constant value in bytes to use when calculating the size of the execution region.

Default

The default size is `10240` bytes.

Related tasks

C6.2.7 Automatically placing __at sections on page C6-612

C1.88 --max_veneer_passes=value

Specifies a limit to the number of veneer generation passes the linker attempts to make when certain conditions are met.

Syntax

`--max_veneer_passes=`*`value`*

Where *`value`* is the maximum number of veneer passes the linker is to attempt. The minimum value you can specify is one.

Usage

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`.

Default

The default number of passes is 10.

Related references

C1.28 --diag_remark=tag[,tag,…] (armlink) on page C1-369

C1.31 --diag_warning=tag[,tag,…] (armlink) on page C1-372

C1.89 --max_visibility=type

Controls the visibility of all symbol definitions.

Syntax

`--max_visibility=type`

Where *type* can be one of:

`default`

Default visibility.

`protected`

Protected visibility.

Usage

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.

Default

The default is `--max_visibility=default`.

Related references

C1.96 --override_visibility on page C1-444

C1.90 --merge, --no_merge

Enables or disables the merging of **`const`** strings that are placed in shareable sections by the compiler.

Usage

Using `--merge` can reduce the size of the image if there are similarities between **`const`** strings.

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`.

Default

The default is `--merge`.

Related references

C1.60 --info=topic[,topic,...] (armlink) on page C1-401

C7.3.3 Load region attributes on page C7-659

C1.91 --merge_litpools, --no_merge_litpools

Attempts to merge identical constants in objects targeted at AArch32 state. The objects must be produced with Arm Compiler 6.

Default

`--merge_litpools` is the default.

Related tasks

C4.10 Merging identical constants on page C4-574

C1.92 --muldefweak, --no_muldefweak

Enables or disables multiple weak definitions of a symbol.

Usage

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.

Default

The default is `--muldefweak`.

C1.93 -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`

Where *`filename`* is the name of the output file, and can include a path.

Usage

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 references

C1.14 --callgraph_file=filename on page C1-353

C1.101 --partial on page C1-449

C1.94 --output_float_abi=option

Specifies the floating-point procedure call standard to advertise in the ELF header of the executable.

Note

Not supported for AArch64 state.

Syntax

`--output_float_abi=`*`option`*

where *`option`* is 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.

Usage

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.

Default

The default option is `auto`.

Related references

D1.14 --decode_build_attributes on page D1-743

D1.59 --text on page D1-793

Related information

ELF for the Arm Architecture

Run-time ABI for the Arm Architecture

C1.95 --overlay_veneers

When using the automatic overlay mechanism, causes `armlink` to redirect calls between overlays to a veneer. The veneer allows an overlay manager to unload and load the correct overlays.

Note

You must use this option if your scatter file includes execution regions with `AUTO_OVERLAY` attribute assigned to them.

Usage

`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 references

C7.4.3 Execution region attributes on page C7-666

Related information

Automatic overlay support

C1.96 --override_visibility

Enables `EXPORT` and `IMPORT` directives in a steering file to override the visibility of a symbol.

Usage

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 references

C1.149 --undefined_and_export=symbol on page C1-500

C10.1 EXPORT steering file command on page C10-712

C10.3 IMPORT steering file command on page C10-714

C1.97 -Omax (armlink)

Enables maximum link time optimization.

`-Omax` automatically enables the `--lto` and `--lto_level=Omax` options.

If you have object files that have been compiled with the `armclang -Omax` option, then you can link them using the `armlink -Omax` option to produce an image with maximum link time optimization.

Related references

C1.83 --lto_level on page C1-430
C1.80 --lto, --no_lto on page C1-426
B1.70 -O (armclang) on page B1-150

Related information

Optimizing across modules with link time optimization

C1.98 --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`

Where `num` is an integer, which can be given in hexadecimal format.

For example, setting `num` to `0xFF` might help to speed up ROM programming time. If `num` is greater than `0xFF`, then the padding byte is cast to a `char`, that is `(char)num`.

Usage

Padding is only inserted:

- 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 concepts

C3.1.2 Input sections, output sections, regions, and program segments on page C3-529

C3.1.3 Load view and execution view of an image on page C3-530

C1.99 --paged

Enables Demand Paging mode to help produce ELF files that can be demand paged efficiently.

Usage

A default page size of `0x8000` bytes is used. You can change this with the `--pagesize` command-line option.

Default

Related concepts

C3.4 Linker support for creating demand-paged files on page C3-546

Related tasks

C6.9 Aligning regions to page boundaries on page C6-638

Related references

C1.100 --pagesize=pagesize on page C1-448

C1.100 --pagesize=pagesize

Allows you to change the page size used when demand paging.

Syntax

`--pagesize=pagesize`

Where *`pagesize`* is the page size in bytes.

Default

The default value is `0x8000`.

Related concepts

C3.4 Linker support for creating demand-paged files on page C3-546

Related tasks

C6.9 Aligning regions to page boundaries on page C6-638

Related references

C1.99 --paged on page C1-447

C1.101 --partial

Creates a partially-linked object that can be used in a subsequent link step.

Restrictions

You cannot use `--partial` with `--scatter`.

Related concepts

C2.3 Partial linking model overview on page C2-520

C1.102 --pie

Species the *Position Independent Executable* (PIE) linking model.

Note

The Bare-metal PIE feature is deprecated.

Note

You must use this option with the `--fpic` and `--ref_pre_init` options.

Related references

C1.51 --fpic on page C1-392
C1.6 --bare_metal_pie on page C1-343
C1.111 --ref_pre_init, --no_ref_pre_init on page C1-460

C1.103 --piveneer, --no_piveneer

Enables or disables the generation of a veneer for a call from *position independent* (PI) code to absolute code.

Usage

When using `--no_piveneer`, an error message is produced if the linker detects a call from PI code to absolute code.

Note

Not supported for AArch64 state.

Default

The default is `--piveneer`.

Related concepts

C3.6.4 Generation of position independent to absolute veneers on page C3-550

C3.6 Linker-generated veneers on page C3-548

C3.6.2 Veneer sharing on page C3-548

C3.6.3 Veneer types on page C3-549

C3.6.5 Reuse of veneers when scatter-loading on page C3-550

Related references

C1.65 --inlineveneer, --no_inlineveneer on page C1-408

C1.156 --veneershare, --no_veneershare on page C1-507

C1.104 --pixolib

Generates a Position Independent eXecute Only (PIXO) library.

Default

`--pixolib` is disabled by default.

Syntax

`--pixolib`

Parameters

None.

Usage

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 --pixolib` command-line option. 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 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.
  - 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

Restrictions

Note

Generation of PIXO libraries is only supported for Armv7-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.

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 references

B1.64 -mpixolib on page B1-143

C1.67 --keep=section_id (armlink) on page C1-410

C1.131 --startup=symbol, --no_startup on page C1-482

Related information

The Arm C Micro-Library

C1.105 --pltgot=type

Specifies the type of *Procedure Linkage Table* (PLT) and *Global Offset Table* (GOT) to use, corresponding to the different addressing modes of the *Base Platform Application Binary Interface* (BPABI).

Note

This option is supported only when using `--base_platform` or `--bpabi`.

Note

Not supported for AArch64 state.

Syntax

`--pltgot=`*`type`*

Where *`type`* is one of the following:

`none`

References to imported symbols are added as dynamic relocations for processing by a platform specific post-linker.

`direct`

References to imported symbols are resolved to read-only pointers to the imported symbols. These are direct pointer references.

Use this type to turn on PLT generation when using `--base_platform`.

`indirect`

The linker creates a GOT and possibly a PLT entry for the imported symbol. The reference refers to PLT or GOT entry.

This type is not supported if you have multiple load regions.

`sbrel`

Same referencing as `indirect`, except that GOT entries are stored as offsets from the static base address for the segment held in R9 at runtime.

This type is not supported if you have multiple load regions.

Default

When the `--bpabi` or `--dll` options are used, the default is `--pltgot=direct`.

When the `--base_platform` option is used, the default is `--pltgot=none`.

Related concepts

C2.5 Base Platform linking model overview on page C2-522

C2.4 Base Platform Application Binary Interface (BPABI) linking model overview on page C2-521

Related references

C1.7 --base_platform on page C1-344

C1.11 --bpabi on page C1-349

C1.106 --pltgot_opts=mode on page C1-455

C1.32 --dll on page C1-373

C1.106 --pltgot_opts=mode

Controls the generation of *Procedure Linkage Table* (PLT) entries for weak references and function calls to relocatable targets within the same file.

Note

Not supported for AArch64 state.

Syntax

`--pltgot_opts=`*`mode`*`[,`*`mode`*`,...]`

Where *`mode`* is one of the following:

`crosslr`
Calls to and from a load region marked `RELOC` go by way of the PLT.

`nocrosslr`
Calls to and from a load region marked `RELOC` do not generate PLT entries.

`noweakrefs`
Generates a `NOP` for a function call, or zero for data. No PLT entry is generated. Weak references to imported symbols remain unresolved.

`weakrefs`
Weak references produce a PLT entry. These references must be resolved at a later link stage.

Default

The default is `--pltgot_opts=nocrosslr,noweakrefs`.

Related references

C1.7 --base_platform on page C1-344

C1.105 --pltgot=type on page C1-454

C1.107 --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`*`"`

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)
    }
}
```

Use `armlink` with the command-line options:

```
--predefine="-DBASE=0x8000" --predefine="-DBASE2=0x1000000" --scatter=filename
```

This passes the command-line options: `-DBASE=0x8000 -DBASE2=0x1000000` to the compiler to preprocess the scatter file.

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 references

C6.11 Preprocessing a scatter file on page C6-640

C1.121 --scatter=filename on page C1-470

C1.108 --preinit, --no_preinit

Enables the linker to use a different image pre-initialization routine if required.

Syntax

`--preinit=`*`symbol`*

If `--preinit=`*`symbol`* is not specified then the default symbol `__arm_preinit_` is assumed.

`--no_preinit` does not take a symbol argument.

Effect

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 processes as R_ARM_ABS32.

Related references

C1.51 --fpic on page C1-392

C1.111 --ref_pre_init, --no_ref_pre_init on page C1-460

C1.6 --bare_metal_pie on page C1-343

C1.109 --privacy (armlink)

Modifies parts of an image to help protect your code.

Usage

The effect of this option is different for images and object files.

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 references

C1.79 --locals, --no_locals on page C1-425

C1.101 --partial on page C1-449

D1.47 --privacy (fromelf) on page D1-780

D1.57 --strip=option[,option,...] on page D1-790

Related information

Options to protect code in object files with fromelf

C1.110 --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.

Usage

The default reference added is `__cpp_initialize__aeabi_`. To change this you can use `--cppinit`.

Use `--no_ref_cpp_init` if you are not going to use the Arm libraries.

Default

The default is `--ref_cpp_init`.

Related references

C1.21 --cppinit, --no_cppinit on page C1-360

Related information

C++ initialization, construction and destruction

C1.111 --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`.

Related references

C1.51 --fpic on page C1-392
C1.108 --preinit, --no_preinit on page C1-457
C1.6 --bare_metal_pie on page C1-343

C1.112 --reloc

Creates a single relocatable load region with contiguous execution regions.

Note

This option is deprecated. Use the *BPABI* on page C8-685 or the *Base Platform linking model* on page C9-705.

Note

Not supported for AArch64 state.

Usage

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.

Restrictions

You cannot use `--reloc` with `--scatter`.

You cannot use this option with `--xo_base`.

Related concepts

C6.13.2 Type 1 image, one load region and contiguous execution regions on page C6-643

C3.2.4 Type 3 image structure, multiple load regions and non-contiguous execution regions on page C3-540

Related information

Base Platform ABI for the Arm Architecture

ELF for the Arm Architecture

C1.113 --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.

Related references

C1.28 --diag_remark=tag[,tag,…] (armlink) on page C1-369
C1.42 --errors=filename on page C1-383

C1.114 --remove, --no_remove

Enables or disables the removal of unused input sections from the image.

Usage

An input section is considered used if it contains an entry point, or if it is referred to from a used section.

By default, unused section elimination is disabled when building *dynamically linked libraries* (DLLs) or shared objects. Use `--remove` to enable unused section elimination.

Use `--remove` with the `--keep` option to retain specific sections in a normal build.

Default

The default is `--remove`.

The default is `--no_remove` in any of these situations:
- You specify `--base_platform` or `--bpabi` with `--dll`.
- You specify `--shared` and `--sysv`.

Related concepts

C4.2 Elimination of unused sections on page C4-563
C3.9 How the linker performs library searching, selection, and scanning on page C3-555
C4.1 Elimination of common section groups on page C4-562

Related references

C1.7 --base_platform on page C1-344
C1.11 --bpabi on page C1-349
C1.123 --shared on page C1-473
C1.144 --sysv on page C1-495
C1.32 --dll on page C1-373
C1.67 --keep=section_id (armlink) on page C1-410

C1.115 --ro_base=address

Sets both the load and execution addresses of the region containing the RO output section at a specified address.

Syntax

`--ro_base=address`

Where *`address`* must be word-aligned.

Usage

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.

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`.

Restrictions

You cannot use `--ro_base` with:

- `--scatter`.

Related references

C1.116 --ropi on page C1-465
C1.117 --rosplit on page C1-466
C1.118 --rw_base=address on page C1-467
C1.161 --xo_base=address on page C1-512
C1.165 --zi_base=address on page C1-516
C1.123 --shared on page C1-473
C1.144 --sysv on page C1-495
C1.121 --scatter=filename on page C1-470

C1.116 --ropi

Makes the load and execution region containing the RO output section position-independent.

Note

Not supported for AArch64 state.

Usage

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`.

Restrictions

You cannot use `--ropi`:

- With `--fpic`, `--scatter`, or `--xo_base`.
- When an object file contains execute-only sections.

Related references

C1.115 --ro_base=address on page C1-464
C1.117 --rosplit on page C1-466
C1.118 --rw_base=address on page C1-467
C1.161 --xo_base=address on page C1-512
C1.165 --zi_base=address on page C1-516
C1.123 --shared on page C1-473
C1.144 --sysv on page C1-495
C1.121 --scatter=filename on page C1-470

C1.117 --rosplit

Splits the default RO load region into two RO output sections.

The RO load region is split into the RO output sections:

- `RO-CODE`.
- `RO-DATA`.

Restrictions

You cannot use `--rosplit` with:

- `--scatter`.

Related references

C1.115 --ro_base=address on page C1-464
C1.116 --ropi on page C1-465
C1.118 --rw_base=address on page C1-467
C1.161 --xo_base=address on page C1-512
C1.165 --zi_base=address on page C1-516
C1.123 --shared on page C1-473
C1.144 --sysv on page C1-495
C1.121 --scatter=filename on page C1-470

C1.118 --rw_base=address

Sets the execution addresses of the region containing the RW output section at a specified address.

Syntax

`--rw_base=address`

Where *`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 references

C1.115 --ro_base=address on page C1-464
C1.116 --ropi on page C1-465
C1.117 --rosplit on page C1-466
C1.161 --xo_base=address on page C1-512
C1.165 --zi_base=address on page C1-516
C1.123 --shared on page C1-473
C1.144 --sysv on page C1-495
C1.121 --scatter=filename on page C1-470

C1.119 --rwpi

Makes the load and execution region containing the RW and ZI output section position-independent.

Note

Not supported for AArch64 state.

Usage

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:

- 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 is used when regions are copied, decompressed, or initialized.

Restrictions

You cannot use `--rwpi`:

- With `--fpic` `--scatter`, or `--xo_base`.
- When an object file contains execute-only sections.

Related references

C1.123 --shared on page C1-473

C1.144 --sysv on page C1-495

C1.130 --split on page C1-481

C1.121 --scatter=filename on page C1-470

C1.120 --scanlib, --no_scanlib

Enables or disables scanning of the Arm libraries to resolve references.

Use `--no_scanlib` if you want to link your own libraries.

Default

The default is `--scanlib`. However, if you specify `--shared`, then the default is `--no_scanlib`.

Related references

C1.132 --stdlib on page C1-483

C1.121 --scatter=filename

Creates an image memory map using the scatter-loading description that is contained in the specified file.

The description provides grouping and placement details of the various regions and sections in the image.

Syntax

`--scatter=filename`

Where *filename* is the name of a scatter file.

Usage

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:

- `--bpabi`.
- `--first`.
- `--last`.
- `--partial`.
- `--reloc`.
- `--ro_base`.
- `--ropi`.
- `--rosplit`.
- `--rw_base`.
- `--rwpi`.
- `--split`.
- `--xo_base`.
- `--zi_base`.

You can use `--dll` when specified with `--base_platform`.

Related concepts

C6.4.5 Examples of using placement algorithms for .ANY sections on page C6-622

C6.4.8 Behavior when .ANY sections overflow because of linker-generated content on page C6-627

Related references

C1.1 --any_contingency on page C1-337

C1.3 --any_sort_order=order on page C1-340

C1.7 --base_platform on page C1-344

C6.11 Preprocessing a scatter file on page C6-640

C1.48 --first=section_id on page C1-389

C1.70 --last=section_id on page C1-415

C1.115 --ro_base=address on page C1-464

C1.116 --ropi on page C1-465

C1.117 --rosplit on page C1-466

C1.118 --rw_base=address on page C1-467

C1.119 --rwpi on page C1-468

C1.130 --split on page C1-481

C1.161 --xo_base=address on page C1-512

C1.165 --zi_base=address on page C1-516

C1.11 --bpabi on page C1-349
C1.32 --dll on page C1-373
C1.101 --partial on page C1-449
C1.112 --reloc on page C1-461
C1.123 --shared on page C1-473
C1.144 --sysv on page C1-495
Chapter C6 Scatter-loading Features on page C6-595

C1.122 --section_index_display=type

Changes the display of the index column when printing memory map output.

Syntax

`--section_index_display=`*`type`*

Where *`type`* is 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.

Usage

Use this option with `--map`.

Default

The default is `--section_index_display=internal`.

Related references

C1.86 --map, --no_map on page C1-434
C1.146 --tiebreaker=option on page C1-497

C1.123 --shared

Creates a *System V* (SysV) shared object.

Default

This option is disabled by default.

Syntax

`--shared`

Parameters

None.

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 references

C1.11 --bpabi on page C1-349
C1.32 --dll on page C1-373
C1.51 --fpic on page C1-392
C1.59 --import_unresolved, --no_import_unresolved on page C1-400
C1.110 --ref_cpp_init, --no_ref_cpp_init on page C1-459
C1.114 --remove, --no_remove on page C1-463
C1.123 --shared on page C1-473
C1.128 --soname=name on page C1-478
C1.144 --sysv on page C1-495
Chapter C8 BPABI and SysV Shared Libraries and Executables on page C8-685

C1.124 --show_cmdline (armlink)

Outputs the command line used by the `armasm`, `armlink`, `fromelf`, and `armar` tools.

Usage

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 references

C1.56 --help (armlink) on page C1-397

C1.159 --via=filename (armlink) on page C1-510

C1.125 --show_full_path

Displays the full path name of an object in any diagnostic messages.

Usage

If the file representing object `obj` has full path name `path/to/obj` then the linker displays `path/to/obj` instead of `obj` in any diagnostic message.

Related references

C1.126 --show_parent_lib on page C1-476
C1.127 --show_sec_idx on page C1-477

C1.126 --show_parent_lib

Displays the library name containing an object in any diagnostic messages.

Usage

If an object `obj` comes from library `lib`, then this option displays `lib(obj)` instead of `obj` in any diagnostic messages.

Related references

C1.125 --show_full_path on page C1-475

C1.127 --show_sec_idx on page C1-477

C1.127 --show_sec_idx

Displays the section index, `sh_idx`, of section in the originating object.

Example

If section `sec` has section index 3 then it is displayed as `sec:3` in all diagnostic messages.

Related references

C1.125 --show_full_path on page C1-475
C1.126 --show_parent_lib on page C1-476

C1.128 --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`

`name` is 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 references

Chapter C8 BPABI and SysV Shared Libraries and Executables on page C8-685

C1.129 --sort=algorithm

Specifies the sorting algorithm used by the linker to determine the order of sections in an output image.

Syntax

`--sort=`*`algorithm`*

where *`algorithm`* is 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 is more dependent on the order of input sections than using, say, `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.

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 is less dependent 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 according to the name of the section, and then by input order if the names are the same.

`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.

Usage

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.

Default

The default algorithm is `--sort=Lexical`. In large region mode, the default algorithm is `--sort=AvgCallDepth`.

Related concepts

C3.3 Section placement with the linker on page C3-543

C7.4 Execution region descriptions on page C7-664

Related references

C1.146 --tiebreaker=option on page C1-497

C1.69 --largeregions, --no_largeregions on page C1-413

C7.4.3 Execution region attributes on page C7-666

C1.130 --split

Splits the default load region, that contains the RO and RW output sections, into separate load regions.

Usage

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.

Restrictions

You cannot use `--split` with `--scatter`.

Related concepts

C3.1 The structure of an Arm® ELF image on page C3-528

Related references

C1.121 --scatter=filename on page C1-470

C1.123 --shared on page C1-473

C1.144 --sysv on page C1-495

C1.131 --startup=symbol, --no_startup

Enables the linker to use alternative C libraries with a different startup symbol if required.

Syntax

`--startup=`*`symbol`*

By default, *`symbol`* is set to `__main`.

`--no_startup` does not take a *`symbol`* argument.

Usage

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.

Default

The default is `--startup=__main`.

Related references

C1.41 --entry=location on page C1-382

C1.132 --stdlib

Specifies the C++ library to use.

Note

This topic includes descriptions of [ALPHA] features. See *Support level definitions* on page A1-39.

Syntax

`--stdlib=`*`library_option`*

where *`library_option`* is one of the following:

`libc++`
The standard C++ library.

`threaded_libc++ [ALPHA]`
The threaded standard C++ library.

Usage

C++ objects compiled with `armclang` and linked with `armlink` use libc++ by default.

Related information

C++ libraries and multithreading [ALPHA]

C1.133 --strict

Instructs the linker to perform additional conformance checks, such as reporting conditions that might result in failures.

Usage

`--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.

Related concepts

C3.13 The strict family of linker options on page C3-559

Related references

C1.134 --strict_flags, --no_strict_flags on page C1-485

C1.135 --strict_ph, --no_strict_ph on page C1-486

C1.137 --strict_relocations, --no_strict_relocations on page C1-488

C1.138 --strict_symbols, --no_strict_symbols on page C1-489

C1.139 --strict_visibility, --no_strict_visibility on page C1-490

C1.30 --diag_suppress=tag[,tag,…] (armlink) on page C1-371

C1.31 --diag_warning=tag[,tag,…] (armlink) on page C1-372

C1.27 --diag_error=tag[,tag,…] (armlink) on page C1-368

C1.42 --errors=filename on page C1-383

Related information

Procedure Call Standard for the Arm Architecture (AAPCS)

C1.134 --strict_flags, --no_strict_flags

Prevent or allow the generation of the EF_ARM_HASENTRY flag.

Usage

The option `--strict_flags` prevents the EF_ARM_HASENTRY flag from being generated.

Default

The default is `--no_strict_flags`.

Related concepts

C3.13 The strict family of linker options on page C3-559

Related references

C1.133 --strict on page C1-484

C1.135 --strict_ph, --no_strict_ph on page C1-486

C1.137 --strict_relocations, --no_strict_relocations on page C1-488

C1.138 --strict_symbols, --no_strict_symbols on page C1-489

C1.139 --strict_visibility, --no_strict_visibility on page C1-490

Related information

Arm ELF Specification (SWS ESPC 0003 B-02)

C1.135 --strict_ph, --no_strict_ph

Enables or disables the sorting of the Program Header table entries.

Usage

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.

Default

The default is `--strict_ph`.

Related concepts

C3.13 The strict family of linker options on page C3-559

Related references

C1.133 --strict on page C1-484

C1.134 --strict_flags, --no_strict_flags on page C1-485

C1.137 --strict_relocations, --no_strict_relocations on page C1-488

C1.138 --strict_symbols, --no_strict_symbols on page C1-489

C1.139 --strict_visibility, --no_strict_visibility on page C1-490

C1.136 --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.

Note

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
```

Related information

L6238E

C1.137 --strict_relocations, --no_strict_relocations

Enables you to ensure *Application Binary Interface* (ABI) compliance of legacy or third party objects.

Usage

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.

Default

The default is `--no_strict_relocations`.

Related concepts

C3.13 The strict family of linker options on page C3-559

Related references

C1.133 --strict on page C1-484

C1.134 --strict_flags, --no_strict_flags on page C1-485

C1.135 --strict_ph, --no_strict_ph on page C1-486

C1.138 --strict_symbols, --no_strict_symbols on page C1-489

C1.139 --strict_visibility, --no_strict_visibility on page C1-490

C1.138 --strict_symbols, --no_strict_symbols

Checks whether or not a mapping symbol type matches an ABI symbol type.

Usage

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.

Default

The default is `--no_strict_symbols`.

Example

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 concepts

C3.13 The strict family of linker options on page C3-559

C5.1 About mapping symbols on page C5-578

Related references

C1.133 --strict on page C1-484

C1.134 --strict_flags, --no_strict_flags on page C1-485

C1.135 --strict_ph, --no_strict_ph on page C1-486

C1.137 --strict_relocations, --no_strict_relocations on page C1-488

C1.139 --strict_visibility, --no_strict_visibility on page C1-490

C1.139 --strict_visibility, --no_strict_visibility

Prevents or allows a hidden visibility reference to match against a shared object.

Usage

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.

Default

The default is `--strict_visibility`.

Related concepts

C3.13 The strict family of linker options on page C3-559

Related references

C1.133 --strict on page C1-484
C1.134 --strict_flags, --no_strict_flags on page C1-485
C1.135 --strict_ph, --no_strict_ph on page C1-486
C1.137 --strict_relocations, --no_strict_relocations on page C1-488
C1.138 --strict_symbols, --no_strict_symbols on page C1-489

C1.140 --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`.

Related references

C1.77 --list_mapping_symbols, --no_list_mapping_symbols on page C1-423

C1.141 --symdefs=filename

Creates a file containing the global symbol definitions from the output image.

Syntax

`--symdefs=`*`filename`*

where *`filename`* is the name of the text file to contain the global symbol definitions.

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.

Usage

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 concepts

C5.5 Access symbols in another image on page C5-587

C1.142 --symver_script=filename

Enables implicit symbol versioning.

Syntax

`--symver_script=filename`

where *`filename`* is a symbol version script.

Related concepts

C8.6 Symbol versioning on page C8-702

C1.143 --symver_soname

Enables implicit symbol versioning to force static binding.

Note

Not supported for AArch64 state.

Usage

Where a symbol has no defined version, the linker uses the *shared object name* (`SONAME`) contained in the file being linked.

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`.

Related concepts

C8.6 Symbol versioning on page C8-702

Related information

Base Platform ABI for the Arm Architecture

C1.144 --sysv

Creates a *System V* (SysV) formatted ELF executable file.

Default

This option is disabled by default.

Syntax

`--sysv`

Parameters

None.

Restrictions

You cannot use this option if an object file contains execute-only sections.

Operation

Note

ELF files produced with the `--sysv` option are demand-paged compliant.

Related concepts

C2.6 SysV linking model overview on page C2-524

C3.4 Linker support for creating demand-paged files on page C3-546

Related references

--add_shared_references, --no_add_shared_references linker option

--arm_linux linker option

C1.11 --bpabi on page C1-349

C1.32 --dll on page C1-373

C1.114 --remove, --no_remove on page C1-463

C1.51 --fpic on page C1-392

C1.59 --import_unresolved, --no_import_unresolved on page C1-400

--linker_script=ld_script linker option

--prelink_support, --no_prelink_support linker option

--sysroot=path linker option

--runpath=pathlist linker option

--use_sysv_default_script, --no_use_sysv_default_script linker option

C10.3 IMPORT steering file command on page C10-714

Chapter C8 BPABI and SysV Shared Libraries and Executables on page C8-685

C1.145 --tailreorder, --no_tailreorder

Moves tail calling sections immediately before their target, if possible, to optimize the branch instruction at the end of a section.

Note

Not supported for AArch64 state.

Usage

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.

Default

The default is `--no_tailreorder`.

Restrictions

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 concepts

C4.7 Linker reordering of tail calling sections on page C4-571

C4.6 About branches that optimize to a NOP on page C4-570

Related references

C1.12 --branchnop, --no_branchnop on page C1-350

C1.146 --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.

Syntax

`--tiebreaker=`*`option`*

where *`option`* is 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.

Default

The default option is `creation`.

Related references

C1.129 --sort=algorithm on page C1-479
C1.86 --map, --no_map on page C1-434
C1.3 --any_sort_order=order on page C1-340

C1.147 --unaligned_access, --no_unaligned_access

Enable or disable unaligned accesses to data on Arm architecture-based processors.

Usage

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

  This error message can be downgraded.

Default

The default is `--unaligned_access`.

C1.148 --undefined=symbol

Prevents the removal of a specified symbol if it is undefined.

Syntax

`--undefined=`*`symbol`*

Usage

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 references

C1.149 --undefined_and_export=symbol on page C1-500

C1.67 --keep=section_id (armlink) on page C1-410

C1.149 --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`

Usage

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.

Related references

C1.96 --override_visibility on page C1-444

C1.148 --undefined=symbol on page C1-499

C1.67 --keep=section_id (armlink) on page C1-410

C10.1 EXPORT steering file command on page C10-712

C1.150 --unresolved=symbol

Takes each reference to an undefined symbol and matches it to the global definition of the specified symbol.

Syntax

`--unresolved=symbol`

`symbol` must be both defined and global, otherwise it appears in the list of undefined symbols and the link step fails.

Usage

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 references

C1.148 --undefined=symbol on page C1-499

C1.149 --undefined_and_export=symbol on page C1-500

C1.151 --use_definition_visibility

Enables the linker to use the visibility of the definition in preference to the visibility of a reference when combining symbols.

Usage

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 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.

C1.152 --userlibpath=pathlist

Specifies a list of paths that the linker is to use to search for user libraries.

Syntax

`--userlibpath=`*`pathlist`*

Where *`pathlist`* is 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 concepts

C3.9 How the linker performs library searching, selection, and scanning on page C3-555

Related references

C1.72 --libpath=pathlist on page C1-417

C1.153 --veneerinject, --no_veneerinject

Enables or disables the placement of veneers outside of the sorting order for the Execution Region.

Usage

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
```

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.

Related references

C1.69 --largeregions, --no_largeregions on page C1-413
C1.154 --veneer_inject_type=type on page C1-505
C1.4 --api, --no_api on page C1-341
C1.129 --sort=algorithm on page C1-479

C1.154 --veneer_inject_type=type

Controls the veneer layout when `--largeregions` mode is on.

Syntax

`--veneer_inject_type=`*`type`*

Where *`type`* is 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.

`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 references

C1.60 --info=topic[,topic,…] (armlink) on page C1-401
C1.153 --veneerinject, --no_veneerinject on page C1-504
C1.155 --veneer_pool_size=size on page C1-506
C1.69 --largeregions, --no_largeregions on page C1-413

C1.155 --veneer_pool_size=size

Sets the contingency size for the veneer pool in an execution region.

Syntax

`--veneer_pool_size=pool`

where `pool` is the size in bytes.

Default

The default size is 102400 bytes.

Related references

C1.154 --veneer_inject_type=type on page C1-505

C1.156 --veneershare, --no_veneershare

Enables or disables veneer sharing. Veneer sharing can cause a significant decrease in image size.

Default

The default is `--veneershare`.

Related concepts

C3.6.2 Veneer sharing on page C3-548

C3.6 Linker-generated veneers on page C3-548

C3.6.3 Veneer types on page C3-549

C3.6.4 Generation of position independent to absolute veneers on page C3-550

Related references

C1.65 --inlineveneer, --no_inlineveneer on page C1-408

C1.103 --piveneer, --no_piveneer on page C1-451

C1.24 --crosser_veneershare, --no_crosser_veneershare on page C1-365

C1.157 --verbose

Prints detailed information about the link operation, including the objects that are included and the libraries from which they are taken.

Usage

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`.

Related references

C1.76 --list=filename on page C1-422

C1.92 --muldefweak, --no_muldefweak on page C1-440

C1.150 --unresolved=symbol on page C1-501

C1.158 --version_number (armlink)

Displays the version of armlink that you are using.

Usage

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 references

C1.56 --help (armlink) on page C1-397

C1.160 --vsn (armlink) on page C1-511

C1.159 --via=filename (armlink)

Reads an additional list of input filenames and tool options from `filename`.

Syntax

`--via=filename`

Where `filename` is the name of a via file containing options to be included on the command line.

Usage

You can enter multiple `--via` options on the `armasm`, `armlink`, `fromelf`, and `armar` command lines. You can also include the `--via` options within a via file.

Related concepts

C.1 Overview of via files on page Appx-C-1172

Related references

C.2 Via file syntax rules on page Appx-C-1173

C1.160 --vsn (armlink)

Displays the version information and the license details.

Note

`--vsn` is intended to report the version information for manual inspection. The `Component` line indicates the release of Arm Compiler 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`.

Example

```
> armlink --vsn
Product: ARM Compiler N.n
Component: ARM Compiler N.n
Tool: armlink [tool_id]
license_type
Software supplied by: ARM Limited
```

Related references

C1.56 --help (armlink) on page C1-397

C1.158 --version_number (armlink) on page C1-509

C1.161 --xo_base=address

Specifies the base address of an *execute-only* (XO) execution region.

Syntax

`--xo_base=`*`address`*

Where *`address`* must be word-aligned.

Usage

When you specify `--xo_base`:

- 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.

Restrictions

You can use `--xo_base` only with the bare-metal linking model.

Note

XO memory is supported only for Armv7-M and Armv8-M architectures.

You cannot use `--xo_base` with:

- `--reloc`.
- `--ropi`.
- `--rwpi`.
- `--scatter`.

Related concepts

C2.2 Bare-metal linking model overview on page C2-519

Related references

C1.115 --ro_base=address on page C1-464
C1.116 --ropi on page C1-465
C1.117 --rosplit on page C1-466
C1.118 --rw_base=address on page C1-467
C1.165 --zi_base=address on page C1-516
C1.121 --scatter=filename on page C1-470
C1.123 --shared on page C1-473
C1.144 --sysv on page C1-495

C1.162 --xref, --no_xref

Lists to stdout all cross-references between input sections.

Default

The default is `--no_xref`.

Related references

C1.163 --xrefdbg, --no_xrefdbg on page C1-514
C1.164 --xref{from|to}=object(section) on page C1-515
C1.76 --list=filename on page C1-422

C1.163 --xrefdbg, --no_xrefdbg

Lists to stdout all cross-references between input debug sections.

Default

The default is `--no_xrefdbg`.

Related references

C1.162 --xref, --no_xref on page C1-513
C1.164 --xref{from|to}=object(section) on page C1-515
C1.76 --list=filename on page C1-422

C1.164 --xref{from|to}=object(section)

Lists to stdout cross-references from and to input sections.

Syntax

`--xref{from|to}=`*`object`*`(`*`section`*`)`

Usage

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.

Related references

C1.162 --xref, --no_xref on page C1-513

C1.163 --xrefdbg, --no_xrefdbg on page C1-514

C1.76 --list=filename on page C1-422

C1.165 --zi_base=address

Specifies the base address of an ER_ZI execution region.

Syntax

`--zi_base=address`

Where *`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:

- `--bpabi`.
- `--base_platform`.
- `--reloc`.
- `--rwpi`.
- `--split`.

You cannot use `--zi_base` with `--scatter`.

Related references

C1.115 --ro_base=address on page C1-464
C1.116 --ropi on page C1-465
C1.117 --rosplit on page C1-466
C1.118 --rw_base=address on page C1-467
C1.161 --xo_base=address on page C1-512
C1.121 --scatter=filename on page C1-470
C1.11 --bpabi on page C1-349

Chapter C2
Linking Models Supported by armlink

Describes the linking models supported by the Arm linker, `armlink`.

It contains the following sections:

- *C2.1 Overview of linking models* on page C2-518.
- *C2.2 Bare-metal linking model overview* on page C2-519.
- *C2.3 Partial linking model overview* on page C2-520.
- *C2.4 Base Platform Application Binary Interface (BPABI) linking model overview* on page C2-521.
- *C2.5 Base Platform linking model overview* on page C2-522.
- *C2.6 SysV linking model overview* on page C2-524.
- *C2.7 Concepts common to both BPABI and SysV linking models* on page C2-525.

C2.1 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 Executables (PIE)

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.

Note

The Bare-metal PIE feature is deprecated.

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.

BPABI

This model supports the DLL-like *Base Platform Application Binary Interface* (BPABI). It is intended to produce applications and DLLs that can run on a platform OS that varies in complexity. The memory model is restricted according to the *Base Platform ABI for the Arm® Architecture* (IHI 0037 C).

Note

Not supported for AArch64 state.

Base Platform

This is an extension to the BPABI model to support scatter-loading.

Note

Not supported for AArch64 state.

You can combine related options in each model to tighten control over the output.

Related concepts

C2.2 Bare-metal linking model overview on page C2-519

C2.3 Partial linking model overview on page C2-520

C2.4 Base Platform Application Binary Interface (BPABI) linking model overview on page C2-521

C2.5 Base Platform linking model overview on page C2-522

Related references

Chapter C8 BPABI and SysV Shared Libraries and Executables on page C8-685

Related information

Base Platform ABI for the Arm Architecture

C2.2 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 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 that can be relocated or are position-independent using a scatter file or command-line options.
- Identify the symbols that can 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 concepts

C3.1.4 Methods of specifying an image memory map with the linker on page C3-532

C2.4 Base Platform Application Binary Interface (BPABI) linking model overview on page C2-521

C9.2 Scatter files for the Base Platform linking model on page C9-708

Related references

C1.161 --xo_base=address on page C1-512

C1.36 --edit=file_list on page C1-377

C1.112 --reloc on page C1-461

C1.115 --ro_base=address on page C1-464

C1.116 --ropi on page C1-465

C1.117 --rosplit on page C1-466

C1.118 --rw_base=address on page C1-467

C1.119 --rwpi on page C1-468

C1.121 --scatter=filename on page C1-470

C1.130 --split on page C1-481

C1.165 --zi_base=address on page C1-516

Chapter C10 Linker Steering File Command Reference on page C10-711

C2.3 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 concepts

C5.6 Edit the symbol tables with a steering file on page C5-590

Related references

C5.6.3 Steering file format on page C5-591

Chapter C10 Linker Steering File Command Reference on page C10-711

C1.36 --edit=file_list on page C1-377

C1.101 --partial on page C1-449

C2.4 Base Platform Application Binary Interface (BPABI) linking model overview

The *Base Platform Application Binary Interface* (BPABI) is a meta-standard for third parties to generate their own platform-specific image formats.

The BPABI model produces as much dynamic information as possible without focusing on any specific platform.

Note

BPABI is not supported for AArch64 state.

To link with this model, use the `--bpabi` command-line option. Other linker command-line options supported by this model are:

- `--dll`.
- `--force_so_throw`, `--no_force_so_throw`.
- `--pltgot=`*`type`*.
- `--ro_base=`*`address`*.
- `--rosplit`.
- `--rw_base=`*`address`*.
- `--rwpi`.

Be aware of the following:

- You cannot use scatter-loading. However, the Base Platform linking model supports scatter-loading.
- The model by default assumes that shared objects cannot throw a C++ exception (`--no_force_so_throw`).
- The default value of the `--pltgot` option is `direct`.
- You must use symbol versioning to ensure that all the required symbols are available at load time.

Related concepts

C2.2 Bare-metal linking model overview on page C2-519

C8.6 Symbol versioning on page C8-702

Related references

C1.11 --bpabi on page C1-349

C1.32 --dll on page C1-373

C1.50 --force_so_throw, --no_force_so_throw on page C1-391

C1.105 --pltgot=type on page C1-454

C1.115 --ro_base=address on page C1-464

C1.117 --rosplit on page C1-466

C1.118 --rw_base=address on page C1-467

C1.119 --rwpi on page C1-468

Related information

Base Platform ABI for the Arm Architecture

C2.5 Base Platform linking model overview

The Base Platform linking model enables you to create dynamically linkable images that do not have the memory map enforced by the *Base Platform Application Binary Interface* (BPABI) linking model.

The Base Platform linking model enables you to:

- Create images with a memory map described in a scatter file.
- Have dynamic relocations so the images can be dynamically linked. The dynamic relocations can also target within the same image.

———— **Note** ————

Base Platform is not supported for AArch64 state.

————————————

———— **Note** ————

The BPABI specification places constraints on the memory model that can be violated using scatter-loading. However, because Base Platform is a superset of BPABI, it is possible to create a BPABI conformant image with Base Platform.

————————————

To link with the Base Platform model, use the `--base_platform` command-line option.

If you specify this option, the linker acts as if you specified `--bpabi`, with the following exceptions:

- Scatter-loading is available with `--scatter`. If you do not specify `--scatter`, then the standard BPABI memory model scatter file is used.
- The following options are available:
  - `--dll`.
  - `--force_so_throw`, `--no_force_so_throw`.
  - `--pltgot=`*`type`*.
  - `--rosplit`.
- The default value of the `--pltgot` option is different to that for `--bpabi`:
  - For `--base_platform`, the default is `--pltgot=none`.
  - For `--bpabi` the default is `--pltgot=direct`.
- Each load region containing code might require a *Procedure Linkage Table* (PLT) section to indirect calls from the load region to functions where the address is not known at static link time. The PLT section for a load region `LR` must be placed in `LR` and be accessible at all times to code within `LR`.

  If you do not use a scatter file, the linker can ensure that the PLT section is placed correctly, and contains entries for calls only to imported symbols. If you specify a scatter file, the linker might not be able to find a suitable location to place the PLT.

  To ensure calls between relocated load regions use a PLT entry:
  - Use the `--pltgot=direct` option to turn on PLT generation.
  - Use the `--pltgot_opts=crosslr` option to add entries in the PLT for calls from and to `RELOC` load regions. The linker generates a PLT for each load region so that calls do not have to be extended to reach a distant PLT.

Be aware of the following:

- The model by default assumes that shared objects cannot throw a C++ exception (`--no_force_so_throw`).
- You must use symbol versioning to ensure that all the required symbols are available at load time.
- There are restrictions on the type of scatter files you can use.

Related concepts

C9.1 Restrictions on the use of scatter files with the Base Platform model on page C9-706

C9.2 Scatter files for the Base Platform linking model on page C9-708

C2.4 Base Platform Application Binary Interface (BPABI) linking model overview on page C2-521

C3.1.4 Methods of specifying an image memory map with the linker on page C3-532

C8.6 Symbol versioning on page C8-702

Related references

C1.7 --base_platform on page C1-344

C1.32 --dll on page C1-373

C1.106 --pltgot_opts=mode on page C1-455

C1.117 --rosplit on page C1-466

C1.121 --scatter=filename on page C1-470

C1.105 --pltgot=type on page C1-454

C2.6 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.
- 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.

C2.7 Concepts common to both BPABI and SysV linking models

For both *Base Platform Application Binary Interface* (BPABI) and *System V* (SysV) linking models, images and shared objects usually run on an existing operating platform.

There are many similarities between the BPABI and the SysV models. The main differences are in the memory model, and in the *Procedure Linkage Table* (PLT) and *Global Offset Table* (GOT) structure, collectively referred to as PLTGOT. There are many options that are common to both models.

Restrictions of the BPABI and SysV

Both the BPABI and SysV models have the following restrictions:

- Unused section elimination treats every symbol that is externally visible as an entry point.
- Read write data compression is not permitted.
- `__AT` sections are not permitted.

Chapter C3

Image Structure and Generation

Describes the image structure and the functionality available in the Arm linker, `armlink`, to generate images.

It contains the following sections:

- *C3.1 The structure of an Arm® ELF image* on page C3-528.
- *C3.2 Simple images* on page C3-536.
- *C3.3 Section placement with the linker* on page C3-543.
- *C3.4 Linker support for creating demand-paged files* on page C3-546.
- *C3.5 Linker reordering of execution regions containing T32 code* on page C3-547.
- *C3.6 Linker-generated veneers* on page C3-548.
- *C3.7 Command-line options used to control the generation of C++ exception tables* on page C3-552.
- *C3.8 Weak references and definitions* on page C3-553.
- *C3.9 How the linker performs library searching, selection, and scanning* on page C3-555.
- *C3.10 How the linker searches for the Arm® standard libraries* on page C3-556.
- *C3.11 Specifying user libraries when linking* on page C3-557.
- *C3.12 How the linker resolves references* on page C3-558.
- *C3.13 The strict family of linker options* on page C3-559.

C3.1 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.

This section contains the following subsections:

- *C3.1.1 Views of the image at each link stage* on page C3-528.
- *C3.1.2 Input sections, output sections, regions, and program segments* on page C3-529.
- *C3.1.3 Load view and execution view of an image* on page C3-530.
- *C3.1.4 Methods of specifying an image memory map with the linker* on page C3-532.
- *C3.1.5 Image entry points* on page C3-533.
- *C3.1.6 Restrictions on image structure* on page C3-535.

C3.1.1 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.

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:

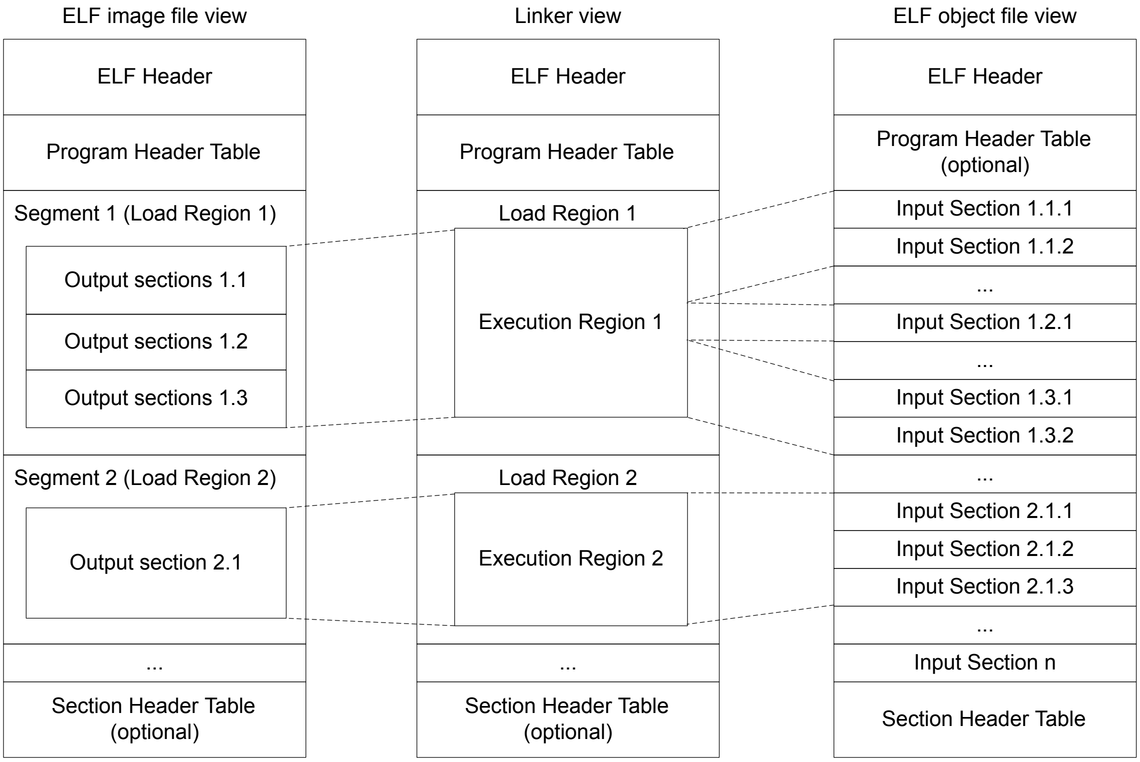


Figure C3-1 Relationship between sections, regions, and segments

C3.1.2 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.

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 Armv7-M and Armv8-M architectures.

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 concepts

C3.1.1 Views of the image at each link stage on page C3-528

C3.1.4 Methods of specifying an image memory map with the linker on page C3-532

C3.3 Section placement with the linker on page C3-543

C3.1.3 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:

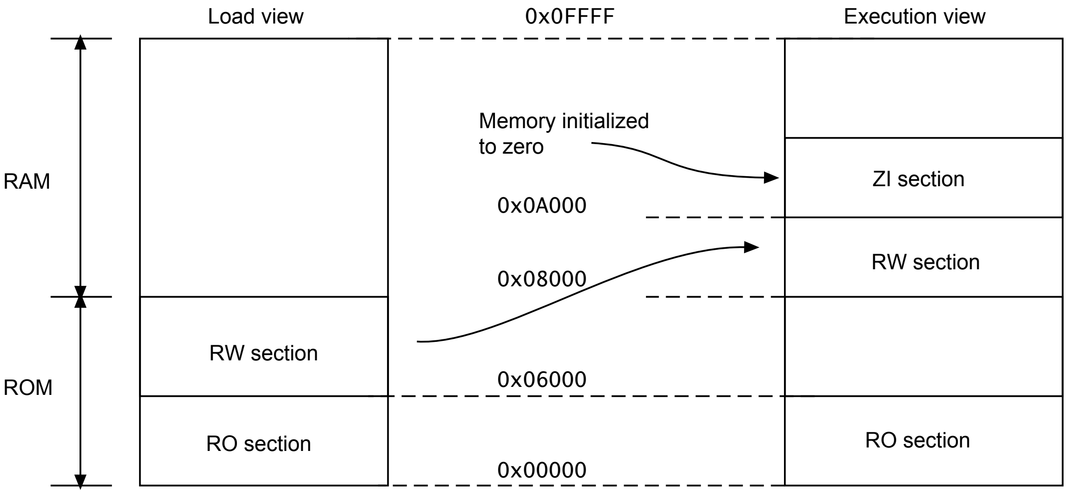


Figure C3-2 Load and execution memory maps for an image without an XO section

The following figure shows load and execution views for an image with an XO section:

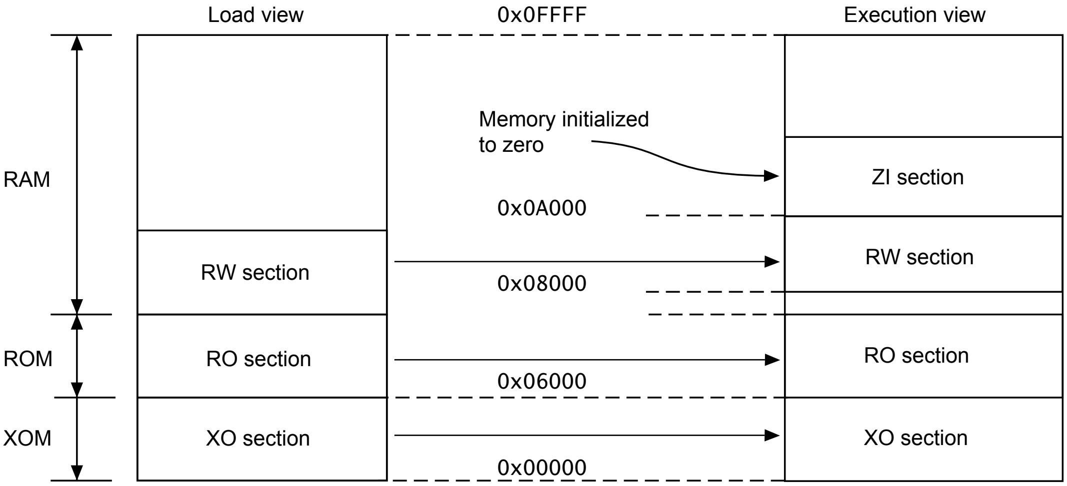


Figure C3-3 Load and execution memory maps for an image with an XO section

Note

XO memory is supported only for Armv7-M and Armv8-M architectures.

The following table compares the load and execution views:

Table C3-1 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 concepts

C3.1.1 Views of the image at each link stage on page C3-528
C3.1.4 Methods of specifying an image memory map with the linker on page C3-532
C3.3 Section placement with the linker on page C3-543
C3.1.2 Input sections, output sections, regions, and program segments on page C3-529

C3.1.4 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.

Table C3-2 Comparison of scatter file and equivalent command-line options

Scatter file	Equivalent command-line options
`LR1 0x0000 0x20000` `{`	
`    ER_RO 0x0 0x2000` `    {`	`--ro_base=0x0`

`        init.o (INIT, +FIRST)`
`        *(+RO)`
`    }`
	`--first=init.o(init)`

Table C3-2 Comparison of scatter file and equivalent command-line options (continued)

Scatter file	Equivalent command-line options

```
    ER_RW 0x400000
    {
        *(+RW)
    }
```

```
--rw_base=0x400000
```

```
    ER_ZI 0x405000
    {
        *(+ZI)
    }
}
```

```
--zi_base=0x405000
```

```
LR_XO 0x8000 0x4000
{
```

```
    ER_XO 0x8000
    {
        *(XO)
    }
}
```

```
--xo_base=0x8000
```

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 concepts

C3.1.3 Load view and execution view of an image on page C3-530
C3.2 Simple images on page C3-536
C3.1 The structure of an Arm® ELF image on page C3-528
C3.1.2 Input sections, output sections, regions, and program segments on page C3-529

Related references

C1.48 --first=section_id on page C1-389
C1.70 --last=section_id on page C1-415
C1.115 --ro_base=address on page C1-464
C1.116 --ropi on page C1-465
C1.117 --rosplit on page C1-466
C1.118 --rw_base=address on page C1-467
C1.119 --rwpi on page C1-468
C1.121 --scatter=filename on page C1-470
C1.130 --split on page C1-481
C1.161 --xo_base=address on page C1-512
C1.165 --zi_base=address on page C1-516

C3.1.5 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. 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.

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.

Entry points set by the `ENTRY` directive

You can select one of many possible entry points for an image. An image can have only one entry point.

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.

For C and C++ programs, the `__main()` function in the C library is also an entry point.

If an embedded image is to be used by a loader, it must have a single initial entry point specified in the header. Use the `--entry` command-line option to select the entry point.

The initial entry point for an image

There can be only one initial entry point for an image, otherwise linker warning `L6305W` is output.

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.

If you do not use the `--entry` option to specify the initial entry point, then:

- 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=0x0`, or optionally `0xFFFF0000` for processors that are using high vectors.

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 concepts

C6.2 Root region and the initial entry point on page C6-602

Related references

C1.41 --entry=location on page C1-382

F6.26 ENTRY on page F6-1049

Related information

List of the armlink error and warning messages

C3.1.6 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.

C3.2 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.

This section contains the following subsections:

- *C3.2.1 Types of simple image* on page C3-536.
- *C3.2.2 Type 1 image structure, one load region and contiguous execution regions* on page C3-537.
- *C3.2.3 Type 2 image structure, one load region and non-contiguous execution regions* on page C3-538.
- *C3.2.4 Type 3 image structure, multiple load regions and non-contiguous execution regions* on page C3-540.

C3.2.1 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.

Related concepts

C6.13 Equivalent scatter-loading descriptions for simple images on page C6-643
C3.2.2 Type 1 image structure, one load region and contiguous execution regions on page C3-537
C3.2.3 Type 2 image structure, one load region and non-contiguous execution regions on page C3-538

C3.2.4 Type 3 image structure, multiple load regions and non-contiguous execution regions on page C3-540

Related references

C1.115 --ro_base=address on page C1-464
C1.117 --rosplit on page C1-466
C1.118 --rw_base=address on page C1-467
C1.121 --scatter=filename on page C1-470
C1.130 --split on page C1-481
C1.161 --xo_base=address on page C1-512
C1.165 --zi_base=address on page C1-516

C3.2.2 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.

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 *execute-only* (XO) code:

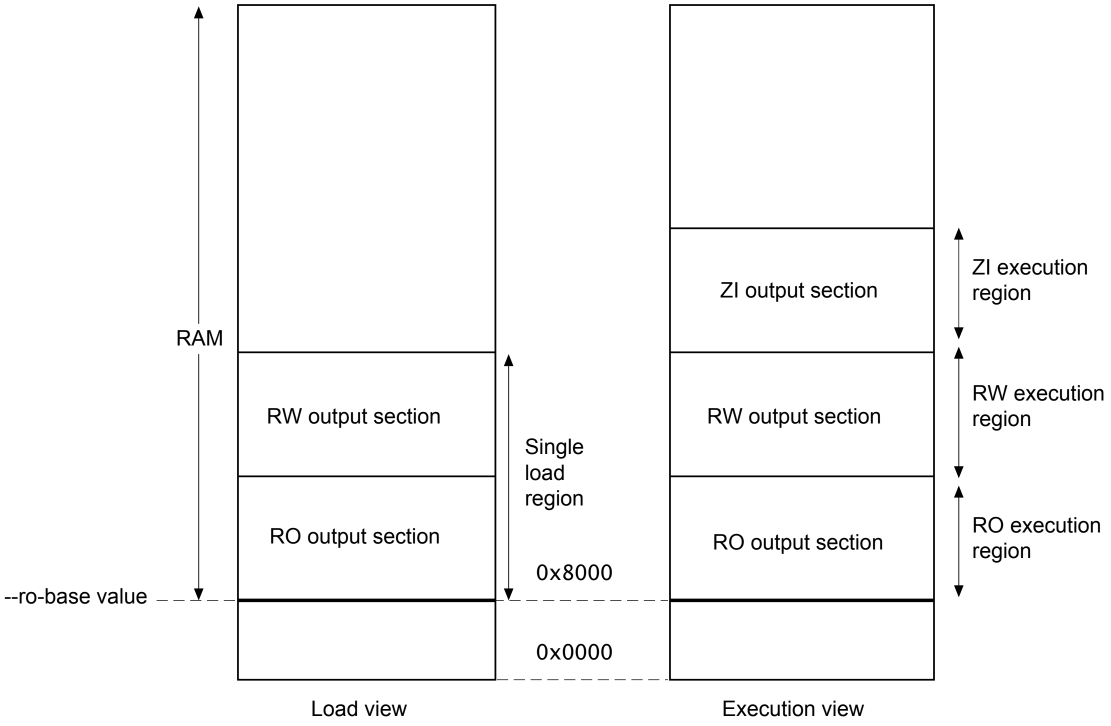


Figure C3-4 Simple Type 1 image

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.

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 Armv7-M and Armv8-M architectures.

Related concepts

C3.1 The structure of an Arm® ELF image on page C3-528
C3.1.2 Input sections, output sections, regions, and program segments on page C3-529
C3.1.3 Load view and execution view of an image on page C3-530

Related references

C1.115 --ro_base=address on page C1-464
C1.161 --xo_base=address on page C1-512
C1.165 --zi_base=address on page C1-516

C3.2.3 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:

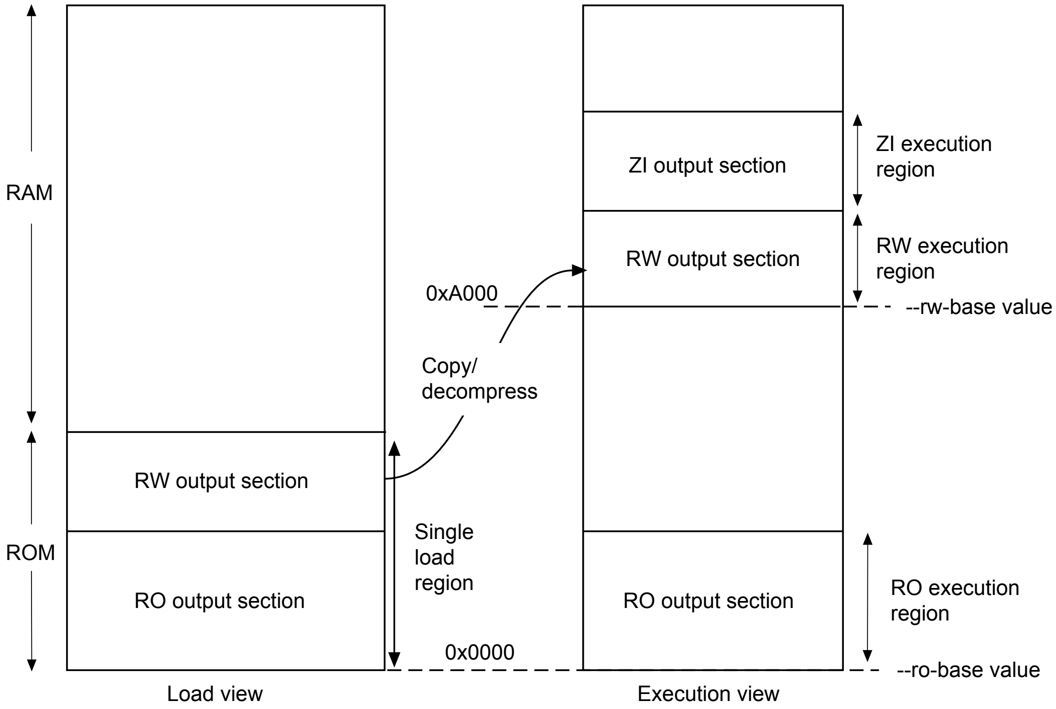


Figure C3-5 Simple Type 2 image

Use the following command for images of this type:

```
armlink --cpu=8-A.32 --ro_base=0x0 --rw_base=0xA000
```

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.

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.

Note

XO memory is supported only for Armv7-M and Armv8-M architectures.

Related concepts

C3.1 The structure of an Arm® ELF image on page C3-528

C3.1.2 Input sections, output sections, regions, and program segments on page C3-529

C3.1.3 Load view and execution view of an image on page C3-530

C3.2.2 Type 1 image structure, one load region and contiguous execution regions on page C3-537

Related references

C1.115 --ro_base=address on page C1-464

C1.118 --rw_base=address on page C1-467

C1.161 --xo_base=address on page C1-512

C1.165 --zi_base=address on page C1-516

C3.2.4 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:

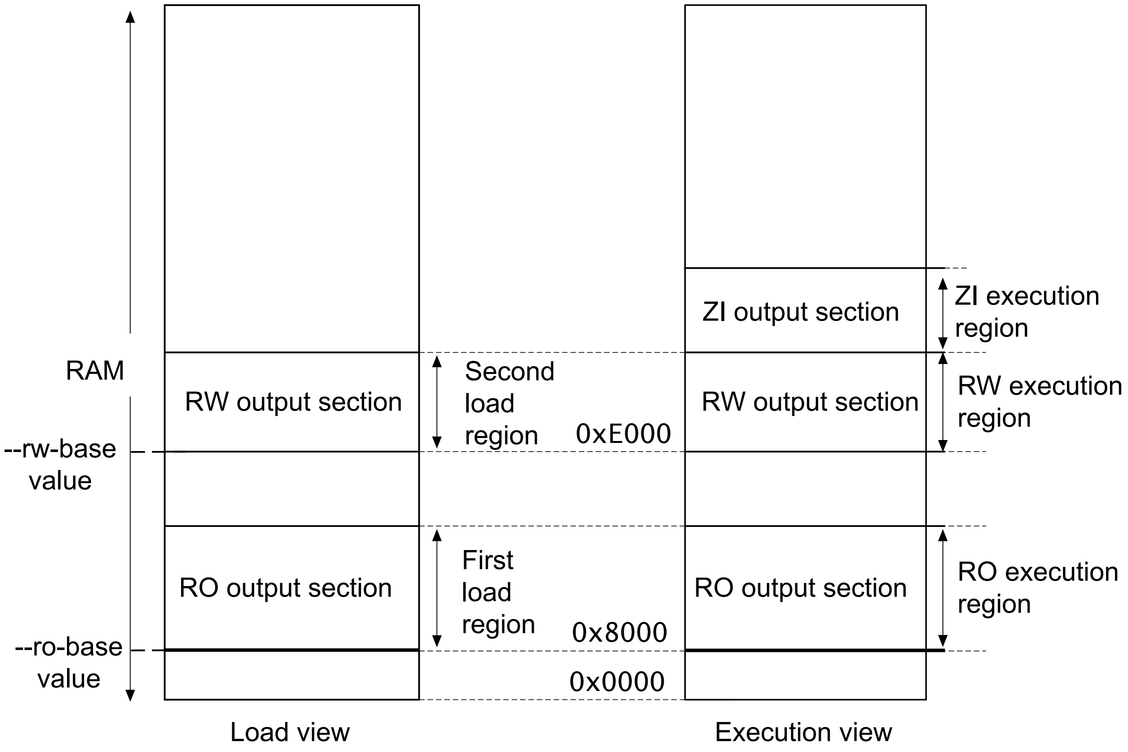


Figure C3-6 Simple Type 3 image

Use the following command for images of this type:

```
armlink --cpu=8-A.32 --split --ro_base 0x8000 --rw_base 0xE000
```

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.

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.

Note

XO memory is supported only for Armv7-M and Armv8-M architectures.

Related concepts

C3.1 The structure of an Arm® ELF image on page C3-528

C3.1.2 Input sections, output sections, regions, and program segments on page C3-529

C3.1.3 Load view and execution view of an image on page C3-530

C3.2.3 Type 2 image structure, one load region and non-contiguous execution regions on page C3-538

Related references

C1.115 --ro_base=address on page C1-464

C1.118 --rw_base=address on page C1-467

C1.161 --xo_base=address on page C1-512

C1.130 --split on page C1-481

C3.3 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.

This section contains the following subsections:

- *C3.3.1 Default section placement* on page C3-543.
- *C3.3.2 Section placement with the FIRST and LAST attributes* on page C3-544.
- *C3.3.3 Section alignment with the linker* on page C3-545.

C3.3.1 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.
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 Armv7-M and Armv8-M architectures.

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.

Handling 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.

Related concepts

C6.2.3 Methods of placing functions and data at specific addresses on page C6-605

C6.3 Example of how to explicitly place a named section with scatter-loading on page C6-617

C3.1 The structure of an Arm® ELF image on page C3-528

C3.6 Linker-generated veneers on page C3-548

C3.3.3 Section alignment with the linker on page C3-545

C3.1.2 Input sections, output sections, regions, and program segments on page C3-529

C3.3.2 Section placement with the FIRST and LAST attributes on page C3-544

C3.5 Linker reordering of execution regions containing T32 code on page C3-547

Related references

C6.4 Placement of unassigned sections on page C6-619

C7.5.2 Syntax of an input section description on page C7-672

C1.129 --sort=algorithm on page C1-479

C3.3.2 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.

  Caution

  `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 concepts

C3.1 The structure of an Arm® ELF image on page C3-528

C3.1.2 Input sections, output sections, regions, and program segments on page C3-529

C3.1.3 Load view and execution view of an image on page C3-530

C6.1 The scatter-loading mechanism on page C6-596

Related references

C7.5.2 Syntax of an input section description on page C7-672

C1.48 --first=section_id on page C1-389

C1.70 --last=section_id on page C1-415

C3.3.3 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

From version 6.6 and later, `--no_legacyalign` is deprecated.

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 concepts

C7.6.9 Example of aligning a base address in execution space but still tightly packed in load space on page C7-683

Related tasks

C6.9 Aligning regions to page boundaries on page C6-638

Related references

C7.3.3 Load region attributes on page C7-659

C1.71 --legacyalign, --no_legacyalign on page C1-416

C7.4.3 Execution region attributes on page C7-666

C3.4 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.

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

This is an example of a demand paged scatter file:

```
LR1 GetPageSize() + SizeOfHeaders()
{
    ER_RO +0
    {
        *(+RO)
    }
}
LR2 +GetPageSize()
{
    ER_RW +0
    {
        *(+RW)
    }
    ER_ZI +0
    {
        *(+ZI)
    }
}
```

Related concepts

C6.1 The scatter-loading mechanism on page C6-596

Related tasks

C6.9 Aligning regions to page boundaries on page C6-638

Related references

C1.121 --scatter=filename on page C1-470

C7.6.7 GetPageSize() function on page C7-682

C1.99 --paged on page C1-447

C1.100 --pagesize=pagesize on page C1-448

C7.6.8 SizeOfHeaders() function on page C7-682

C3.5 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 concepts

C3.6 Linker-generated veneers on page C3-548

Related references

C1.69 --largeregions, --no_largeregions on page C1-413

C3.6 Linker-generated veneers

Veneers are small sections of code generated by the linker and inserted into your program.

This section contains the following subsections:

- *C3.6.1 What is a veneer?* on page C3-548.
- *C3.6.2 Veneer sharing* on page C3-548.
- *C3.6.3 Veneer types* on page C3-549.
- *C3.6.4 Generation of position independent to absolute veneers* on page C3-550.
- *C3.6.5 Reuse of veneers when scatter-loading* on page C3-550.
- *C3.6.6 Generation of secure gateway veneers* on page C3-551.

C3.6.1 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.

  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 concepts

C3.6.2 Veneer sharing on page C3-548

C3.6.3 Veneer types on page C3-549

C3.6.4 Generation of position independent to absolute veneers on page C3-550

C3.6.5 Reuse of veneers when scatter-loading on page C3-550

C3.6.2 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)
    }
}
```

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 concepts

C3.6.1 What is a veneer? on page C3-548

C6.1 The scatter-loading mechanism on page C6-596

Related references

Chapter C7 Scatter File Syntax on page C7-655

C1.156 --veneershare, --no_veneershare on page C1-507

C3.6.3 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.

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.
- 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 concepts

C3.6.1 What is a veneer? on page C3-548

Related references

C1.88 --max_veneer_passes=value on page C1-436

C1.65 --inlineveneer, --no_inlineveneer on page C1-408

C3.6.4 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.

Note

Not supported for AArch64 state.

Related concepts

C3.6.1 What is a veneer? on page C3-548

Related references

C1.88 --max_veneer_passes=value on page C1-436

C1.103 --piveneer, --no_piveneer on page C1-451

C3.6.5 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.
  - 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 concepts

C3.6.1 What is a veneer? on page C3-548

C7.3.4 Inheritance rules for load region address attributes on page C7-661

C7.3.5 Inheritance rules for the RELOC address attribute on page C7-662

C7.4.4 Inheritance rules for execution region address attributes on page C7-669

Related references

C7.3.3 Load region attributes on page C7-659

C3.6.6 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.

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 concepts

C6.6 Placement of CMSE veneer sections for a Secure image on page C6-631

Related references

C1.57 --import_cmse_lib_in=filename on page C1-398

C1.58 --import_cmse_lib_out=filename on page C1-399

Related information

Building Secure and Non-secure Images Using Armv8-M Security Extensions

C3.7 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 references

C1.31 --diag_warning=tag[,tag,…] (armlink) on page C1-372

C1.43 --exceptions, --no_exceptions on page C1-384

B1.21 -fexceptions, -fno-exceptions on page B1-78

C3.8 Weak references and definitions

Weak references and definitions provide additional flexibility in the way the linker includes various functions and variables in a build.

Weak references and definitions are typically used in connection with library functions.

Weak references

If the linker cannot resolve normal, non-weak, references to symbols from the content loaded so far, it attempts to do so by finding the symbol in a library:

- If it is unable to find such a reference, the linker reports an error.
- 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 ensures the section is not removed by the linker as an unused section. Each non-weak reference must be resolved by exactly one definition. If there are multiple definitions, the linker reports an error.

Symbols can be given weak binding by the compiler and assembler.

The linker does not load an object from a library to resolve a weak reference. It is able to resolve the weak reference only if the definition is included in the image for other reasons. The weak reference does not cause the linker to mark the section containing the definition as used, so it might be removed by the linker as unused. The definition might already exist in the image for several 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.
- The symbol definition is in a section that has been specified using `--keep`, or contains an `ENTRY` point.
- 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.

In summary, a weak reference is resolved if the definition is already included in the image, but it does not determine if that definition is included.

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 just does not happen.

Weak definitions

You can mark a function or variable definition as weak in a source file. A weak symbol definition is then present in the created object file.

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.

Example of a weak reference

A library contains a function `foo()`, that is called in some builds of an application but not in others. If it is used, `init_foo()` must be called first. You can use weak references to automate the call to `init_foo()`.

The library can define `init_foo()` and `foo()` in the same ELF section. The application initialization code must call `init_foo()` weakly. If the application includes `foo()` for any reason, it also includes `init_foo()` and this is called from the initialization code. In any builds that do not include `foo()`, the call to `init_foo()` is removed by the linker.

Typically, the code for multiple functions defined within a single source file is placed into a single ELF section by the compiler. However, certain build options might alter this behavior, so you must use them with caution if your build is relying on the grouping of files into ELF sections. The compiler command-line option `-ffunction-sections` results in each function being placed in its own section. In this example, compiling the library with this option results in `foo()` and `init_foo()` being placed in separate sections. Therefore `init_foo()` is not automatically included in the build due to a call to `foo()`.

In this example, there is no need to rebuild the initialization code between builds that include `foo()` and do not include `foo()`. There is also no possibility of accidentally building an application with a version of the initialization code that does not call `init_foo()`, and other parts of the application that call `foo()`.

An example of `foo.c` source code that is typically built into a library is:

```
void init_foo()
{
    // Some initialization code
}
void foo()
{
     // A function that is included in some builds
     // and requires init_foo() to be called first.
}
```

An example of `init.c` is:

```
__attribute__((weak)) void init_foo(void);
int main(void)
{
    init_foo();
    //  Rest of code that may make calls to foo() directly or indirectly.
}
```

An example of a weak reference generated by the assembler is:

```
init.s:
  main:
    ...
    bl      init_foo
    // Rest of code

    .weak   init_foo
```

Example of a weak definition

You can provide a simple or dummy implementation of a function as a weak definition. This enables you to build software with defined behavior without having to provide a full implementation of the function. It also enables you to provide a full implementation for some builds if required.

Related concepts

C3.9 How the linker performs library searching, selection, and scanning on page C3-555

C3.12 How the linker resolves references on page C3-558

Related references

C1.67 --keep=section_id (armlink) on page C1-410

C1.114 --remove, --no_remove on page C1-463

F6.28 EXPORT or GLOBAL on page F6-1051

F6.46 IMPORT and EXTERN on page F6-1071

F6.26 ENTRY on page F6-1049

Related information

NOP

B

C3.9 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.

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 concepts

C3.10 How the linker searches for the Arm® standard libraries on page C3-556

Related references

C1.67 --keep=section_id (armlink) on page C1-410

C1.114 --remove, --no_remove on page C1-463

C1.120 --scanlib, --no_scanlib on page C1-469

C3.10 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 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.

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 concepts

C3.9 How the linker performs library searching, selection, and scanning on page C3-555

Related references

C1.72 --libpath=pathlist on page C1-417

Related information

C and C++ library naming conventions

The C and C++ libraries

Toolchain environment variables

C3.11 Specifying user libraries when linking

You can specify your own libraries when linking.

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
```

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 concepts

C3.10 How the linker searches for the Arm® standard libraries on page C3-556

Related references

C1.72 --libpath=pathlist on page C1-417

C1.152 --userlibpath=pathlist on page C1-503

Related information

The C and C++ libraries

Toolchain environment variables

C3.12 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 step *1.b*.

      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 step *1.a*. 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.

Related concepts

C3.9 How the linker performs library searching, selection, and scanning on page C3-555

C3.10 How the linker searches for the Arm® standard libraries on page C3-556

Related tasks

C3.11 Specifying user libraries when linking on page C3-557

Related references

C1.72 --libpath=pathlist on page C1-417

Related information

Toolchain environment variables

List of the armlink error and warning messages

C3.13 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 references

C1.133 --strict on page C1-484

C1.137 --strict_relocations, --no_strict_relocations on page C1-488

C1.138 --strict_symbols, --no_strict_symbols on page C1-489

C1.139 --strict_visibility, --no_strict_visibility on page C1-490

Chapter C4
Linker Optimization Features

Describes the optimization features available in the Arm linker, `armlink`.

It contains the following sections:

- *C4.1 Elimination of common section groups* on page C4-562.
- *C4.2 Elimination of unused sections* on page C4-563.
- *C4.3 Optimization with RW data compression* on page C4-564.
- *C4.4 Function inlining with the linker* on page C4-567.
- *C4.5 Factors that influence function inlining* on page C4-568.
- *C4.6 About branches that optimize to a NOP* on page C4-570.
- *C4.7 Linker reordering of tail calling sections* on page C4-571.
- *C4.8 Restrictions on reordering of tail calling sections* on page C4-572.
- *C4.9 Linker merging of comment sections* on page C4-573.
- *C4.10 Merging identical constants* on page C4-574.

C4.1 Elimination of common section groups

The linker can detect multiple copies of section groups, and discard the additional copies.

Arm Compiler 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.
- 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 concepts

C4.2 Elimination of unused sections on page C4-563

C3.1.2 Input sections, output sections, regions, and program segments on page C3-529

Related references

C1.8 --bestdebug, --no_bestdebug on page C1-346

C4.2 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 `--remove`, `--no_remove`, `--first`, `--last`, and `--keep` linker options.

Unused section elimination requires an entry point. Therefore, if no entry point is specified for an image, use the `--entry` linker option to specify an entry point.

Use the `--info unused` linker option to instruct the linker to generate a list of the unused sections that it eliminates.

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 Armv8-M Security Extensions.
- 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 `-ffunction-sections` compiler command-line option to instruct the compiler to generate one ELF section for each function in the source file.

Related concepts

C4.1 Elimination of common section groups on page C4-562

C3.1.2 Input sections, output sections, regions, and program segments on page C3-529

C3.8 Weak references and definitions on page C3-553

Related references

C1.114 --remove, --no_remove on page C1-463

C1.41 --entry=location on page C1-382

C1.48 --first=section_id on page C1-389

C1.67 --keep=section_id (armlink) on page C1-410

C1.70 --last=section_id on page C1-415

C1.60 --info=topic[,topic,…] (armlink) on page C1-401

Related information

Building Secure and Non-secure Images Using Armv8-M Security Extensions

C4.3 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.

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.

This section contains the following subsections:
- *C4.3.1 How the linker chooses a compressor* on page C4-564.
- *C4.3.2 Options available to override the compression algorithm used by the linker* on page C4-564.
- *C4.3.3 How compression is applied* on page C4-565.
- *C4.3.4 Considerations when working with RW data compression* on page C4-565.

C4.3.1 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 concepts

C4.3.2 Options available to override the compression algorithm used by the linker on page C4-564
C4.3 Optimization with RW data compression on page C4-564
C4.3.3 How compression is applied on page C4-565
C4.3.4 Considerations when working with RW data compression on page C4-565

C4.3.2 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 …
```

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 concepts

C4.3 Optimization with RW data compression on page C4-564

C4.3.3 How compression is applied on page C4-565

C4.3.1 How the linker chooses a compressor on page C4-564

C4.3.4 Considerations when working with RW data compression on page C4-565

Related references

C1.25 --datacompressor=opt on page C1-366

C4.3.3 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.
- The length of the phrase.

Related concepts

C4.3 Optimization with RW data compression on page C4-564

C4.3.2 Options available to override the compression algorithm used by the linker on page C4-564

C4.3.1 How the linker chooses a compressor on page C4-564

C4.3.4 Considerations when working with RW data compression on page C4-565

Related references

C1.25 --datacompressor=opt on page C1-366

C4.3.4 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 concepts

C4.3 Optimization with RW data compression on page C4-564

C4.3.1 How the linker chooses a compressor on page C4-564

C4.3.2 Options available to override the compression algorithm used by the linker on page C4-564

C4.3.3 How compression is applied on page C4-565

Related references

C5.3.3 Load$$ execution region symbols on page C5-581

Chapter C6 Scatter-loading Features on page C6-595

C1.86 --map, --no_map on page C1-434

Chapter C7 Scatter File Syntax on page C7-655

C4.4 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.

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 Armv7-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 concepts

C4.5 Factors that influence function inlining on page C4-568

C4.2 Elimination of unused sections on page C4-563

Related references

C1.60 --info=topic[,topic,…] (armlink) on page C1-401

C1.63 --inline, --no_inline on page C1-406

C4.5 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:

  Table C4-1 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 concepts

C4.6 About branches that optimize to a NOP on page C4-570

Related information

Conditional instructions

ADD

B

CPS

IT

MOV

MRS (PSR to general-purpose register)

MSR (general-purpose register to PSR)

C4.6 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 concepts

C4.7 Linker reordering of tail calling sections on page C4-571

Related references

C1.12 --branchnop, --no_branchnop on page C1-350

C4.7 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 concepts

C4.6 About branches that optimize to a NOP on page C4-570

C4.8 Restrictions on reordering of tail calling sections on page C4-572

C3.6.3 Veneer types on page C3-549

Related references

C1.60 --info=topic[,topic,…] (armlink) on page C1-401

C1.145 --tailreorder, --no_tailreorder on page C1-496

C4.8 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 concepts

C4.7 Linker reordering of tail calling sections on page C4-571

C4.9 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 references

C1.20 --comment_section, --no_comment_section on page C1-359

C1.46 --filtercomment, --no_filtercomment on page C1-387

D1.57 --strip=option[,option,…] on page D1-790

C4.10 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 6. If you compile with the `armclang -ffunction-sections` option, the merge is more efficient. This option is the default.

The following procedure is an example that shows the merging feature.

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.

Results:
Because `0xdeadbeef` is a difficult constant to create using instructions, a literal pool is created, for example:

```
...
f1:
    .fnstart
@ BB#0:
    ldr    r0, __arm_cp.0_0
    bx    lr
    .p2align    2
@ BB#1:
__arm_cp.0_0:
    .long    3735928559              @ 0xdeadbeef
...
    .fnend

...
    .code    16                      @ @f2
    .thumb_func
f2:
    .fnstart
@ BB#0:
    ldr    r0, __arm_cp.1_0
    bx    lr
    .p2align    2
@ BB#1:
__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`

   Results: 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 references

C1.91 --merge_litpools, --no_merge_litpools on page C1-439

B1.14 -ffunction-sections, -fno-function-sections on page B1-69

Chapter C5
Accessing and Managing Symbols with armlink

Describes how to access and manage symbols with the Arm linker, `armlink`.

It contains the following sections:

- *C5.1 About mapping symbols* on page C5-578.
- *C5.2 Linker-defined symbols* on page C5-579.
- *C5.3 Region-related symbols* on page C5-580.
- *C5.4 Section-related symbols* on page C5-585.
- *C5.5 Access symbols in another image* on page C5-587.
- *C5.6 Edit the symbol tables with a steering file* on page C5-590.
- *C5.7 Use of $Super$$ and $Sub$$ to patch symbol definitions* on page C5-593.

C5.1 About mapping symbols

Mapping symbols are generated by the compiler and assembler to identify various inline transitions.

For Armv7-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

`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 references

C1.77 --list_mapping_symbols, --no_list_mapping_symbols on page C1-423

C1.138 --strict_symbols, --no_strict_symbols on page C1-489

F5.1 Symbol naming rules on page F5-987

D1.57 --strip=option[,option,…] on page D1-790

D1.59 --text on page D1-793

Related information

ELF for the Arm Architecture

C5.2 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.
  - RO linker defined symbols cannot be defined with respect to a region that contains only XO sections.

Note

XO memory is supported only for Armv7-M and Armv8-M architectures.

Related concepts

C5.3.7 Methods of importing linker-defined symbols in C and C++ on page C5-583

C5.3.8 Methods of importing linker-defined symbols in Arm® assembly language on page C5-584

C5.3 Region-related symbols

The linker generates various types of region-related symbols that you can access if required.

This section contains the following subsections:
- *C5.3.1 Types of region-related symbols* on page C5-580.
- *C5.3.2 Image$$ execution region symbols* on page C5-580.
- *C5.3.3 Load$$ execution region symbols* on page C5-581.
- *C5.3.4 Load$$LR$$ load region symbols* on page C5-582.
- *C5.3.5 Region name values when not scatter-loading* on page C5-583.
- *C5.3.6 Linker defined symbols and scatter files* on page C5-583.
- *C5.3.7 Methods of importing linker-defined symbols in C and C++* on page C5-583.
- *C5.3.8 Methods of importing linker-defined symbols in Arm® assembly language* on page C5-584.

C5.3.1 Types of 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 these symbols are generated for each region in the scatter file.

If you are not using scatter-loading, the symbols are generated for the default region names. That is, the region names are fixed and the same types of symbol are supplied.

Related concepts

C5.3.5 Region name values when not scatter-loading on page C5-583

Related references

C5.3.2 Image$$ execution region symbols on page C5-580
C5.3.3 Load$$ execution region symbols on page C5-581
C5.3.4 Load$$LR$$ load region symbols on page C5-582

C5.3.2 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 C5-1 Image$$ execution region symbols

Symbol	**Description**
`Image$$`*`region_name`*`$$Base`	Execution address of the region.
`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.

Table C5-1 Image$$ execution region symbols (continued)

Symbol	Description
`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 concepts

C5.3.1 Types of region-related symbols on page C5-580

C5.3.3 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 C5-2 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.
`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.

Table C5-2 Load$$ execution region symbols (continued)

Symbol	Description
`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 concepts

C5.3.1 Types of region-related symbols on page C5-580
C5.3.7 Methods of importing linker-defined symbols in C and C++ on page C5-583
C5.3.8 Methods of importing linker-defined symbols in Arm® assembly language on page C5-584
C5.3.5 Region name values when not scatter-loading on page C5-583
C4.3 Optimization with RW data compression on page C4-564

Related references

C5.3.2 Image$$ execution region symbols on page C5-580
C5.3.4 Load$$LR$$ load region symbols on page C5-582
C7.4.3 Execution region attributes on page C7-666

C5.3.4 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 C5-3 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 concepts

C5.3.1 Types of region-related symbols on page C5-580

C3.1 The structure of an Arm® ELF image on page C3-528

C3.1.2 Input sections, output sections, regions, and program segments on page C3-529

C3.1.3 Load view and execution view of an image on page C3-530

C5.3.5 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`.

Related concepts

C5.3.1 Types of region-related symbols on page C5-580

C5.4 Section-related symbols on page C5-585

Related references

C5.3.2 Image$$ execution region symbols on page C5-580

C5.3.3 Load$$ execution region symbols on page C5-581

C5.3.6 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.
- Defines both stack and heap. The linker also generates special stack and heap symbols.

Related references

Chapter C6 Scatter-loading Features on page C6-595

C1.121 --scatter=filename on page C1-470

C5.3.7 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[];`.

The following examples show how to obtain the correct value:

Importing a linker-defined symbol

```
extern int Image$$ER_ZI$$Limit;
heap_base = (uintptr_t)&Image$$ER_ZI$$Limit;
```

Importing symbols that define a ZI output section

```
extern int Image$$ER_ZI$$Length;
extern char Image$$ER_ZI$$Base[];
memset(Image$$ER_ZI$$Base, 0, (size_t)&Image$$ER_ZI$$Length);
```

Related references

C5.3.2 Image$$ execution region symbols on page C5-580

C5.3.8 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.

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 references

C5.3.2 Image$$ execution region symbols on page C5-580

F6.46 IMPORT and EXTERN on page F6-1071

Related information

A32 and T32 Instructions

C5.4 Section-related symbols

Section-related symbols are symbols generated by the linker when it creates an image without scatter-loading.

This section contains the following subsections:
- *C5.4.1 Types of section-related symbols* on page C5-585.
- *C5.4.2 Image symbols* on page C5-585.
- *C5.4.3 Input section symbols* on page C5-586.

C5.4.1 Types of section-related symbols

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.
- Input section symbols, for every input section present in the image.

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*.

Related references

C5.4.2 Image symbols on page C5-585

C5.4.3 Input section symbols on page C5-586

C5.4.2 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:

Table C5-4 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

- Arm recommends 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 re-implementing `__user_setup_stackheap()` to set the heap and stack boundaries.

Related concepts

C3.2 Simple images on page C3-536

C3.8 Weak references and definitions on page C3-553

Related tasks

C6.1.4 Placing the stack and heap with a scatter file on page C6-597

Related references

C6.1.3 Linker-defined symbols that are not defined when scatter-loading on page C6-597

Related information

__user_setup_stackheap()

C5.4.3 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 C5-5 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 concepts

C3.1.2 Input sections, output sections, regions, and program segments on page C3-529

Related references

Chapter C6 Scatter-loading Features on page C6-595

C5.5 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.

This section contains the following subsections:

- *C5.5.1 Creating a symdefs file* on page C5-587.
- *C5.5.2 Outputting a subset of the global symbols* on page C5-587.
- *C5.5.3 Reading a symdefs file* on page C5-588.
- *C5.5.4 Symdefs file format* on page C5-588.

C5.5.1 Creating a symdefs file

You can specify a symdefs file on the linker command-line.

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.

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.

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 tasks

C5.5.2 Outputting a subset of the global symbols on page C5-587

C5.5.3 Reading a symdefs file on page C5-588

Related references

C5.5.4 Symdefs file format on page C5-588

C1.141 --symdefs=filename on page C1-492

C5.5.2 Outputting a subset of the global symbols

You can use a symdefs file to output a subset of the global symbols to another application.

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 concepts

C5.5 Access symbols in another image on page C5-587

Related tasks

C5.5.1 Creating a symdefs file on page C5-587

Related references

C5.5.4 Symdefs file format on page C5-588

C1.141 --symdefs=filename on page C1-492

C5.5.3 Reading a symdefs file

A symdefs file can be considered as an object file with symbol information but no code or data.

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 references

C5.5.4 Symdefs file format on page C5-588

C5.5.4 Symdefs file format

A 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`*

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

This example shows a typical symdefs file format:

```
#<SYMDEFS># ARM Linker, 6030169: Last Updated: Date
;value type name, this is an added comment
0x00008000 A __main
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 tasks

C5.5.3 Reading a symdefs file on page C5-588

C5.5.1 Creating a symdefs file on page C5-587

C5.6 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.

This section contains the following subsections:

- *C5.6.1 Specifying steering files on the linker command-line* on page C5-590.
- *C5.6.2 Steering file command summary* on page C5-590.
- *C5.6.3 Steering file format* on page C5-591.
- *C5.6.4 Hide and rename global symbols with a steering file* on page C5-592.

C5.6.1 Specifying steering files on the linker command-line

You can specify one or more steering files on the linker command-line.

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 references

C5.6.2 Steering file command summary on page C5-590

C5.6.3 Steering file format on page C5-591

C5.6.2 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.

The steering file commands are:

Table C5-6 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 tasks

C5.6.1 Specifying steering files on the linker command-line on page C5-590

Related references

C5.6.3 Steering file format on page C5-591

C1.36 --edit=file_list on page C1-377

C10.1 EXPORT steering file command on page C10-712

C10.2 HIDE steering file command on page C10-713

C10.3 IMPORT steering file command on page C10-714

C10.4 RENAME steering file command on page C10-715

C10.5 REQUIRE steering file command on page C10-716

C10.6 RESOLVE steering file command on page C10-717

C10.7 SHOW steering file command on page C10-719

C5.6.3 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.
- 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 tasks

C5.6.1 Specifying steering files on the linker command-line on page C5-590

Related references

C5.6.2 Steering file command summary on page C5-590

C10.1 EXPORT steering file command on page C10-712

C10.2 HIDE steering file command on page C10-713

C10.3 IMPORT steering file command on page C10-714

C10.4 RENAME steering file command on page C10-715

C10.5 REQUIRE steering file command on page C10-716

C10.6 RESOLVE steering file command on page C10-717
C10.7 SHOW steering file command on page C10-719

C5.6.4 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
```

Example of hiding symbols:

HIDE steering command example

```
; Hides all global symbols with the ‘internal’ prefix
HIDE internal*
```

Related concepts

C5.6 Edit the symbol tables with a steering file on page C5-590

Related tasks

C5.6.1 Specifying steering files on the linker command-line on page C5-590

Related references

C5.6.2 Steering file command summary on page C5-590
C5.5.4 Symdefs file format on page C5-588
C10.2 HIDE steering file command on page C10-713
C10.4 RENAME steering file command on page C10-715
C1.36 --edit=file_list on page C1-377

C5.7 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()`, `$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` will 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`.

Example

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

Chapter C6
Scatter-loading Features

Describes the scatter-loading features and how you use scatter files with the Arm linker, `armlink`, to create complex images.

It contains the following sections:

- *C6.1 The scatter-loading mechanism* on page C6-596.
- *C6.2 Root region and the initial entry point* on page C6-602.
- *C6.3 Example of how to explicitly place a named section with scatter-loading* on page C6-617.
- *C6.4 Placement of unassigned sections* on page C6-619.
- *C6.5 Placing veneers with a scatter file* on page C6-630.
- *C6.6 Placement of CMSE veneer sections for a Secure image* on page C6-631.
- *C6.7 Reserving an empty block of memory* on page C6-633.
- *C6.8 Placement of Arm® C and C++ library code* on page C6-635.
- *C6.9 Aligning regions to page boundaries* on page C6-638.
- *C6.10 Aligning execution regions and input sections* on page C6-639.
- *C6.11 Preprocessing a scatter file* on page C6-640.
- *C6.12 Example of using expression evaluation in a scatter file to avoid padding* on page C6-642.
- *C6.13 Equivalent scatter-loading descriptions for simple images* on page C6-643.
- *C6.14 How the linker resolves multiple matches when processing scatter files* on page C6-650.
- *C6.15 How the linker resolves path names when processing scatter files* on page C6-652.
- *C6.16 Scatter file to ELF mapping* on page C6-653.

C6.1 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.

This section contains the following subsections:

- *C6.1.1 Overview of scatter-loading* on page C6-596.
- *C6.1.2 When to use scatter-loading* on page C6-596.
- *C6.1.3 Linker-defined symbols that are not defined when scatter-loading* on page C6-597.
- *C6.1.4 Placing the stack and heap with a scatter file* on page C6-597.
- *C6.1.5 Scatter-loading command-line options* on page C6-598.
- *C6.1.6 Scatter-loading images with a simple memory map* on page C6-599.
- *C6.1.7 Scatter-loading images with a complex memory map* on page C6-600.

C6.1.1 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 concepts

C6.1.2 When to use scatter-loading on page C6-596

C6.16 Scatter file to ELF mapping on page C6-653

C3.1 The structure of an Arm® ELF image on page C3-528

Related references

C5.3 Region-related symbols on page C5-580

C6.1.2 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.

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 concepts

C6.1.1 Overview of scatter-loading on page C6-596

C6.1.3 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 re-implement `__user_setup_stackheap()`, an error message is generated.

Related concepts

C5.2 Linker-defined symbols on page C5-579

Related tasks

C6.1.4 Placing the stack and heap with a scatter file on page C6-597

C6.1.4 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.

Note

- If you re-implement `__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 the vector table. See *Stack and heap configuration* in *AN179 - Cortex®-M3 Embedded Software Development* for more details.

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.

Example:

```
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 references

C5.3 Region-related symbols on page C5-580

Related information

__user_setup_stackheap()

C6.1.5 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`*.

You can use `--scatter` with the `--base_platform` linking model.

Simple memory maps

For simple memory maps, you can place code and data with with the following memory map related command-line options:

- `--bpabi`.
- `--dll`.
- `--partial`.
- `--ro_base`.
- `--rw_base`.
- `--ropi`.
- `--rwpi`.
- `--rosplit`.
- `--split`.
- `--reloc`.
- `--xo_base`
- `--zi_base`.

Note

Apart from `--dll`, you cannot use `--scatter` with these options.

Related concepts

C2.5 Base Platform linking model overview on page C2-522
C6.1 The scatter-loading mechanism on page C6-596
C6.1.2 When to use scatter-loading on page C6-596
C6.13 Equivalent scatter-loading descriptions for simple images on page C6-643

Related references

C1.7 --base_platform on page C1-344
C1.11 --bpabi on page C1-349
C1.32 --dll on page C1-373
C1.101 --partial on page C1-449
C1.112 --reloc on page C1-461
C1.115 --ro_base=address on page C1-464
C1.116 --ropi on page C1-465
C1.117 --rosplit on page C1-466
C1.118 --rw_base=address on page C1-467
C1.119 --rwpi on page C1-468
C1.121 --scatter=filename on page C1-470
C1.130 --split on page C1-481
C1.161 --xo_base=address on page C1-512
C1.165 --zi_base=address on page C1-516
Chapter C7 Scatter File Syntax on page C7-655

C6.1.6 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:

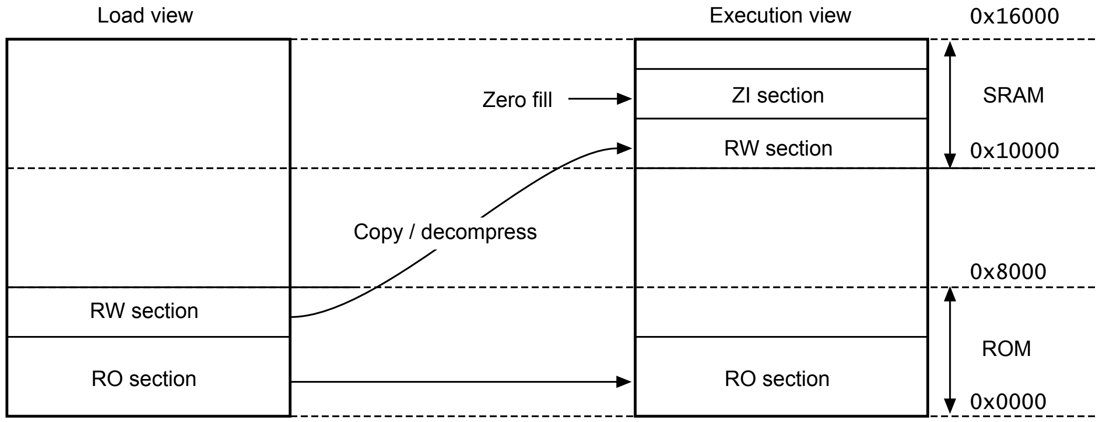


Figure C6-1 Simple scatter-loaded memory map

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),
                             ; 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 concepts

C6.16 Scatter file to ELF mapping on page C6-653

C6.1 The scatter-loading mechanism on page C6-596

C6.1.2 When to use scatter-loading on page C6-596

Related references

C1.115 --ro_base=address on page C1-464

C1.118 --rw_base=address on page C1-467

C1.161 --xo_base=address on page C1-512

C6.1.7 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:

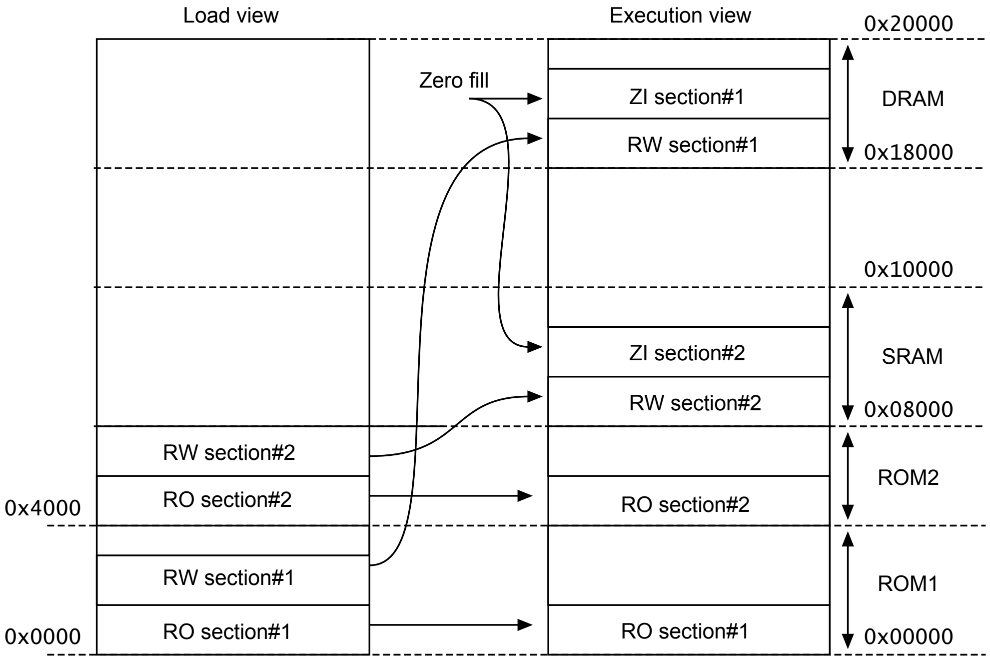


Figure C6-2 Complex memory map

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
    }
}
```

Caution

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 concepts

C6.1 The scatter-loading mechanism on page C6-596
C6.2.1 Effect of the ABSOLUTE attribute on a root region on page C6-602
C6.2.2 Effect of the FIXED attribute on a root region on page C6-604
C7.6.10 Scatter files containing relative base address load regions and a ZI execution region on page C7-683
C6.16 Scatter file to ELF mapping on page C6-653
C6.1.2 When to use scatter-loading on page C6-596

C6.2 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
}
```

This section contains the following subsections:

- *C6.2.1 Effect of the ABSOLUTE attribute on a root region* on page C6-602.
- *C6.2.2 Effect of the FIXED attribute on a root region* on page C6-604.
- *C6.2.3 Methods of placing functions and data at specific addresses* on page C6-605.
- *C6.2.4 Placing functions and data in a named section* on page C6-609.
- *C6.2.5 Placing __at sections at a specific address* on page C6-611.
- *C6.2.6 Restrictions on placing __at sections* on page C6-612.
- *C6.2.7 Automatically placing __at sections* on page C6-612.
- *C6.2.8 Manually placing __at sections* on page C6-614.
- *C6.2.9 Placing a key in flash memory with an __at section* on page C6-615.

C6.2.1 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 concepts

C6.2 Root region and the initial entry point on page C6-602

C6.2.2 Effect of the FIXED attribute on a root region on page C6-604

C7.3 Load region descriptions on page C7-658

C7.4 Execution region descriptions on page C7-664
C7.3.6 Considerations when using a relative address +offset for a load region on page C7-662
C7.4.5 Considerations when using a relative address +offset for execution regions on page C7-670
C7.3.4 Inheritance rules for load region address attributes on page C7-661
C7.3.5 Inheritance rules for the RELOC address attribute on page C7-662
C7.4.4 Inheritance rules for execution region address attributes on page C7-669

Related references

C7.3.3 Load region attributes on page C7-659
C7.4.3 Execution region attributes on page C7-666
F6.26 ENTRY on page F6-1049

C6.2.2 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:

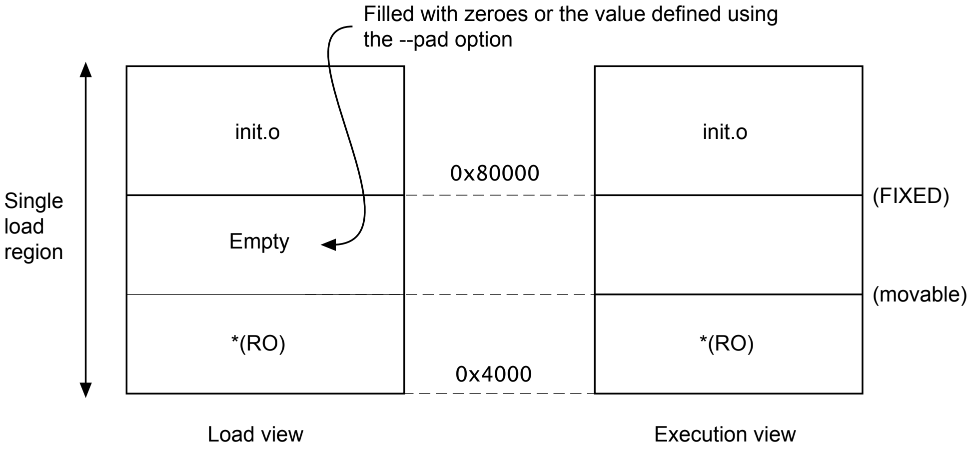


Figure C6-3 Memory map for fixed execution regions

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.

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
}
```

Related concepts

C7.4 Execution region descriptions on page C7-664

C7.3.4 Inheritance rules for load region address attributes on page C7-661

C7.3.5 Inheritance rules for the RELOC address attribute on page C7-662

C7.4.4 Inheritance rules for execution region address attributes on page C7-669

Related references

C7.3.3 Load region attributes on page C7-659

C7.4.3 Execution region attributes on page C7-666

C6.2.3 Methods of placing functions and data at specific addresses

There are various methods available to place functions and data at specific addresses.

Placing functions and data at specific addresses

To 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 rest of the 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 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 do this:

- By default, the compiler places each function and variable in individual ELF sections. This can be overridden using 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.
- Use the `.section` directive from assembly language. In assembly code, the smallest locatable unit is a `.section`.

Related concepts

C6.3 Example of how to explicitly place a named section with scatter-loading on page C6-617

C6.2.6 Restrictions on placing __at sections on page C6-612

Related tasks

C6.2.5 Placing __at sections at a specific address on page C6-611

Related references

C1.5 --autoat, --no_autoat on page C1-342

C1.86 --map, --no_map on page C1-434

C1.121 --scatter=filename on page C1-470

C1.93 -o filename, --output=filename (armlink) on page C1-441

F6.7 AREA on page F6-1027

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.

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 references

C1.5 --autoat, --no_autoat on page C1-342

C1.86 --map, --no_map on page C1-434

C1.93 -o filename, --output=filename (armlink) on page C1-441

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
       }
       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.

Related references

C1.5 --autoat, --no_autoat on page C1-342

C1.86 --map, --no_map on page C1-434

C1.93 -o filename, --output=filename (armlink) on page C1-441

C1.121 --scatter=filename on page C1-470

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
    {
    }
}
```

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 references

C1.5 --autoat, --no_autoat on page C1-342
C1.86 --map, --no_map on page C1-434
C1.93 -o filename, --output=filename (armlink) on page C1-441
C1.121 --scatter=filename on page C1-470

C6.2.4 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")))`.

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`*`")))`.

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.

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-eabi-none  -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.

Example:
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.

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 *Placing functions and data at specific addresses* on page C6-605.

Related concepts

C6.2.6 Restrictions on placing __at sections on page C6-612

Related tasks

C6.2.5 Placing __at sections at a specific address on page C6-611

Related references

C1.5 --autoat, --no_autoat on page C1-342

C1.121 --scatter=filename on page C1-470

C6.2.5 Placing __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.

Related concepts

C6.2.6 Restrictions on placing __at sections on page C6-612

Related tasks

Placing functions and data at specific addresses on page C6-605

C6.2.4 Placing functions and data in a named section on page C6-609

C6.2.7 Automatically placing __at sections on page C6-612

C6.2.8 Manually placing __at sections on page C6-614

C6.2.9 Placing a key in flash memory with an __at section on page C6-615

Related references

C1.5 --autoat, --no_autoat on page C1-342

C6.2.6 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 tasks

C6.2.5 Placing __at sections at a specific address on page C6-611

Related information

Base Platform ABI for the Arm Architecture

C6.2.7 Automatically placing __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 and execution region for the `__at` section.

All linker execution regions created by `--autoat` have the `UNINIT` scatter-loading attribute. If you require a 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 produces an error message.

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_0x00008000
int variable __attribute__((section(".ARM.__at_0x00008000")));
```

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 and execution region for the __at section
; .ARM.__at_0x8000 because it lies outside all candidate regions.
```

Related concepts

C7.4 Execution region descriptions on page C7-664

C6.2.6 Restrictions on placing __at sections on page C6-612

Related tasks

C6.2.5 Placing __at sections at a specific address on page C6-611

C6.2.8 Manually placing __at sections on page C6-614
C6.2.9 Placing a key in flash memory with an __at section on page C6-615
C6.2.4 Placing functions and data in a named section on page C6-609

Related references

C1.5 --autoat, --no_autoat on page C1-342
C1.115 --ro_base=address on page C1-464
C1.118 --rw_base=address on page C1-467
C1.161 --xo_base=address on page C1-512
C1.165 --zi_base=address on page C1-516
C7.4.3 Execution region attributes on page C7-666
C1.87 --max_er_extension=size on page C1-435

Related information

__attribute__((section("name"))) variable attribute

C6.2.8 Manually placing __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.

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 concepts

C7.4 Execution region descriptions on page C7-664
C6.2.6 Restrictions on placing __at sections on page C6-612

Related tasks

C6.2.5 Placing __at sections at a specific address on page C6-611
C6.2.7 Automatically placing __at sections on page C6-612
C6.2.9 Placing a key in flash memory with an __at section on page C6-615
C6.2.4 Placing functions and data in a named section on page C6-609

Related references

C1.5 --autoat, --no_autoat on page C1-342

C7.4.3 Execution region attributes on page C7-666

Related information

__attribute__((section("name"))) variable attribute

C6.2.9 Placing 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 concepts

C7.4 Execution region descriptions on page C7-664

C3.3.2 Section placement with the FIRST and LAST attributes on page C3-544

Related tasks

C6.2.5 Placing __at sections at a specific address on page C6-611

C6.2.7 Automatically placing __at sections on page C6-612

C6.2.8 Manually placing __at sections on page C6-614

Related references

C1.5 --autoat, --no_autoat on page C1-342

Related concepts

C6.2.1 Effect of the ABSOLUTE attribute on a root region on page C6-602

C6.2.2 Effect of the FIXED attribute on a root region on page C6-604

C3.1 The structure of an Arm® ELF image on page C3-528

Related references

C6.8 Placement of Arm® C and C++ library code on page C6-635

C6.3 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
------
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 & 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 0x1FFFF
    }
}
```

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`.

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 concepts

C6.2.2 Effect of the FIXED attribute on a root region on page C6-604
C7.3 Load region descriptions on page C7-658
C7.4 Execution region descriptions on page C7-664
C7.3.4 Inheritance rules for load region address attributes on page C7-661
C7.3.5 Inheritance rules for the RELOC address attribute on page C7-662
C7.4.4 Inheritance rules for execution region address attributes on page C7-669

Related references

C7.3.3 Load region attributes on page C7-659
C7.4.3 Execution region attributes on page C7-666
F6.26 ENTRY on page F6-1049

C6.4 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. However, 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.

In a scatter file, you can also:

- Assign a priority to a `.ANY` selector. This gives 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.

This section contains the following subsections:

- *C6.4.1 Default rules for placing unassigned sections* on page C6-619.
- *C6.4.2 Command-line options for controlling the placement of unassigned sections* on page C6-620.
- *C6.4.3 Prioritizing the placement of unassigned sections* on page C6-620.
- *C6.4.4 Specify the maximum region size permitted for placing unassigned sections* on page C6-621.
- *C6.4.5 Examples of using placement algorithms for .ANY sections* on page C6-622.
- *C6.4.6 Example of next_fit algorithm showing behavior of full regions, selectors, and priority* on page C6-624.
- *C6.4.7 Examples of using sorting algorithms for .ANY sections* on page C6-625.
- *C6.4.8 Behavior when .ANY sections overflow because of linker-generated content* on page C6-627.

C6.4.1 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 concepts

C6.14 How the linker resolves multiple matches when processing scatter files on page C6-650

C6.4.8 Behavior when .ANY sections overflow because of linker-generated content on page C6-627

Related references

C1.2 --any_placement=algorithm on page C1-338

C1.1 --any_contingency on page C1-337

C6.4 Placement of unassigned sections on page C6-619

C7.5.2 Syntax of an input section description on page C7-672

C1.60 --info=topic[,topic,...] (armlink) on page C1-401

C6.4.2 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`.

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 concepts

C6.4.5 Examples of using placement algorithms for .ANY sections on page C6-622

C6.4.6 Example of next_fit algorithm showing behavior of full regions, selectors, and priority on page C6-624

C6.4.7 Examples of using sorting algorithms for .ANY sections on page C6-625

C6.4.8 Behavior when .ANY sections overflow because of linker-generated content on page C6-627

Related references

C1.3 --any_sort_order=order on page C1-340

C1.86 --map, --no_map on page C1-434

C1.122 --section_index_display=type on page C1-472

C1.146 --tiebreaker=option on page C1-497

C1.2 --any_placement=algorithm on page C1-338

C1.1 --any_contingency on page C1-337

C6.4.3 Prioritizing the placement of unassigned sections

You can give a priority ordering when placing unassigned sections with multiple `.ANY` module selectors.

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.

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 concepts

C6.4.5 Examples of using placement algorithms for .ANY sections on page C6-622

C6.4.6 Example of next_fit algorithm showing behavior of full regions, selectors, and priority on page C6-624

C6.4.7 Examples of using sorting algorithms for .ANY sections on page C6-625

C6.4.8 Behavior when .ANY sections overflow because of linker-generated content on page C6-627

C6.14 How the linker resolves multiple matches when processing scatter files on page C6-650

Related references

C1.3 --any_sort_order=order on page C1-340

C1.86 --map, --no_map on page C1-434

C1.122 --section_index_display=type on page C1-472

C1.146 --tiebreaker=option on page C1-497

C6.4.4 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.
- 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 concepts

C6.4.5 Examples of using placement algorithms for .ANY sections on page C6-622

C6.4.6 Example of next_fit algorithm showing behavior of full regions, selectors, and priority on page C6-624

C6.4.7 Examples of using sorting algorithms for .ANY sections on page C6-625

C6.4.8 Behavior when .ANY sections overflow because of linker-generated content on page C6-627

Related references

C1.3 --any_sort_order=order on page C1-340

C1.86 --map, --no_map on page C1-434

C1.1 --any_contingency on page C1-337

C6.4.5 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:

Table C6-1 Input section properties for placement of .ANY sections

Name	Size
`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(+R0)` 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:

- 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 concepts

C6.4.2 Command-line options for controlling the placement of unassigned sections on page C6-620

C6.4.6 Example of next_fit algorithm showing behavior of full regions, selectors, and priority on page C6-624

C6.4.4 Specify the maximum region size permitted for placing unassigned sections on page C6-621

Related tasks

C6.4.3 Prioritizing the placement of unassigned sections on page C6-620

Related references

C1.121 --scatter=filename on page C1-470

C6.4.6 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 C6-2 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


    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 concepts

C6.4.4 Specify the maximum region size permitted for placing unassigned sections on page C6-621

C6.4.2 Command-line options for controlling the placement of unassigned sections on page C6-620

C6.4.5 Examples of using placement algorithms for .ANY sections on page C6-622

C6.14 How the linker resolves multiple matches when processing scatter files on page C6-650

C6.4.8 Behavior when .ANY sections overflow because of linker-generated content on page C6-627

Related tasks

C6.4.3 Prioritizing the placement of unassigned sections on page C6-620

Related references

C1.121 --scatter=filename on page C1-470

C6.4.7 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:

Table C6-3 Input section properties and ordering for sections_a.o and sections_b.o

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 C6-4 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 C6-5 Sort order for cmdline algorithm

Name	Size
`seca_1`	`0x4`
`seca_2`	`0x4`
`seca_3`	`0x10`
`seca_4`	`0x14`
`secb_1`	`0x4`
`secb_2`	`0x4`

Table C6-5 Sort order for cmdline algorithm (continued)

Name	Size
`secb_3`	`0x10`
`secb_4`	`0x14`

That is, the input sections are sorted by command-line index.

Related concepts

C6.4.2 Command-line options for controlling the placement of unassigned sections on page C6-620

C6.4.4 Specify the maximum region size permitted for placing unassigned sections on page C6-621

Related tasks

C6.4.3 Prioritizing the placement of unassigned sections on page C6-620

Related references

C1.3 --any_sort_order=order on page C1-340

C1.121 --scatter=filename on page C1-470

C6.4.8 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:

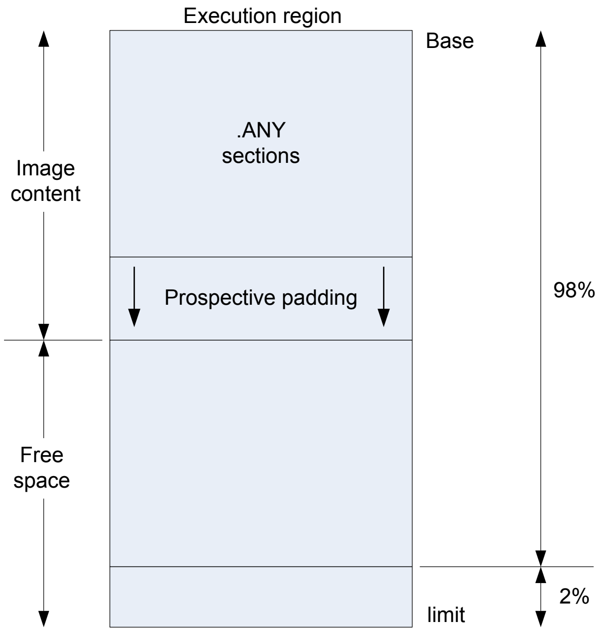


Figure C6-4 .ANY contingency

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.

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
   {
      .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
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 concepts

C6.4.2 Command-line options for controlling the placement of unassigned sections on page C6-620

C6.4.4 Specify the maximum region size permitted for placing unassigned sections on page C6-621

Related tasks

C6.4.3 Prioritizing the placement of unassigned sections on page C6-620

Related references

C1.1 --any_contingency on page C1-337

C1.60 --info=topic[,topic,...] (armlink) on page C1-401

C7.5.2 Syntax of an input section description on page C7-672

C7.4.3 Execution region attributes on page C7-666

C6.5 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

1. 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.

   Note

   There are no state-change veneers in A64.

Related concepts

C3.6 Linker-generated veneers on page C3-548

C6.6 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.

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

...

    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 concepts

C3.6.6 Generation of secure gateway veneers on page C3-551

C6.2.6 Restrictions on placing __at sections on page C6-612

Related tasks

C6.2.5 Placing __at sections at a specific address on page C6-611

C6.2.7 Automatically placing __at sections on page C6-612

C6.2.8 Manually placing __at sections on page C6-614

C6.7 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.

This section contains the following subsections:

- *C6.7.1 Characteristics of a reserved empty block of memory* on page C6-633.
- *C6.7.2 Example of reserving an empty block of memory* on page C6-633.

C6.7.1 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.

C6.7.2 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.

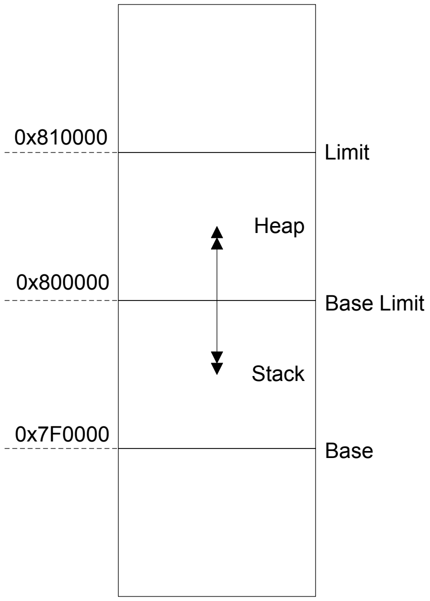


Figure C6-5 Reserving a region for the stack

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.

Related concepts

C7.4 Execution region descriptions on page C7-664

Related references

C5.3.2 Image$$ execution region symbols on page C5-580

C7.4.3 Execution region attributes on page C7-666

C6.8 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.

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.

This section contains the following subsections:

- *C6.8.1 Placing code in a root region* on page C6-635.
- *C6.8.2 Placing Arm® C library code* on page C6-635.
- *C6.8.3 Placing Arm® C++ library code* on page C6-636.

C6.8.1 Placing 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 concepts

C6.2.1 Effect of the ABSOLUTE attribute on a root region on page C6-602

C6.2.2 Effect of the FIXED attribute on a root region on page C6-604

C6.2 Root region and the initial entry point on page C6-602

Related tasks

C6.8.2 Placing Arm® C library code on page C6-635

C6.8.3 Placing Arm® C++ library code on page C6-636

C6.8.2 Placing 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
    {
        * (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 tasks

C6.8.1 Placing code in a root region on page C6-635

C6.8.3 Placing Arm® C++ library code on page C6-636

Related information

C and C++ library naming conventions

C6.8.3 Placing 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.

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
    {
        *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 tasks

C6.8.1 Placing code in a root region on page C6-635

C6.8.2 Placing Arm® C library code on page C6-635

Related information

C and C++ library naming conventions

C6.9 Aligning 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.

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 concepts

C3.4 Linker support for creating demand-paged files on page C3-546

C7.6 Expression evaluation in scatter files on page C7-677

C6.12 Example of using expression evaluation in a scatter file to avoid padding on page C6-642

C7.6.9 Example of aligning a base address in execution space but still tightly packed in load space on page C7-683

Related tasks

C6.10 Aligning execution regions and input sections on page C6-639

Related references

C7.6.6 AlignExpr(expr, align) function on page C7-681

C7.6.7 GetPageSize() function on page C7-682

C1.100 --pagesize=pagesize on page C1-448

C7.3.3 Load region attributes on page C7-659

C7.4.3 Execution region attributes on page C7-666

C1.99 --paged on page C1-447

C6.10 Aligning 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
}
```

Related concepts

C7.5 Input section descriptions on page C7-672

Related tasks

C6.9 Aligning regions to page boundaries on page C6-638

Related references

C7.4.3 Execution region attributes on page C7-666

C6.11 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 `#! armclang --target=arm-arm-none-eabi -march=armv8-a -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
```

This section contains the following subsections:

- *C6.11.1 Default behavior for `armclang` -E in a scatter file* on page C6-640.
- *C6.11.2 Using other preprocessors in a scatter file* on page C6-640.

C6.11.1 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 relative `#includes` 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 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.

C6.11.2 Using 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.

Related concepts

C7.6 Expression evaluation in scatter files on page C7-677

Related references

C1.107 --predefine="string" on page C1-456

C1.121 --scatter=filename on page C1-470

C6.12 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
    {
        …
    }
}
```

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 concepts

C7.6.9 Example of aligning a base address in execution space but still tightly packed in load space on page C7-683

Related references

C7.6.6 AlignExpr(expr, align) function on page C7-681

C7.4.3 Execution region attributes on page C7-666

C6.13 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.

This section contains the following subsections:

- *C6.13.1 Command-line options for creating simple images* on page C6-643.
- *C6.13.2 Type 1 image, one load region and contiguous execution regions* on page C6-643.
- *C6.13.3 Type 2 image, one load region and non-contiguous execution regions* on page C6-645.
- *C6.13.4 Type 3 image, multiple load regions and non-contiguous execution regions* on page C6-646.

C6.13.1 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.

Related concepts

C6.13.2 Type 1 image, one load region and contiguous execution regions on page C6-643

C7.3 Load region descriptions on page C7-658

C6.13.3 Type 2 image, one load region and non-contiguous execution regions on page C6-645

C6.13.4 Type 3 image, multiple load regions and non-contiguous execution regions on page C6-646

Related references

C1.112 --reloc on page C1-461

C1.115 --ro_base=address on page C1-464

C1.116 --ropi on page C1-465

C1.118 --rw_base=address on page C1-467

C1.119 --rwpi on page C1-468

C1.121 --scatter=filename on page C1-470

C1.130 --split on page C1-481

C1.161 --xo_base=address on page C1-512

C7.3.3 Load region attributes on page C7-659

C6.13.2 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.

`--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.
    }
    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. 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.

Note

XO memory is supported only for Armv7-M and Armv8-M architectures.

Related concepts

C6.13.1 Command-line options for creating simple images on page C6-643
C7.3 Load region descriptions on page C7-658
C7.3.6 Considerations when using a relative address +offset for a load region on page C7-662
C7.4.5 Considerations when using a relative address +offset for execution regions on page C7-670

Related references

C1.115 --ro_base=address on page C1-464
C1.116 --ropi on page C1-465
C7.3.3 Load region attributes on page C7-659
C1.112 --reloc on page C1-461

C6.13.3 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 Armv7-M and Armv8-M architectures.

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
{
    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.
    }
}
```

`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 concepts

C7.3 Load region descriptions on page C7-658

C7.3.6 Considerations when using a relative address +offset for a load region on page C7-662

C7.4.5 Considerations when using a relative address +offset for execution regions on page C7-670

Related references

C1.115 --ro_base=address on page C1-464

C1.118 --rw_base=address on page C1-467

C1.161 --xo_base=address on page C1-512

C7.3.3 Load region attributes on page C7-659

C6.13.4 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.

Note

XO memory is supported only for Armv7-M and Armv8-M architectures.

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.
{
    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)
     }
}
```

Related concepts

C7.3 Load region descriptions on page C7-658
C7.3.6 Considerations when using a relative address +offset for a load region on page C7-662
C7.4.5 Considerations when using a relative address +offset for execution regions on page C7-670
C7.3.4 Inheritance rules for load region address attributes on page C7-661
C7.3.5 Inheritance rules for the RELOC address attribute on page C7-662
C7.4.4 Inheritance rules for execution region address attributes on page C7-669

Related references

C1.112 --reloc on page C1-461
C1.115 --ro_base=address on page C1-464
C1.118 --rw_base=address on page C1-467
C1.130 --split on page C1-481
C1.161 --xo_base=address on page C1-512
C7.3.3 Load region attributes on page C7-659

C6.14 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.
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.
- 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`*s:
  - `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.

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 concepts

C7.5 Input section descriptions on page C7-672

Related references

C6.4 Placement of unassigned sections on page C6-619

C7.2 Syntax of a scatter file on page C7-657

C7.5.2 Syntax of an input section description on page C7-672

C6.15 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 references

C7.2 Syntax of a scatter file on page C7-657

C6.16 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.

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)
…
```

Related concepts

C6.1.1 Overview of scatter-loading on page C6-596

C6.1.6 Scatter-loading images with a simple memory map on page C6-599

Chapter C7
Scatter File Syntax

Describes the format of scatter files.

It contains the following sections:

- *C7.1 BNF notation used in scatter-loading description syntax* on page C7-656.
- *C7.2 Syntax of a scatter file* on page C7-657.
- *C7.3 Load region descriptions* on page C7-658.
- *C7.4 Execution region descriptions* on page C7-664.
- *C7.5 Input section descriptions* on page C7-672.
- *C7.6 Expression evaluation in scatter files* on page C7-677.

C7.1 BNF notation used in scatter-loading description syntax

Scatter-loading description syntax uses standard BNF notation.

The following table summarizes the *Backus-Naur Form* (BNF) symbols that are used for describing the syntax of scatter-loading descriptions.

Table C7-1 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 references

C7.2 Syntax of a scatter file on page C7-657

C7.2 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:

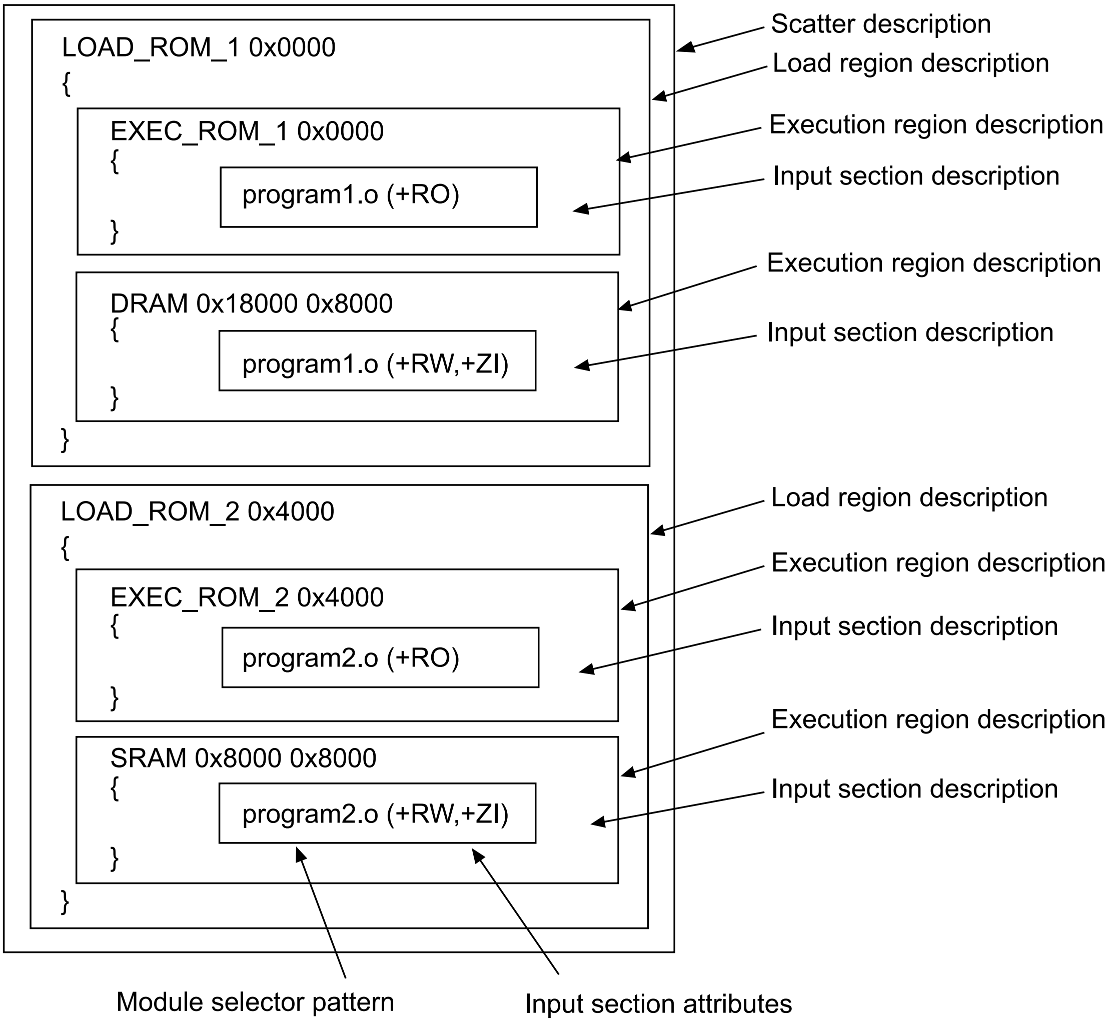


Figure C7-1 Components of a scatter file

Related concepts

C7.3 Load region descriptions on page C7-658

C7.4 Execution region descriptions on page C7-664

Related references

Chapter C6 Scatter-loading Features on page C6-595

C7.3 Load region descriptions

A load region description specifies the region of memory where its child execution regions are to be placed.

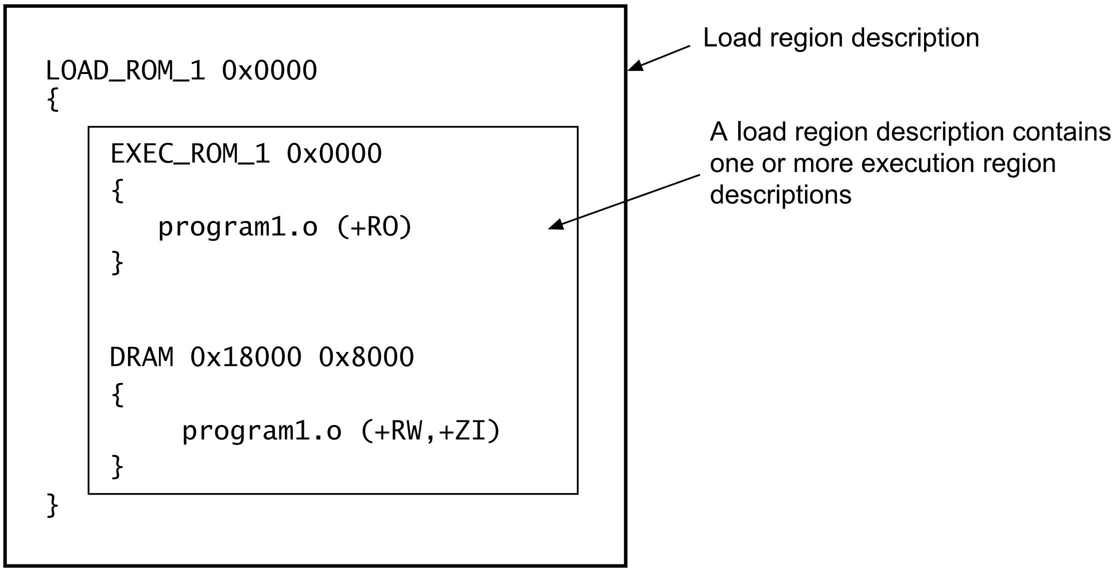
This section contains the following subsections:
- *C7.3.1 Components of a load region description* on page C7-658.
- *C7.3.2 Syntax of a load region description* on page C7-659.
- *C7.3.3 Load region attributes* on page C7-659.
- *C7.3.4 Inheritance rules for load region address attributes* on page C7-661.
- *C7.3.5 Inheritance rules for the RELOC address attribute* on page C7-662.
- *C7.3.6 Considerations when using a relative address +offset for a load region* on page C7-662.

C7.3.1 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 (used by the linker to identify different load regions).
- A base address (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:



```
LOAD_ROM_1 0x0000
{
    EXEC_ROM_1 0x0000
    {
       program1.o (+RO)
    }

    DRAM 0x18000 0x8000
    {
         program1.o (+RW,+ZI)
    }
}
```

Load region description

A load region description contains one or more execution region descriptions

Figure C7-2 Components of a load region description

Related concepts

C7.3.4 Inheritance rules for load region address attributes on page C7-661

C7.3.5 Inheritance rules for the RELOC address attribute on page C7-662

C7.4.4 Inheritance rules for execution region address attributes on page C7-669

C7.6 Expression evaluation in scatter files on page C7-677

Related tasks

C6.9 Aligning regions to page boundaries on page C6-638

Related references

C7.3.2 Syntax of a load region description on page C7-659

C7.3.3 Load region attributes on page C7-659
Chapter C6 Scatter-loading Features on page C6-595

C7.3.2 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:

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 concepts

C7.3.5 Inheritance rules for the RELOC address attribute on page C7-662
C7.3.6 Considerations when using a relative address +offset for a load region on page C7-662
C7.3.4 Inheritance rules for load region address attributes on page C7-661
C7.6 Expression evaluation in scatter files on page C7-677

Related references

C7.3.1 Components of a load region description on page C7-658
C7.3.3 Load region attributes on page C7-659
C7.1 BNF notation used in scatter-loading description syntax on page C7-656
C7.2 Syntax of a scatter file on page C7-657
C5.3 Region-related symbols on page C5-580

C7.3.3 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`*

Increase the alignment constraint for the load region from 4 to *`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.

`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.

Note

`PI` is not supported for AArch64 state.

Note

This attribute is not supported if an image contains execute-only sections.

`PROTECTED`

The `PROTECTED` keyword prevents:

- Overlapping of load regions.
- Veneer sharing.
- String sharing with the `--merge` option.

`RELOC`

Note

- This attribute is deprecated when *Base Platform* on page C9-705 is not enabled.
- `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 concepts

C7.6.9 Example of aligning a base address in execution space but still tightly packed in load space on page C7-683

C3.3.3 Section alignment with the linker on page C3-545

C3.6.5 Reuse of veneers when scatter-loading on page C3-550

C7.3.6 Considerations when using a relative address +offset for a load region on page C7-662
C7.3.4 Inheritance rules for load region address attributes on page C7-661
C7.3.5 Inheritance rules for the RELOC address attribute on page C7-662
C3.6.2 Veneer sharing on page C3-548
C3.6.4 Generation of position independent to absolute veneers on page C3-550
C4.3 Optimization with RW data compression on page C4-564

Related tasks

C6.9 Aligning regions to page boundaries on page C6-638

Related references

C1.90 --merge, --no_merge on page C1-438
C7.3.1 Components of a load region description on page C7-658
C7.3.2 Syntax of a load region description on page C7-659

C7.3.4 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
{
```

```
    …
}
```

Related concepts

C7.4.4 Inheritance rules for execution region address attributes on page C7-669
C7.3.6 Considerations when using a relative address +offset for a load region on page C7-662
C7.3.5 Inheritance rules for the RELOC address attribute on page C7-662

Related references

C7.3.1 Components of a load region description on page C7-658
C7.4.1 Components of an execution region description on page C7-664
C7.3.2 Syntax of a load region description on page C7-659

C7.3.5 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 concepts

C9.1 Restrictions on the use of scatter files with the Base Platform model on page C9-706
C7.3.4 Inheritance rules for load region address attributes on page C7-661
C7.4.4 Inheritance rules for execution region address attributes on page C7-669
C7.4.5 Considerations when using a relative address +offset for execution regions on page C7-670
C7.3.6 Considerations when using a relative address +offset for a load region on page C7-662
C2.5 Base Platform linking model overview on page C2-522

Related references

C7.3.1 Components of a load region description on page C7-658
C7.3.2 Syntax of a load region description on page C7-659
C7.4.1 Components of an execution region description on page C7-664

C7.3.6 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:

- If the `+offset` load region LR2 follows a load region LR1 containing ZI data, then LR2 overlaps the ZI data. To fix this, use the `ImageLimit()` function to specify the base address of LR2.
- 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`.
- 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 is because the linker calculates the `+offset` based on the uncompressed size of the preceding region. However, this gap disappears when the RW data is decompressed at load time.

Related concepts

C7.3.4 Inheritance rules for load region address attributes on page C7-661

C7.6.3 Execution address built-in functions for use in scatter files on page C7-678

Related references

C7.2 Syntax of a scatter file on page C7-657

C7.4 Execution region descriptions

An execution region description specifies the region of memory where parts of your image are to be placed at run-time.

This section contains the following subsections:

- *C7.4.1 Components of an execution region description* on page C7-664.
- *C7.4.2 Syntax of an execution region description* on page C7-664.
- *C7.4.3 Execution region attributes* on page C7-666.
- *C7.4.4 Inheritance rules for execution region address attributes* on page C7-669.
- *C7.4.5 Considerations when using a relative address +offset for execution regions* on page C7-670.

C7.4.1 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 (used by the linker to identify different execution regions).
- A base address (either absolute or relative).
- Attributes that specify the properties of the execution region.
- An optional maximum size specification.
- One or more input section descriptions (the modules placed into this execution region).

The following figure shows the components of a typical execution region description:

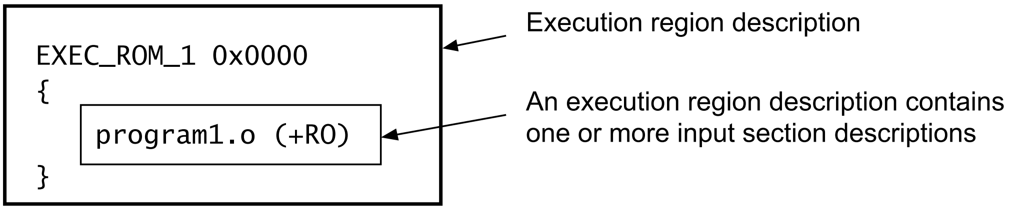


Figure C7-3 Components of an execution region description

Related concepts

C7.3.4 Inheritance rules for load region address attributes on page C7-661

C7.3.5 Inheritance rules for the RELOC address attribute on page C7-662

C7.4.4 Inheritance rules for execution region address attributes on page C7-669

C7.6 Expression evaluation in scatter files on page C7-677

C7.5 Input section descriptions on page C7-672

Related tasks

C6.9 Aligning regions to page boundaries on page C6-638

Related references

C7.4.2 Syntax of an execution region description on page C7-664

C7.4.3 Execution region attributes on page C7-666

Chapter C6 Scatter-loading Features on page C6-595

C7.3.3 Load region attributes on page C7-659

C7.4.2 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.

base_address

Specifies the address where objects in the region are to be linked. *`base_address`* must be word-aligned.

Note

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.

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 concepts

C7.4.5 Considerations when using a relative address +offset for execution regions on page C7-670
C7.6 Expression evaluation in scatter files on page C7-677
C2.5 Base Platform linking model overview on page C2-522
C9.1 Restrictions on the use of scatter files with the Base Platform model on page C9-706
C7.3.4 Inheritance rules for load region address attributes on page C7-661
C7.3.5 Inheritance rules for the RELOC address attribute on page C7-662
C7.5 Input section descriptions on page C7-672

Related tasks

C6.9 Aligning regions to page boundaries on page C6-638

Related references

C7.4.1 Components of an execution region description on page C7-664
C7.4.3 Execution region attributes on page C7-666
Chapter C6 Scatter-loading Features on page C6-595

C5.3 Region-related symbols on page C5-580

C7.4.3 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.

Note

`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()`.

Note

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.

`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.

`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 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.

`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.

`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` full of 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.

Note

`PI` is not supported for AArch64 state.

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 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.

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 concepts

C6.4.8 Behavior when .ANY sections overflow because of linker-generated content on page C6-627

C3.3.3 Section alignment with the linker on page C3-545

C6.12 Example of using expression evaluation in a scatter file to avoid padding on page C6-642

C7.6.9 Example of aligning a base address in execution space but still tightly packed in load space on page C7-683

C7.4.5 Considerations when using a relative address +offset for execution regions on page C7-670

C7.6 Expression evaluation in scatter files on page C7-677

C4.3 Optimization with RW data compression on page C4-564

C7.4.4 Inheritance rules for execution region address attributes on page C7-669

Related tasks

C6.9 Aligning regions to page boundaries on page C6-638

C6.10 Aligning execution regions and input sections on page C6-639

Related references

C7.4.2 Syntax of an execution region description on page C7-664

C5.3.3 Load$$ execution region symbols on page C5-581

C7.6.6 AlignExpr(expr, align) function on page C7-681

C7.1 BNF notation used in scatter-loading description syntax on page C7-656

C1.1 --any_contingency on page C1-337

C5.3.2 Image$$ execution region symbols on page C5-580

C7.5.2 Syntax of an input section description on page C7-672

C1.95 --overlay_veneers on page C1-443

C1.129 --sort=algorithm on page C1-479

Related information

Overlay support in Arm Compiler

C7.4.4 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.

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 concepts

C7.3.6 Considerations when using a relative address +offset for a load region on page C7-662

C7.3.4 Inheritance rules for load region address attributes on page C7-661

C7.4.5 Considerations when using a relative address +offset for execution regions on page C7-670

Related references

C7.3.1 Components of a load region description on page C7-658

C7.4.1 Components of an execution region description on page C7-664

C7.4.2 Syntax of an execution region description on page C7-664

C7.4.5 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 concepts

C7.4.4 Inheritance rules for execution region address attributes on page C7-669

C7.3.5 Inheritance rules for the RELOC address attribute on page C7-662

Related references

C7.2 Syntax of a scatter file on page C7-657

C7.5 Input section descriptions

An input section description is a pattern that identifies input sections.

This section contains the following subsections:
- *C7.5.1 Components of an input section description* on page C7-672.
- *C7.5.2 Syntax of an input section description* on page C7-672.
- *C7.5.3 Examples of module and input section specifications* on page C7-675.

C7.5.1 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.

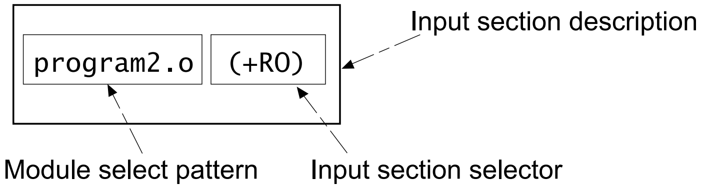


Figure C7-4 Components of an input section description

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 references

C7.5.2 Syntax of an input section description on page C7-672
C7.2 Syntax of a scatter file on page C7-657
C1.101 --partial on page C1-449

C7.5.2 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
```

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 leading path name).
- The full name of the library (including path name) the section is extracted from. 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`.
- `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.

——— **Caution** ———

`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.

——— **Note** ———

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.

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.

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.

  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. If, for example, 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.

Related concepts

C6.4.8 Behavior when .ANY sections overflow because of linker-generated content on page C6-627

C7.5.3 Examples of module and input section specifications on page C7-675

C6.4.5 Examples of using placement algorithms for .ANY sections on page C6-622

C6.4.6 Example of next_fit algorithm showing behavior of full regions, selectors, and priority on page C6-624

C6.4.7 Examples of using sorting algorithms for .ANY sections on page C6-625

Related tasks

C6.10 Aligning execution regions and input sections on page C6-639

Related references

C7.5.1 Components of an input section description on page C7-672

C7.1 BNF notation used in scatter-loading description syntax on page C7-656

C7.2 Syntax of a scatter file on page C7-657

C6.4 Placement of unassigned sections on page C6-619

C7.5.3 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 references

C7.5.1 Components of an input section description on page C7-672

C7.5.2 Syntax of an input section description on page C7-672

C7.6 Expression evaluation in scatter files

Scatter files frequently contain numeric constants. These can be specific values, or the result of an expression.

This section contains the following subsections:

- *C7.6.1 Expression usage in scatter files* on page C7-677.
- *C7.6.2 Expression rules in scatter files* on page C7-678.
- *C7.6.3 Execution address built-in functions for use in scatter files* on page C7-678.
- *C7.6.4 ScatterAssert function and load address related functions* on page C7-680.
- *C7.6.5 Symbol related function in a scatter file* on page C7-681.
- *C7.6.6 AlignExpr(expr, align) function* on page C7-681.
- *C7.6.7 GetPageSize() function* on page C7-682.
- *C7.6.8 SizeOfHeaders() function* on page C7-682.
- *C7.6.9 Example of aligning a base address in execution space but still tightly packed in load space* on page C7-683.
- *C7.6.10 Scatter files containing relative base address load regions and a ZI execution region* on page C7-683.

C7.6.1 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.

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 concepts

C7.6.2 Expression rules in scatter files on page C7-678
C7.6.3 Execution address built-in functions for use in scatter files on page C7-678
C7.6.4 ScatterAssert function and load address related functions on page C7-680
C7.6.5 Symbol related function in a scatter file on page C7-681
C7.3.6 Considerations when using a relative address +offset for a load region on page C7-662
C7.4.5 Considerations when using a relative address +offset for execution regions on page C7-670
C7.6.9 Example of aligning a base address in execution space but still tightly packed in load space on page C7-683

Related references

C7.2 Syntax of a scatter file on page C7-657
C7.3.2 Syntax of a load region description on page C7-659
C7.4.2 Syntax of an execution region description on page C7-664

C7.6.2 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.

  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 concepts

C7.6.1 Expression usage in scatter files on page C7-677
C7.6.3 Execution address built-in functions for use in scatter files on page C7-678
C7.6.4 ScatterAssert function and load address related functions on page C7-680
C7.6.5 Symbol related function in a scatter file on page C7-681
C7.3.6 Considerations when using a relative address +offset for a load region on page C7-662
C7.4.5 Considerations when using a relative address +offset for execution regions on page C7-670
C7.6.9 Example of aligning a base address in execution space but still tightly packed in load space on page C7-683

Related references

C7.2 Syntax of a scatter file on page C7-657
C7.3.2 Syntax of a load region description on page C7-659
C7.4.2 Syntax of an execution region description on page C7-664

C7.6.3 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.

Table C7-2 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`.

Related concepts

C7.6.1 Expression usage in scatter files on page C7-677
C7.6.2 Expression rules in scatter files on page C7-678
C7.6.4 ScatterAssert function and load address related functions on page C7-680
C7.6.5 Symbol related function in a scatter file on page C7-681
C7.3.6 Considerations when using a relative address +offset for a load region on page C7-662

C7.6.10 Scatter files containing relative base address load regions and a ZI execution region on page C7-683
C7.4.5 Considerations when using a relative address +offset for execution regions on page C7-670
C7.6.9 Example of aligning a base address in execution space but still tightly packed in load space on page C7-683

Related references

C7.2 Syntax of a scatter file on page C7-657
C7.3.2 Syntax of a load region description on page C7-659
C7.4.2 Syntax of an execution region description on page C7-664
C7.6.6 AlignExpr(expr, align) function on page C7-681
C5.3.2 Image$$ execution region symbols on page C5-580

C7.6.4 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 C7-3 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)
    }
    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 concepts

C7.6.1 Expression usage in scatter files on page C7-677

C7.6.2 Expression rules in scatter files on page C7-678
C7.6.3 Execution address built-in functions for use in scatter files on page C7-678
C7.6.5 Symbol related function in a scatter file on page C7-681
C7.6.9 Example of aligning a base address in execution space but still tightly packed in load space on page C7-683

Related references

C7.2 Syntax of a scatter file on page C7-657
C7.3.2 Syntax of a load region description on page C7-659
C7.4.2 Syntax of an execution region description on page C7-664
C5.3.3 Load$$ execution region symbols on page C5-581

C7.6.5 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

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)
    }
}
```

Related concepts

C7.6.1 Expression usage in scatter files on page C7-677
C7.6.2 Expression rules in scatter files on page C7-678
C7.6.3 Execution address built-in functions for use in scatter files on page C7-678
C7.6.4 ScatterAssert function and load address related functions on page C7-680
C7.6.9 Example of aligning a base address in execution space but still tightly packed in load space on page C7-683

Related references

C7.2 Syntax of a scatter file on page C7-657
C7.3.2 Syntax of a load region description on page C7-659
C7.4.2 Syntax of an execution region description on page C7-664

C7.6.6 AlignExpr(expr, align) function

Aligns an address expression to a specified boundary.

This function returns:

`(expr + (align-1)) & ~(align-1))`

Where:

- `expr` is a valid address expression.
- `align` is the alignment, and must be a positive power of 2.

It increases `expr` until:

`expr ≡ 0 (mod align)`

Example

This example aligns the address of `ER2` on an 8-byte boundary:

```
ER +0
{
    …
}
ER2 AlignExpr(+0x8000,8)
{
    …
}
```

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.
- 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.

Related references

C7.4.3 Execution region attributes on page C7-666

C7.6.7 GetPageSize() function

Returns the page size when an image is demand paged, and is useful when used with the `AlignExpr` function.

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.

By default the internal page size is set to `8000`, but you can change it with the `--pagesize` command-line option.

Example

This example aligns the base address of `ER` to a Page Boundary:

```
ER AlignExpr(+0, GetPageSize())
{
    …
}
```

Related concepts

C7.6.9 Example of aligning a base address in execution space but still tightly packed in load space on page C7-683

Related references

C1.100 --pagesize=pagesize on page C1-448

C7.6.6 AlignExpr(expr, align) function on page C7-681

C7.6.8 SizeOfHeaders() function

Returns the size of ELF header plus the estimated size of the Program Header table.

This is useful when writing demand paged images to start code and data immediately after the ELF header and Program Header table.

Example

This example sets the base of `LR1` to start immediately after the ELF header and Program Headers:

```
LR1 SizeOfHeaders()
{
    …
}
```

Related concepts

C7.6.9 Example of aligning a base address in execution space but still tightly packed in load space on page C7-683

C3.4 Linker support for creating demand-paged files on page C3-546

Related tasks

C6.9 Aligning regions to page boundaries on page C6-638

C7.6.9 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 references

C7.3.3 Load region attributes on page C7-659

C7.4.3 Execution region attributes on page C7-666

C7.6.7 GetPageSize() function on page C7-682

C7.6.8 SizeOfHeaders() function on page C7-682

C7.3.2 Syntax of a load region description on page C7-659

C7.4.2 Syntax of an execution region description on page C7-664

C7.6.6 AlignExpr(expr, align) function on page C7-681

C7.6.10 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
{
    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 concepts

C7.6 Expression evaluation in scatter files on page C7-677
C7.6.1 Expression usage in scatter files on page C7-677
C7.6.2 Expression rules in scatter files on page C7-678
C7.6.3 Execution address built-in functions for use in scatter files on page C7-678

Related references

C7.2 Syntax of a scatter file on page C7-657
C7.3.2 Syntax of a load region description on page C7-659
C7.4.2 Syntax of an execution region description on page C7-664
C5.3.2 Image$$ execution region symbols on page C5-580

Chapter C8
BPABI and SysV Shared Libraries and Executables

Describes how the Arm linker, `armlink`, supports the *Base Platform Application Binary Interface* (BPABI) and *System V* (SysV) shared libraries and executables.

It contains the following sections:

- *C8.1 About the Base Platform Application Binary Interface (BPABI)* on page C8-686.
- *C8.2 Platforms supported by the BPABI* on page C8-687.
- *C8.3 Features common to all BPABI models* on page C8-688.
- *C8.4 SysV linking model* on page C8-691.
- *C8.5 Bare metal and DLL-like memory models* on page C8-697.
- *C8.6 Symbol versioning* on page C8-702.

C8.1 About the Base Platform Application Binary Interface (BPABI)

The *Base Platform Application Binary Interface* (BPABI) is a meta-standard for third parties to generate their own platform-specific image formats.

Many embedded systems use an *operating system* (OS) to manage the resources on a device. In many cases this is a large, single executable with a *Real-Time Operating System* (RTOS) that tightly integrates with the applications.

To run an application or use a shared library on a platform OS, you must conform to the *Application Binary Interface* (ABI) for the platform and also the ABI for the Arm architecture. This can involve substantial changes to the linker output, for example, a custom file format. To support such a wide variety of platforms, the ABI for the Arm architecture provides the BPABI.

The BPABI provides a base standard from which a platform ABI can be derived. The linker produces a BPABI conforming ELF image or shared library. A platform specific tool called a post-linker translates this ELF output file into a platform-specific file format. Post linker tools are provided by the platform OS vendor. The following figure shows the BPABI tool flow.

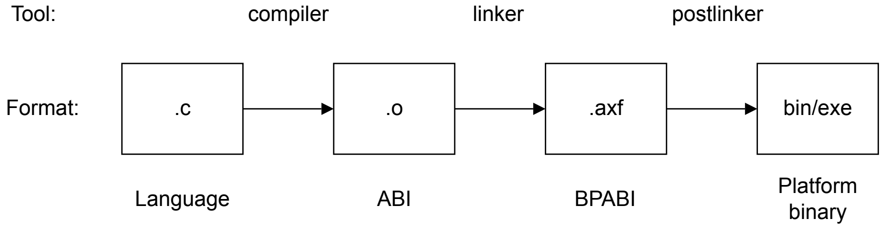


Figure C8-1 BPABI tool flow

Related concepts

C8.2 Platforms supported by the BPABI on page C8-687

Related information

Base Platform ABI for the Arm Architecture

AN242 Dynamic Linking with the Arm Compiler toolchain

C8.2 Platforms supported by the BPABI

The *Base Platform Application Binary Interface* (BPABI) defines different platform models based on the type of shared library.

The platform models are:

Bare metal

The bare metal model is designed for an offline dynamic loader or a simple module loader. References between modules are resolved by the loader directly without any additional support structures.

DLL-like

The *dynamically linked library* (DLL) like model sacrifices transparency between the dynamic and static library in return for better load and run-time efficiency.

Note

The DLL-like model is not supported for AArch64 state.

Linker support for the BPABI

The Arm linker supports all three BPABI models enabling you to link a collection of objects and libraries into a:

- Bare metal executable image.
- BPABI DLL shared object.
- BPABI executable file.

Related concepts

C8.1 About the Base Platform Application Binary Interface (BPABI) on page C8-686

Related references

C1.32 --dll on page C1-373

C8.3 Features common to all BPABI models

Some features are common to all BPABI models.

The linker enables you to build *Base Platform Application Binary Interface* (BPABI) shared libraries and to link objects against shared libraries. The following features are common to all BPABI models:

- Symbol importing.
- Symbol exporting.
- Versioning.
- Visibility of symbols.

This section contains the following subsections:

- *C8.3.1 About importing and exporting symbols for BPABI models* on page C8-688.
- *C8.3.2 Symbol visibility for BPABI models* on page C8-688.
- *C8.3.3 Automatic import and export for BPABI models* on page C8-689.
- *C8.3.4 Manual import and export for BPABI models* on page C8-689.
- *C8.3.5 Symbol versioning for BPABI models* on page C8-690.
- *C8.3.6 RW compression for BPABI models* on page C8-690.

C8.3.1 About importing and exporting symbols for BPABI models

How symbols are imported and exported depends on the platform model.

In traditional linking, all symbols must be defined at link time for linking into a single executable file containing all the required code and data. In platforms that support dynamic linking, symbol binding can be delayed to load-time or in some cases, run-time. Therefore, the application can be split into a number of modules, where a module is either an executable or a shared library. Any symbols that are defined in modules other than the current module are placed in the dynamic symbol table. Any functions that are suitable for dynamically linking to at load or runtime are also listed in the dynamic symbol table.

There are two ways to control the contents of the dynamic symbol table:

- Automatic rules that infer the contents from the ELF symbol visibility property.
- Manual directives that are present in a steering file.

Related concepts

C8.3.3 Automatic import and export for BPABI models on page C8-689

C8.3.1 About importing and exporting symbols for BPABI models on page C8-688

C8.3.2 Symbol visibility for BPABI models on page C8-688

C8.3.4 Manual import and export for BPABI models on page C8-689

C8.3.5 Symbol versioning for BPABI models on page C8-690

C8.3.6 RW compression for BPABI models on page C8-690

C8.6.3 The symbol versioning script file on page C8-703

Related references

C8.5.3 Linker command-line options for bare metal and DLL-like models on page C8-698

C8.3.2 Symbol visibility for BPABI models

For *Base Platform Application Binary Interface* (BPABI) models, each symbol has a visibility property that can be controlled by compiler switches, a steering file, or attributes in the source code.

If a symbol is a reference, the visibility controls the definitions that the linker can use to define the symbol.

If a symbol is a definition, the visibility controls whether the symbol can be made visible outside the current module.

The visibility options defined by the ELF specification are:

Table C8-1 Symbol visibility

Visibility	Reference	Definition
`STV_DEFAULT`	Symbol can be bound to a definition in a shared object.	Symbol can be made visible outside the module. It can be preempted by the dynamic linker by a definition from another module.
`STV_PROTECTED`	Symbol must be resolved within the module.	Symbol can be made visible outside the module. It cannot be preempted at run-time by a definition from another module.
`STV_HIDDEN` `STV_INTERNAL`	Symbol must be resolved within the module.	Symbol is not visible outside the module.

Symbol preemption can happen in *dynamically linked library* (DLL) like implementations of the BPABI. The platform owner defines how this works. See the documentation for your specific platform for more information.

Related concepts

C4.3 Optimization with RW data compression on page C4-564

C8.6.3 The symbol versioning script file on page C8-703

Related references

C8.5.3 Linker command-line options for bare metal and DLL-like models on page C8-698

C1.89 --max_visibility=type on page C1-437

C1.96 --override_visibility on page C1-444

C10.1 EXPORT steering file command on page C10-712

C10.3 IMPORT steering file command on page C10-714

C10.5 REQUIRE steering file command on page C10-716

C1.151 --use_definition_visibility on page C1-502

F6.28 EXPORT or GLOBAL on page F6-1051

C8.3.3 Automatic import and export for BPABI models

The linker can automatically import and export symbols for BPABI models.

This behavior depends on a combination of the symbol visibility in the input object file, if the output is an executable or a shared library. This depends on what type of linking model is being used.

Related concepts

C8.3 Features common to all BPABI models on page C8-688

C8.6 Symbol versioning on page C8-702

Related references

C8.5.3 Linker command-line options for bare metal and DLL-like models on page C8-698

C8.3.4 Manual import and export for BPABI models

You can directly control the import and export of symbols with a linker steering file.

You can use linker steering files to:

- Manually control dynamic import and export.
- Override the automatic rules.

The steering file commands available to control the dynamic symbol table contents are:

- `EXPORT`.
- `IMPORT`.
- `REQUIRE`.

Related concepts

C5.6 Edit the symbol tables with a steering file on page C5-590

Related references

C10.1 EXPORT steering file command on page C10-712

C10.3 IMPORT steering file command on page C10-714

C10.5 REQUIRE steering file command on page C10-716

C8.3.5 Symbol versioning for BPABI models

Symbol versioning provides a way to tightly control the interface of a shared library.

When a symbol is imported from a shared library that has versioned symbols, `armlink` binds to the most recent (default) version of the symbol. At load or run-time when the platform OS resolves the symbol version, it always resolves to the version selected by `armlink`, even if there is a more recent version available. This process is automatic.

When a symbol is exported from an executable or a shared library, it can be given a version. `armlink` supports explicit symbol versioning where you use a script to precisely define the versions.

Related concepts

C8.6 Symbol versioning on page C8-702

C8.3.6 RW compression for BPABI models

The decompressor for compressed RW data is tightly integrated into the start-up code in the Arm C library.

When running an application on a platform OS, this functionality must be provided by the platform or platform libraries. Therefore, RW compression is turned off when linking a *Base Platform Application Binary Interface* (BPABI) file because there is no decompressor. It is not possible to turn compression back on again.

Related concepts

C4.3 Optimization with RW data compression on page C4-564

C8.4 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.

This section contains the following subsections:

- *C8.4.1 SysV standard memory model* on page C8-691.
- *C8.4.2 Using the C and C++ libraries* on page C8-692.
- *C8.4.3 Using a dynamic Linker* on page C8-693.
- *C8.4.4 Automatic dynamic symbol table rules in the SysV linking model* on page C8-694.
- *C8.4.5 Symbol definitions defined for SysV compatibility with glibc* on page C8-694.
- *C8.4.6 Addressing modes in the SysV linking model* on page C8-695.
- *C8.4.7 Thread local storage in the SysV linking model* on page C8-696.
- *C8.4.8 Linker command-line options for the SysV linking model* on page C8-696.

C8.4.1 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
    {
        *(.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 AlignExpr(+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)
    }
    .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.

C8.4.2 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 *C3.10 How the linker searches for the Arm® standard libraries* on page C3-556, 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 (`.prenit_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);
  ```

  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.

The code of the program must be compiled with the `-nostdlib` and `-nostdlibinc` `armclang` command-line options to indicate to the compiler to not use the Arm C and C++ libraries.

Executable and shared objects should be linked with the `--no_scanlib` `armlink` command-line option.

C8.4.3 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.

C8.4.4 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:

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 concepts

C8.4.6 Addressing modes in the SysV linking model on page C8-695

C8.4.7 Thread local storage in the SysV linking model on page C8-696

Related references

C8.4.8 Linker command-line options for the SysV linking model on page C8-696

C1.44 --export_all, --no_export_all on page C1-385

C1.45 --export_dynamic, --no_export_dynamic on page C1-386

Related information

ELF for the Arm Architecture

C8.4.5 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`.
- 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 concepts

C8.4 SysV linking model on page C8-691

Related information

ELF for the Arm Architecture

C8.4.6 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 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.

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 concepts

C8.4.4 Automatic dynamic symbol table rules in the SysV linking model on page C8-694

C8.4.7 Thread local storage in the SysV linking model on page C8-696

Related references

C8.4.8 Linker command-line options for the SysV linking model on page C8-696

C1.30 --diag_suppress=tag[,tag,…] (armlink) on page C1-371

C1.51 --fpic on page C1-392

C1.59 --import_unresolved, --no_import_unresolved on page C1-400

C1.123 --shared on page C1-473

Related information

--apcs=qualifier...qualifier compiler option

C8.4.7 Thread local storage in the SysV linking model

`armlink` supports the traditional Arm Linux thread local storage (TLS) model, in the AArch32 state.

The *Addenda to*, and *Errata in*, the ABI for the Arm Architecture describes the Arm Linux thread local storage (TLS) model.

Note

`armlink` does not support the newer TLS descriptor model. The *Application Binary Interface* (ABI) *ELF for the Arm® Architecture* describes the *New experimental TLS relocations* used by this model.

Note

`armlink` only supports TLS in the AArch32 state, and not in the AArch64 state.

Related concepts

C8.4 SysV linking model on page C8-691

Related information

Addenda to, and Errata in, the ABI for the Arm Architecture (ABI-addenda)

C8.4.8 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:

- `--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`.

Related references

Chapter C1 armlink Command-line Options on page C1-333

C8.5 Bare metal and DLL-like memory models

If you are developing applications or DLLs for a specific platform OS that are based around the BPABI, there are some features that you must be aware of.

You must use the following information in conjunction with the platform documentation:

- BPABI standard memory model.
- Mandatory symbol versioning in the BPABI DLL-like model.
- Automatic dynamic symbol table rules in the BPABI DLL-like model.
- Addressing modes in the BPABI DLL-like model.
- C++ initialization in the BPABI DLL-like model.

If you are implementing a platform OS, you must use this information in conjunction with the BPABI specification.

Note

The DLL-like model is not supported for AArch64 state.

This section contains the following subsections:
- *C8.5.1 BPABI standard memory model* on page C8-697.
- *C8.5.2 Customization of the BPABI standard memory model* on page C8-698.
- *C8.5.3 Linker command-line options for bare metal and DLL-like models* on page C8-698.
- *C8.5.4 Mandatory symbol versioning in the BPABI DLL-like model* on page C8-699.
- *C8.5.5 Automatic dynamic symbol table rules in the BPABI DLL-like model* on page C8-700.
- *C8.5.6 Addressing modes in the BPABI DLL-like model* on page C8-700.
- *C8.5.7 C++ initialization in the BPABI DLL-like model* on page C8-701.

C8.5.1 BPABI standard memory model

Base Platform Application Binary Interface (BPABI) files have a standard memory model that is described in the BPABI specification.

When you use the `--bpabi` command-line option, the linker automatically applies the standard memory model and ignores any scatter file that you specify on the command-line. This is equivalent to the following image layout:

```
LR_1 <read-only base address>
{
    ER_RO  +0
    {
        *(+RO)
    }
}
LR_2 <read-write base address>
{
    ER_RW  +0
    {
        *(+RW)
    }
    ER_ZI  +0
    {
        *(+ZI)
    }
}
```

The BPABI model is also referred to as the bare metal and DLL-like memory model.

Note

The DLL-like model is not supported for AArch64 state.

Related concepts

C8.5.2 Customization of the BPABI standard memory model on page C8-698

C8.5.2 Customization of the BPABI standard memory model

You can customize the BPABI standard memory model with the memory map related command-line options.

Note

If you specify the option `--ropi`, `LR_1` is marked as position-independent. Likewise, if you specify the option `--rwpi`, `LR_2` is marked as position-independent.

Note

In most cases, you must specify the `--ro_base` and `--rw_base` switches, because the default values, `0x8000` and `0` respectively, might not be suitable for your platform. These addresses do not have to reflect the addresses to which the image is relocated at run time.

If you require a more complicated memory layout, use the Base Platform linking model, `--base_platform`.

Related concepts

C2.5 Base Platform linking model overview on page C2-522

Related references

C1.11 --bpabi on page C1-349
C1.7 --base_platform on page C1-344
C1.115 --ro_base=address on page C1-464
C1.116 --ropi on page C1-465
C1.117 --rosplit on page C1-466
C1.118 --rw_base=address on page C1-467
C1.119 --rwpi on page C1-468
C1.161 --xo_base=address on page C1-512

C8.5.3 Linker command-line options for bare metal and DLL-like models

There are linker command-line options available for building bare metal executables and *dynamically linked library* (DLL) like models for a platform OS.

The command-line options are:

Table C8-2 Turning on BPABI support

Command-line options	Description
`--base_platform`	To use scatter-loading with *Base Platform Application Binary Interface* (BPABI).
`--bpabi`	To produce a BPABI executable.
`--bpabi --dll`	To produce a BPABI DLL.

Note

The DLL-like model is not supported for AArch64 state.

Additional linker command-line options for the BPABI DLL-like model

There are additional linker command-line options available for the BPABI DLL-like model.

The additional command-line options are:

- `--export_all`, `--no_export_all`.
- `--pltgot=type`.
- `--pltgot_opts=`*`mode`*.
- `--ro_base=address`.
- `--ropi`.
- `--rosplit`.
- `--rw_base=`*`address`*.
- `--rwpi`.
- `--symver_script=`*`filename`*.
- `--symver_soname`.

Related concepts

C8.5.1 BPABI standard memory model on page C8-697

C8.5.5 Automatic dynamic symbol table rules in the BPABI DLL-like model on page C8-700

C8.5.6 Addressing modes in the BPABI DLL-like model on page C8-700

C8.5.4 Mandatory symbol versioning in the BPABI DLL-like model on page C8-699

Related references

C8.5.3 Linker command-line options for bare metal and DLL-like models on page C8-698

C1.7 --base_platform on page C1-344

C1.11 --bpabi on page C1-349

C1.32 --dll on page C1-373

C1.44 --export_all, --no_export_all on page C1-385

C1.105 --pltgot=type on page C1-454

C1.106 --pltgot_opts=mode on page C1-455

C1.116 --ropi on page C1-465

C1.117 --rosplit on page C1-466

C1.118 --rw_base=address on page C1-467

C1.119 --rwpi on page C1-468

C1.142 --symver_script=filename on page C1-493

C1.143 --symver_soname on page C1-494

Chapter C1 armlink Command-line Options on page C1-333

Related information

Base Platform ABI for the Arm Architecture

C8.5.4 Mandatory symbol versioning in the BPABI DLL-like model

The *Base Platform Application Binary Interface* (BPABI) DLL-like model requires static binding to ensure a symbol can be searched for at run-time.

This is because a post-linker might translate the symbolic information in a BPABI DLL to an import or export table that is indexed by an ordinal. In which case, it is not possible to search for a symbol at run-time.

Static binding is enforced in the BPABI with the use of symbol versioning. The command-line option `--symver_soname` is on by default for BPABI files, this means that all exported symbols are given a version based on the name of the DLL.

Note

The DLL-like model is not supported for AArch64 state.

Related concepts

C8.6 Symbol versioning on page C8-702

Related references

C1.142 --symver_script=filename on page C1-493

C1.143 --symver_soname on page C1-494

C8.5.5 Automatic dynamic symbol table rules in the BPABI DLL-like model

There are rules that apply to dynamic symbol tables for the *Base Platform Application Binary Interface* (BPABI) DLL-like model.

The following rules apply:

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 `--export_all` or `--export_dynamic` is set.

DLL

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.

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.

Note

The DLL-like model is not supported for AArch64 state.

You can manually export and import symbols using the `EXPORT` and `IMPORT` steering file commands. Use the `--edit` command-line option to specify a steering file command.

Related concepts

C5.6 Edit the symbol tables with a steering file on page C5-590

Related references

C5.6.2 Steering file command summary on page C5-590

C5.6.3 Steering file format on page C5-591

C1.36 --edit=file_list on page C1-377

C1.44 --export_all, --no_export_all on page C1-385

C1.45 --export_dynamic, --no_export_dynamic on page C1-386

C10.1 EXPORT steering file command on page C10-712

C10.3 IMPORT steering file command on page C10-714

C8.5.6 Addressing modes in the BPABI DLL-like model

The main difference between the bare metal and *Base Platform Application Binary Interface* (BPABI) DLL-like models is the addressing mode used when accessing imported and own-program code and data.

There are four options available that correspond to categories in the BPABI specification:

- None.
- Direct references.
- Indirect references.
- Relative static base address references.

You can control the selection of the required addressing mode with the following command-line options:

- `--pltgot`.
- `--pltgot_opts`.

———— **Note** ————

The DLL-like model is not supported for AArch64 state.

————————————

Related references

C1.105 --pltgot=type on page C1-454

C1.106 --pltgot_opts=mode on page C1-455

C8.5.7 C++ initialization in the BPABI DLL-like model

A *dynamically linked library* (DLL) supports the initialization of static constructors with a table that contains references to initializer functions that perform the initialization.

The table is stored in an ELF section with a special section type of `SHT_INIT_ARRAY`. For each of these initializers there is a relocation of type `R_ARM_TARGET1` to a function that performs the initialization.

The ELF *Application Binary Interface* (ABI) specification describes `R_ARM_TARGET1` as either a relative form, or an absolute form.

The Arm C libraries use the relative form. For example, if the linker detects a definition of the Arm C library `__cpp_initialize__aeabi`, it uses the relative form of `R_ARM_TARGET1` otherwise it uses the absolute form.

———— **Note** ————

The DLL-like model is not supported for AArch64 state.

————————————

Related concepts

C8.5.1 BPABI standard memory model on page C8-697

C8.5.4 Mandatory symbol versioning in the BPABI DLL-like model on page C8-699

C8.5.5 Automatic dynamic symbol table rules in the BPABI DLL-like model on page C8-700

C8.5.6 Addressing modes in the BPABI DLL-like model on page C8-700

Related references

C8.5.3 Linker command-line options for bare metal and DLL-like models on page C8-698

Related information

Initialization of the execution environment and execution of the application

C++ initialization, construction and destruction

C8.6 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.

This section contains the following subsections:

- *C8.6.1 Overview of symbol versioning* on page C8-702.
- *C8.6.2 Embedded symbols* on page C8-702.
- *C8.6.3 The symbol versioning script file* on page C8-703.
- *C8.6.4 Example of creating versioned symbols* on page C8-704.
- *C8.6.5 Linker options for enabling implicit symbol versioning* on page C8-704.

C8.6.1 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 concepts

C8.6.3 The symbol versioning script file on page C8-703

Related references

D1.58 --symbolversions, --no_symbolversions on page D1-792

C8.6.2 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`*.

The following example places the symbols `foo@ver1`, `foo@@ver2`, and `bar@@ver1` into the object symbol table:

```
int old_function(void) __asm__("foo@ver1");
int new_function(void) __asm__("foo@@ver2");
int other_function(void) __asm__("bar@@ver1");
```

The linker reads these symbols and creates version definitions `ver1` and `ver2`. The symbol `foo` is associated with a non-default version of `ver1`, and with a default version of `ver2`. The symbol `bar` is associated with a default version of `ver1`.

There is no way to create associations between versions with this method.

C8.6.3 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_names` in this version definition are local to the shared object and are not versioned.
- `"global:"` indicates that all subsequent `symbol_names` 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 concepts

C8.6.1 Overview of symbol versioning on page C8-702

C8.6.5 Linker options for enabling implicit symbol versioning on page C8-704

C8.6.4 Example of creating versioned symbols on page C8-704

Related references

C1.142 --symver_script=filename on page C1-493

C8.6.4 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 `foo@ver1`, `foo@@ver2`, and `bar@@ver1` into the object symbol table:

```
int old_function(void) __asm__("foo@ver1");
int new_function(void) __asm__("foo@@ver2");
int other_function(void) __asm__("bar@@ver1");
```

The corresponding script file includes the addition of dependency information so that `ver2` depends on `ver1` is:

```
ver1
{
    global:
        foo; bar;
    local:
        *;
};
ver2
{
    global:
        foo;
} ver1;
```

Related concepts

C8.6 Symbol versioning on page C8-702

C8.6.5 Linker options for enabling implicit symbol versioning on page C8-704

Related references

C1.142 --symver_script=filename on page C1-493

Chapter F3 Writing A32/T32 Instructions in armasm Syntax Assembly Language on page F3-925

C8.6.5 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 concepts

C8.6.3 The symbol versioning script file on page C8-703

C8.6 Symbol versioning on page C8-702

C8.6.2 Embedded symbols on page C8-702

Related references

C1.143 --symver_soname on page C1-494

Chapter C9
Features of the Base Platform Linking Model

Describes features of the Base Platform linking model supported by the Arm linker, `armlink`.

Note

The Base Platform linking model is not supported for AArch64 state.

It contains the following sections:

- *C9.1 Restrictions on the use of scatter files with the Base Platform model* on page C9-706.
- *C9.2 Scatter files for the Base Platform linking model* on page C9-708.
- *C9.3 Placement of PLT sequences with the Base Platform model* on page C9-710.

C9.1 Restrictions on the use of scatter files with the Base Platform model

The Base Platform model supports scatter files, with some restrictions.

Although there are no restrictions on the keywords you can use in a scatter file, there are restrictions on the types of scatter files you can use:

- A load region marked with the `RELOC` attribute must contain only execution regions with a relative base address of *`+offset`*. The following examples show valid and invalid scatter files using the `RELOC` attribute and *`+offset`* relative base address:

  Valid scatter file example using

  ```
  # This is valid. All execution regions have +offset addresses.
  LR1 0x8000 RELOC
  {
      ER_RELATIVE +0
      {
          *(+RO)
      }
  }
  ```

  Invalid scatter file example using

  ```
  # This is not valid. One execution region has an absolute base address.
  LR1 0x8000 RELOC
  {
      ER_RELATIVE +0
      {
          *(+RO)
      }
      ER_ABSOLUTE 0x1000
      {
          *(+RW)
      }
  }
  ```

- Any load region that requires a PLT section must contain at least one execution region containing code, that is not marked `OVERLAY`. This execution region holds the PLT section. An `OVERLAY` region cannot be used as the PLT must remain in memory at all times. The following examples show valid and invalid scatter files that define execution regions requiring a PLT section:

  Valid scatter file example for a load region that requires a PLT section

  ```
  # This is valid. ER_1 contains code and is not OVERLAY.
  LR_NEEDING_PLT 0x8000
  {
      ER_1 +0
      {
          *(+RO)
      }
  }
  ```

  Invalid scatter file example for a load region that requires a PLT section

  ```
  # This is not valid. All execution regions containing code are marked OVERLAY.
  LR_NEEDING_PLT 0x8000
  {
      ER_1 +0 OVERLAY
      {
          *(+RO)
      }
      ER_2 +0
      {
          *(+RW)
      }
  }
  ```

- If a load region requires a PLT section, then the PLT section must be placed within the load region. By default, if a load region requires a PLT section, the linker places the PLT section in the first execution region containing code. You can override this choice with a scatter-loading selector.

  If there is more than one load region containing code, the PLT section for a load region with name *`name`* is `.plt_`*`name`*. If there is only one load region containing code, the PLT section is called `.plt`.

The following examples show valid and invalid scatter files that place a PLT section:

Valid scatter file example for placing a PLT section

```
#This is valid. The PLT section for LR1 is placed in LR1.
LR1 0x8000
{
    ER1 +0
    {
        *(+RO)
    }
    ER2 +0
    {
        *(.plt_LR1)
    }
}
LR2 0x10000
{
    ER1 +0
    {
        *(other_code)
    }
}
```

Invalid scatter file example for placing a PLT section

```
#This is not valid. The PLT section of LR1 has been placed in LR2.
LR1 0x8000
{
    ER1 +0
    {
        *(+RO)
    }
}
LR2 0x10000
{
    ER1 +0
    {
        *(.plt_LR1)
    }
}
```

Related concepts

C2.5 Base Platform linking model overview on page C2-522

C9.3 Placement of PLT sequences with the Base Platform model on page C9-710

C7.3.4 Inheritance rules for load region address attributes on page C7-661

C7.3.5 Inheritance rules for the RELOC address attribute on page C7-662

C7.4.4 Inheritance rules for execution region address attributes on page C7-669

Related references

C7.3.3 Load region attributes on page C7-659

C7.4.3 Execution region attributes on page C7-666

C9.2 Scatter files for the Base Platform linking model

Scatter files containing relocatable and non-relocatable load regions for the Base Platform linking model.

Standard BPABI scatter file with relocatable load regions

If you do not specify a scatter file when linking for the Base Platform linking model, the linker uses a default scatter file defined for the standard *Base Platform Application Binary Interface* (BPABI) memory model. This scatter file defines the following relocatable load regions:

```
LR1 0x8000 RELOC
{
    ER_RO +0
    {
        *(+RO)
    }
}
LR2 0x0 RELOC
{
    ER_RW +0
    {
        *(+RW)
    }
    ER_ZI +0
    {
        *(+ZI)
    }
}
```

This example conforms to the BPABI, because it has the same two-region format as the BPABI specification.

Scatter file with some load regions that are not relocatable

This example shows two load regions LR1 and LR2 that are not relocatable.

```
LR1 0x8000
{
    ER_RO +0
    {
        *(+RO)
    }
    ER_RW +0
    {
        *(+RW)
    }
    ER_ZI +0
    {
        *(+ZI)
    }
}
LR2 0x10000
{
    ER_KNOWN_ADDRESS +0
    {
        *(fixedsection)
    }
}
LR3 0x20000 RELOC
{
    ER_RELOCATABLE +0
    {
        *(floatingsection)
    }
}
```

The linker does not have to generate dynamic relocations between LR1 and LR2 because they have fixed addresses. However, the `RELOC` load region LR3 might be widely separated from load regions LR1 and LR2 in the address space. Therefore, dynamic relocations are required between LR1 and LR3, and LR2 and LR3.

Use the options `--pltgot=direct --pltgot_opts=crosslr` to ensure a PLT is generated for each load region.

Related concepts

C2.2 Bare-metal linking model overview on page C2-519

C2.4 Base Platform Application Binary Interface (BPABI) linking model overview on page C2-521

C9.1 Restrictions on the use of scatter files with the Base Platform model on page C9-706

Related references

C7.3.3 Load region attributes on page C7-659

C9.3 Placement of PLT sequences with the Base Platform model

The linker supports *Procedure Linkage Table* (PLT) generation for multiple load regions containing code when linking in Base Platform mode.

To turn on PLT generation when in Base Platform mode (`--base_platform`) use `--pltgot=`*`option`* that generates PLT sequences. You can use the option `--pltgot_opts=crosslr` to add entries in the PLT for calls from and to `RELOC` load-regions. PLT generation for multiple Load Regions is only supported for `--pltgot=direct`.

The `--pltgot_opts=crosslr` option is useful when you have multiple load regions that might be moved relative to each other when the image is dynamically loaded. The linker generates a PLT for each load region so that calls do not have to be extended to reach a distant PLT.

Placement of linker generated PLT sections:

- When there is only one load region there is one PLT. The linker creates a section called `.plt` with an object `anon$$obj.o`.
- When there are multiple load regions, a PLT section is created for each load region that requires one. By default, the linker places the PLT section in the first execution region containing code. You can override this by specifying the exact PLT section name in the scatter file.

  For example, a load region with name *`LR_NAME`* the PLT section is called `.plt_`*`LR_NAME`* with an object of `anon$$obj.o`. To precisely name this PLT section in a scatter file, use the selector:

  `anon$$obj.o(.plt_`*`LR_NAME`*`)`

Be aware of the following:

- The linker gives an error message if the PLT for load region *`LR_NAME`* is moved out of load region *`LR_NAME`*.
- The linker gives an error message if load region *`LR_NAME`* contains a mixture of `RELOC` and non-`RELOC` execution regions. This is because it cannot guarantee that the `RELOC` execution regions are able to reach the PLT at run-time.
- `--pltgot=indirect` and `--pltgot=sbrel` are not supported for multiple load regions.

Related concepts

C2.5 Base Platform linking model overview on page C2-522

Related references

C1.7 --base_platform on page C1-344

C1.105 --pltgot=type on page C1-454

C1.106 --pltgot_opts=mode on page C1-455

Chapter C10

Linker Steering File Command Reference

Describes the steering file commands supported by the Arm linker, `armlink`.

It contains the following sections:

- *C10.1 EXPORT steering file command* on page C10-712.
- *C10.2 HIDE steering file command* on page C10-713.
- *C10.3 IMPORT steering file command* on page C10-714.
- *C10.4 RENAME steering file command* on page C10-715.
- *C10.5 REQUIRE steering file command* on page C10-716.
- *C10.6 RESOLVE steering file command* on page C10-717.
- *C10.7 SHOW steering file command* on page C10-719.

C10.1 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 concepts

C5.6 Edit the symbol tables with a steering file on page C5-590

Related references

C10.3 IMPORT steering file command on page C10-714

C1.96 --override_visibility on page C1-444

C10.2 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 concepts

C5.6 Edit the symbol tables with a steering file on page C5-590

Related references

C10.7 SHOW steering file command on page C10-719

C1.36 --edit=file_list on page C1-377

C1.101 --partial on page C1-449

C10.3 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 concepts

C5.6 Edit the symbol tables with a steering file on page C5-590

Related references

C1.96 --override_visibility on page C1-444

C10.1 EXPORT steering file command on page C10-712

C10.4 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`.

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 concepts

C5.6 Edit the symbol tables with a steering file on page C5-590

C10.5 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 concepts

C5.6 Edit the symbol tables with a steering file on page C5-590

C10.6 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 foobar;
void MyMP3_Init()
{
}
void MyMP3_Play()
{
}
```

Create a steering file, `ed.txt`, containing the line:

`RESOLVE MP3* AS MyMP3*`.

Enter the following command:

```
armlink file1.o file2.o --edit ed.txt --unresolved foobar
```

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` (`foobar`, `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 `foobar`.
- 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 concepts

C5.6 Edit the symbol tables with a steering file on page C5-590

Related references

C1.36 --edit=file_list on page C1-377

C1.150 --unresolved=symbol on page C1-501

C10.7 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 concepts

C5.6 Edit the symbol tables with a steering file on page C5-590

Related references

C10.2 HIDE steering file command on page C10-713

Part D

fromelf Reference

Chapter D1
fromelf Command-line Options

Describes the command-line options of the `fromelf` image converter provided with Arm Compiler.

It contains the following sections:

- *D1.1 --base [[object_file::]load_region_ID=]num* on page D1-725.
- *D1.2 --bin* on page D1-727.
- *D1.3 --bincombined* on page D1-728.
- *D1.4 --bincombined_base=address* on page D1-729.
- *D1.5 --bincombined_padding=size,num* on page D1-730.
- *D1.6 --cad* on page D1-731.
- *D1.7 --cadcombined* on page D1-733.
- *D1.8 --compare=option[,option,…]* on page D1-734.
- *D1.9 --continue_on_error* on page D1-736.
- *D1.10 --cpu=list (fromelf)* on page D1-737.
- *D1.11 --cpu=name (fromelf)* on page D1-738.
- *D1.12 --datasymbols* on page D1-741.
- *D1.13 --debugonly* on page D1-742.
- *D1.14 --decode_build_attributes* on page D1-743.
- *D1.15 --diag_error=tag[,tag,…] (fromelf)* on page D1-745.
- *D1.16 --diag_remark=tag[,tag,…] (fromelf)* on page D1-746.
- *D1.17 --diag_style={arm|ide|gnu} (fromelf)* on page D1-747.
- *D1.18 --diag_suppress=tag[,tag,…] (fromelf)* on page D1-748.
- *D1.19 --diag_warning=tag[,tag,…] (fromelf)* on page D1-749.
- *D1.20 --disassemble* on page D1-750.
- *D1.21 --dump_build_attributes* on page D1-751.
- *D1.22 --elf* on page D1-752.
- *D1.23 --emit=option[,option,…]* on page D1-753.

- *D1.24 --expandarrays* on page D1-755.
- *D1.25 --extract_build_attributes* on page D1-756.
- *D1.26 --fieldoffsets* on page D1-757.
- *D1.27 --fpu=list (fromelf)* on page D1-759.
- *D1.28 --fpu=name (fromelf)* on page D1-760.
- *D1.29 --globalize=option[,option,...]* on page D1-761.
- *D1.30 --help (fromelf)* on page D1-762.
- *D1.31 --hide=option[,option,...]* on page D1-763.
- *D1.32 --hide_and_localize=option[,option,...]* on page D1-764.
- *D1.33 --i32* on page D1-765.
- *D1.34 --i32combined* on page D1-766.
- *D1.35 --ignore_section=option[,option,...]* on page D1-767.
- *D1.36 --ignore_symbol=option[,option,...]* on page D1-768.
- *D1.37 --in_place* on page D1-769.
- *D1.38 --info=topic[,topic,...] (fromelf)* on page D1-770.
- *D1.39 input_file (fromelf)* on page D1-771.
- *D1.40 --interleave=option* on page D1-773.
- *D1.41 --linkview, --no_linkview* on page D1-774.
- *D1.42 --localize=option[,option,...]* on page D1-775.
- *D1.43 --m32* on page D1-776.
- *D1.44 --m32combined* on page D1-777.
- *D1.45 --only=section_name* on page D1-778.
- *D1.46 --output=destination* on page D1-779.
- *D1.47 --privacy (fromelf)* on page D1-780.
- *D1.48 --qualify* on page D1-781.
- *D1.49 --relax_section=option[,option,...]* on page D1-782.
- *D1.50 --relax_symbol=option[,option,...]* on page D1-783.
- *D1.51 --rename=option[,option,...]* on page D1-784.
- *D1.52 --select=select_options* on page D1-785.
- *D1.53 --show=option[,option,...]* on page D1-786.
- *D1.54 --show_and_globalize=option[,option,...]* on page D1-787.
- *D1.55 --show_cmdline (fromelf)* on page D1-788.
- *D1.56 --source_directory=path* on page D1-789.
- *D1.57 --strip=option[,option,...]* on page D1-790.
- *D1.58 --symbolversions, --no_symbolversions* on page D1-792.
- *D1.59 --text* on page D1-793.
- *D1.60 --version_number (fromelf)* on page D1-795.
- *D1.61 --vhx* on page D1-796.
- *D1.62 --via=file (fromelf)* on page D1-797.
- *D1.63 --vsn (fromelf)* on page D1-798.
- *D1.64 -w* on page D1-799.
- *D1.65 --wide64bit* on page D1-800.
- *D1.66 --widthxbanks* on page D1-801.

D1.1 --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.

———— **Note** ————

Not supported for AArch64 state.

————————————

Syntax

`--base [[`*`object_file`*`::]`*`load_region_ID`*`=]`*`num`*

Where:

`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.

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.

Restrictions

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.

Examples

The following table shows examples:

Table D1-1 Examples of using --base

`--base 0`	decimal value
`--base 0x8000`	hexadecimal value
`--base #0=0`	base address for the first load region
`--base foo.o::*=0`	base address for all load regions in `foo.o`
`--base #0=0,#1=0x8000`	base address for the first and second load regions

Related references

D1.33 --i32 on page D1-765

D1.34 --i32combined on page D1-766

D1.43 --m32 on page D1-776

D1.44 --m32combined on page D1-777

Related information

General considerations when using fromelf

D1.2 --bin

Produces plain binary output, one file for each load region. You can split the output from this option into multiple files with the `--widthxbanks` option.

Restrictions

The following restrictions apply:

- You cannot use this option with object files.
- You must use `--output` with this option.

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

To convert an ELF file to a plain binary file, for example `outfile.bin`, enter:

```
fromelf --bin --output=outfile.bin infile.axf
```

Related references

D1.46 --output=destination on page D1-779
D1.66 --widthxbanks on page D1-801

D1.3 --bincombined

Produces plain binary output. It generates one output file for an image containing multiple load regions.

Usage

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.

Use this option with `--bincombined_base` and `--bincombined_padding` to change the default values for the base address and padding.

Restrictions

The following restrictions apply:

- You cannot use this option with object files.
- You must use `--output` with this option.

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 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.

Examples

To produce a binary file that can be loaded at start address `0x1000`, enter:

```
fromelf --bincombined --bincombined_base=0x1000 --output=out.bin in.axf
```

To produce plain binary output and fill the space between load regions with copies of the 32-bit word `0x12345678`, enter:

```
fromelf --bincombined --bincombined_padding=4,0x12345678 --output=out.bin in.axf
```

Related concepts

C3.1.2 Input sections, output sections, regions, and program segments on page C3-529

Related references

D1.4 --bincombined_base=address on page D1-729

D1.5 --bincombined_padding=size,num on page D1-730

D1.46 --output=destination on page D1-779

D1.66 --widthxbanks on page D1-801

D1.4 --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.

Syntax

`--bincombined_base=address`

Where *`address`* is 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.

Default

By default the start address of the first load region in memory is used as the base address.

Restrictions

You must use `--bincombined` with this option. If you omit `--bincombined`, a warning message is displayed.

Example

```
--bincombined --bincombined_base=0x1000
```

Related concepts

C3.1.2 Input sections, output sections, regions, and program segments on page C3-529

Related references

D1.3 --bincombined on page D1-728

D1.5 --bincombined_padding=size,num on page D1-730

D1.5 --bincombined_padding=size,num

Enables you to specify a different padding value from the default used by the `--bincombined` output mode.

Syntax

`--bincombined_padding=`*`size`*`,`*`num`*
Where:

`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.

———— **Note** ————

`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.

————————————

Default

The default is `--bincombined_padding=1,0xFF`.

Restrictions

You must use `--bincombined` with this option. If you omit `--bincombined`, a warning message is displayed.

Examples

The following examples show how to use `--bincombined_padding`:

`--bincombined --bincombined_padding=4,0x12345678`

This example produces plain binary output and fills the space between load regions with copies of the 32-bit word 0x12345678.

`--bincombined --bincombined_padding=2,0x1234`

This example produces plain binary output and fills the space between load regions with copies of the 16-bit halfword 0x1234.

`--bincombined --bincombined_padding=2,0x01`

This example when specified for big endian memory, fills the space between load regions with `0x0100`.

Related references

D1.3 --bincombined on page D1-728

D1.4 --bincombined_base=address on page D1-729

D1.6 --cad

Produces a C array definition or C++ array definition containing binary output.

Usage

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.

Restrictions

You cannot use this option with object files.

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

The following examples show how to use `--cad`:

- To produce an array definition for an image that has a single load region, enter:

```
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
};
```

- For an image that has multiple load regions, the following commands create a file for each load region in the directory *root*`\myprojects\multiload\load_regions`:

```
cd root\myprojects\multiload
fromelf --cad image_multiload.axf --output 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.

Related concepts

C3.1.2 Input sections, output sections, regions, and program segments on page C3-529

Related references

D1.7 --cadcombined on page D1-733

D1.46 --output=destination on page D1-779

D1.7 --cadcombined

Produces a C array definition or C++ array definition containing binary output.

Usage

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.

Restrictions

You cannot use this option with object files.

Example

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 references

D1.6 --cad on page D1-731

D1.46 --output=destination on page D1-779

D1.8 --compare=option[,option,…]

Compares two input files and prints the differences.

Usage

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.

Syntax

`--compare=option[,option,…]`

Where *option* is 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:
- 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.

Related references

D1.35 --ignore_section=option[,option,…] on page D1-767
D1.36 --ignore_symbol=option[,option,…] on page D1-768
D1.49 --relax_section=option[,option,…] on page D1-782
D1.50 --relax_symbol=option[,option,…] on page D1-783

D1.9 --continue_on_error

Reports any errors and then continues.

Usage

Use `--diag_warning=error` instead of this option.

Related references

D1.19 --diag_warning=tag[,tag,…] (fromelf) on page D1-749

D1.10 --cpu=list (fromelf)

Lists the architecture and processor names that are supported by the `--cpu=name` option.

Syntax

```
--cpu=list
```

Related references

D1.11 --cpu=name (fromelf) on page D1-738

D1.11 --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.

Syntax

`--cpu=`*`name`*

Where *`name`* is 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 D1-2 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 Security Extensions 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.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.

Table D1-2 Supported Arm architectures (continued)

Architecture name	Description
`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.
`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 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`.

Usage

The following general points apply to processor and architecture options:

Processors

- Selecting the processor selects the appropriate architecture, *Floating-Point Unit* (FPU), and memory organization.

Architectures

- If you specify an architecture name for the `--cpu` option, machine code is disassembled by options such as `-c` or `--disassemble` 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.

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

Default

If you do not specify a `--cpu` option, then `fromelf` disassembles machine instructions in an architecture-independent way. This means that `fromelf` disassembles anything that it recognizes as an instruction by some architecture.

Note

To disassemble 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.

Example

To specify the Cortex-M4 processor, use:

```
--cpu=Cortex-M4
```

Related references

D1.10 --cpu=list (fromelf) on page D1-737
D1.20 --disassemble on page D1-750
D1.38 --info=topic[,topic,…] (fromelf) on page D1-770
D1.59 --text on page D1-793

D1.12 --datasymbols

Modifies the output information of data sections so that symbol definitions are interleaved.

Usage

You can use this option only with `--text -d`.

Related references

D1.59 --text on page D1-793

D1.13 --debugonly

Removes the content of any code or data sections.

Usage

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.

Restrictions

You must use `--elf` with this option.

Example

To create an ELF file, `debugout.axf`, from the ELF file `infile.axf`, containing only debug information, enter:

```
fromelf --elf --debugonly --output=debugout.axf infile.axf
```

Related references

D1.22 --elf on page D1-752

D1.14 --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*.

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-eabi-none -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)
        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 references

D1.21 --dump_build_attributes on page D1-751

D1.23 --emit=option[,option,…] on page D1-753

D1.25 --extract_build_attributes on page D1-756

Related information

Application Binary Interface for the Arm Architecture

D1.15 --diag_error=tag[,tag,…] (fromelf)

Sets diagnostic messages that have a specific tag to Error severity.

Syntax

`--diag_error=`*`tag[,tag,…]`*

Where *`tag`* can be:

- A diagnostic message number to set to error severity. This is the four-digit number, *nnnn*, with the tool letter prefix, but without the letter suffix indicating the severity.
- `warning`, to treat all warnings as errors.

Related references

D1.16 --diag_remark=tag[,tag,...] (fromelf) on page D1-746
D1.17 --diag_style={arm|ide|gnu} (fromelf) on page D1-747
D1.18 --diag_suppress=tag[,tag,...] (fromelf) on page D1-748
D1.19 --diag_warning=tag[,tag,...] (fromelf) on page D1-749

D1.16 --diag_remark=tag[,tag,…] (fromelf)

Sets diagnostic messages that have a specific tag to Remark severity.

Syntax

`--diag_remark=tag[,tag,…]`

Where `tag` is a comma-separated list of diagnostic message numbers. This is the four-digit number, *nnnn*, with the tool letter prefix, but without the letter suffix indicating the severity.

Related references

D1.15 --diag_error=tag[,tag,…] (fromelf) on page D1-745
D1.17 --diag_style={arm|ide|gnu} (fromelf) on page D1-747
D1.18 --diag_suppress=tag[,tag,…] (fromelf) on page D1-748
D1.19 --diag_warning=tag[,tag,…] (fromelf) on page D1-749

D1.17 --diag_style={arm|ide|gnu} (fromelf)

Specifies the display style for diagnostic messages.

Syntax

`--diag_style=string`

Where *`string`* is 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`.

Usage

`--diag_style=gnu` matches the format reported by the GNU Compiler, `gcc`.

`--diag_style=ide` matches the format reported by Microsoft Visual Studio.

Default

The default is `--diag_style=arm`.

Related references

D1.15 --diag_error=tag[,tag,…] (fromelf) on page D1-745
D1.16 --diag_remark=tag[,tag,…] (fromelf) on page D1-746
D1.18 --diag_suppress=tag[,tag,…] (fromelf) on page D1-748
D1.19 --diag_warning=tag[,tag,…] (fromelf) on page D1-749

D1.18 --diag_suppress=tag[,tag,…] (fromelf)

Suppresses diagnostic messages that have a specific tag.

Syntax

`--diag_suppress=`*`tag[,tag,…]`*

Where *`tag`* can be:

- A diagnostic message number to be suppressed. This is the four-digit number, *nnnn*, with the tool letter prefix, but without the letter suffix indicating the severity.
- `error`, to suppress all errors that can be downgraded.
- `warning`, to suppress all warnings.

Related references

D1.15 --diag_error=tag[,tag,…] (fromelf) on page D1-745
D1.16 --diag_remark=tag[,tag,…] (fromelf) on page D1-746
D1.17 --diag_style={arm|ide|gnu} (fromelf) on page D1-747
D1.19 --diag_warning=tag[,tag,…] (fromelf) on page D1-749

D1.19 --diag_warning=tag[,tag,…] (fromelf)

Sets diagnostic messages that have a specific tag to Warning severity.

Syntax

`--diag_warning=tag[,tag,…]`

Where `tag` can be:

- A diagnostic message number to set to warning severity. This is the four-digit number, *nnnn*, with the tool letter prefix, but without the letter suffix indicating the severity.
- `error`, to set all errors that can be downgraded to warnings.

Related references

D1.15 --diag_error=tag[,tag,...] (fromelf) on page D1-745
D1.16 --diag_remark=tag[,tag,...] (fromelf) on page D1-746
D1.17 --diag_style={arm|ide|gnu} (fromelf) on page D1-747
D1.19 --diag_warning=tag[,tag,...] (fromelf) on page D1-749

D1.20 --disassemble

Displays a disassembled version of the image to `stdout`. Disassembly is generated in `armasm` assembler syntax and not GNU assembler syntax.

Usage

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 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` cannot assemble code containing SVE instructions.

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 references

D1.11 --cpu=name (fromelf) on page D1-738
D1.23 --emit=option[,option,…] on page D1-753
D1.40 --interleave=option on page D1-773
D1.46 --output=destination on page D1-779
D1.59 --text on page D1-793

D1.21 --dump_build_attributes

Prints the contents of the build attributes section in raw hexadecimal form.

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 references

D1.14 --decode_build_attributes on page D1-743
D1.23 --emit=option[,option,…] on page D1-753
D1.25 --extract_build_attributes on page D1-756
D1.59 --text on page D1-793

D1.22 --elf

Selects ELF output mode.

Usage

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 on page D1-742`.
- `--globalize on page D1-761`.
- `--hide on page D1-763`.
- `--hide_and_localize on page D1-764`.
- `--in_place on page D1-769`.
- `--linkview on page D1-774` or `--no_linkview on page D1-774`. This option is deprecated.
- `--localize on page D1-775`.
- `--rename on page D1-784`.
- `--show on page D1-786`.
- `--show_and_globalize on page D1-787`.
- `--strip on page D1-790`.
- `--symbolversions on page D1-792` or `--no_symbolversions on page D1-792`.

Restrictions

You must use `--output` with this option. For more information, see `--output` on page D1-779.

D1.23 --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.

Restrictions

You can use this option only in text mode.

Syntax

`--emit=option[,option,…]`

Where *`option`* is 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 `--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.

Use this option with `--disassemble`.

`got`

Prints the contents of the *Global Offset Table* (GOT) section.

`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.

Related references

D1.20 --disassemble on page D1-750
D1.14 --decode_build_attributes on page D1-743
D1.24 --expandarrays on page D1-755
D1.59 --text on page D1-793

D1.24 --expandarrays

Prints data addresses, including arrays that are expanded both inside and outside structures.

Restrictions

You can use this option with `--text -a` or with `--fieldoffsets`.

Example

The following example shows the output for a struct containing arrays when `--fieldoffsets --expandarrays` is specified:

```
// 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
```

Related references

D1.26 --fieldoffsets on page D1-757

D1.59 --text on page D1-793

D1.25 --extract_build_attributes

Prints only the build attributes in a form that depends on the type of attribute.

Usage

Prints the build attributes in:

- Human-readable form for standard build attributes.
- Raw hexadecimal form for nonstandard build attributes.

Restrictions

This option has no effect for AArch64 state inputs.

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)
        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 references

D1.14 --decode_build_attributes on page D1-743

D1.21 --dump_build_attributes on page D1-751

D1.23 --emit=option[,option,…] on page D1-753

D1.59 --text on page D1-793

D1.26 --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.

Usage

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.

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 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.

Restrictions

This option:

- Requires that the object or image file has debug information.
- Can be used in text mode and with `--expandarrays`.

Examples

The following examples show how to use `--fieldoffsets`:

- To produce an output listing to `stdout` that contains all the field offsets from all structures in the file `inputfile.o`, enter:

  ```
  fromelf --fieldoffsets inputfile.o
  ```
- 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
  ```
- 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
  ```
- 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
|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 C:\Program Files\DS-5\bin\..\include
\stdio.h
|__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 references

D1.24 --expandarrays on page D1-755

D1.48 --qualify on page D1-781

D1.52 --select=select_options on page D1-785

D1.59 --text on page D1-793

F6.27 EQU on page F6-1050

F6.44 GET or INCLUDE on page F6-1068

Related information

Miscellaneous directives

D1.27 --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`

Related references

D1.28 --fpu=name (fromelf) on page D1-760

D1.28 --fpu=name (fromelf)

Specifies the target FPU architecture.

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

Syntax

`--fpu=`*`name`*

Where *`name`* is 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.

Note

Software floating-point linkage is not supported for AArch64 state.

Usage

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.

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.

Related references

D1.20 --disassemble on page D1-750
D1.27 --fpu=list (fromelf) on page D1-759
D1.38 --info=topic[,topic,…] (fromelf) on page D1-770
D1.59 --text on page D1-793

D1.29 --globalize=option[,option,…]

Converts the selected symbols to global symbols.

Syntax

`--globalize=option[,option,…]`

Where *`option`* is 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 references

D1.22 --elf on page D1-752

D1.31 --hide=option[,option,...] on page D1-763

D1.30 --help (fromelf)

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.

Related references

D1.55 --show_cmdline (fromelf) on page D1-788
D1.60 --version_number (fromelf) on page D1-795
D1.63 --vsn (fromelf) on page D1-798

D1.31 --hide=option[,option,…]

Changes the symbol visibility property to mark selected symbols as hidden.

Syntax

`--hide=option[,option,…]`

Where `option` is 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 references

D1.22 --elf on page D1-752

D1.53 --show=option[,option,…] on page D1-786

D1.32 --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,…]`

Where *`option`* is 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 references

D1.22 --elf on page D1-752

D1.33 --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.

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.

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.

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

To convert the ELF file `infile.axf` to an Intel Hex-32 format file, for example `outfile.bin`, enter:

```
fromelf --i32 --output=outfile.bin infile.axf
```

Related references

D1.1 --base [[object_file::]load_region_ID=]num on page D1-725
D1.34 --i32combined on page D1-766
D1.46 --output=destination on page D1-779

D1.34 --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.

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.

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 references

D1.1 --base [[object_file::]load_region_ID=]num on page D1-725
D1.33 --i32 on page D1-765
D1.46 --output=destination on page D1-779

D1.35 --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,…]`

Where *`option`* is 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:

- 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 references

D1.8 --compare=option[,option,…] on page D1-734
D1.36 --ignore_symbol=option[,option,…] on page D1-768
D1.49 --relax_section=option[,option,…] on page D1-782

D1.36 --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`*`,…]`

Where *`option`* is 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:

- 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 references

D1.8 --compare=option[,option,…] on page D1-734
D1.35 --ignore_section=option[,option,…] on page D1-767
D1.50 --relax_symbol=option[,option,…] on page D1-783

D1.37 --in_place

Enables the translation of ELF members in an input file to overwrite the previous content.

Restrictions

You must use `--elf` with this option.

Example

To remove debug information from members of the library file `test.a`, enter:

```
fromelf --elf --in_place --strip=debug test.a
```

Related references

D1.22 --elf on page D1-752

D1.57 --strip=option[,option,…] on page D1-790

D1.38 --info=topic[,topic,…] (fromelf)

Prints information about specific topics.

Syntax

`--info=topic[,topic,…]`

Where *`topic`* is 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

You can use this option only in text mode.

Related references

D1.59 --text on page D1-793

D1.39 input_file (fromelf)

Specifies the ELF file or archive containing ELF files to be processed.

Usage

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.

?

Matched 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:

```
'archive.a(??str*)'
```

Any files in the archive that are not processed are included in the output archive together with the processed files.

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 references

D1.2 --bin on page D1-727
D1.6 --cad on page D1-731
D1.7 --cadcombined on page D1-733
D1.8 --compare=option[,option,…] on page D1-734
D1.22 --elf on page D1-752
D1.33 --i32 on page D1-765
D1.34 --i32combined on page D1-766
D1.37 --in_place on page D1-769

D1.43 --m32 on page D1-776

D1.44 --m32combined on page D1-777

D1.46 --output=destination on page D1-779

D1.59 --text on page D1-793

D1.61 --vhx on page D1-796

Related information

Examples of processing ELF files in an archive

D1.40 --interleave=option

Inserts the original source code as comments into the disassembly if debug information is present.

Syntax

`--interleave=`*`option`*

Where *`option`* can be 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.

`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.

`source`

Interleaves comments containing source code. If the source code is no longer available then `fromelf` interleaves in the same way as `line_numbers`.

`source_only`

Interleaves comments containing source code. If the source code is no longer available then `fromelf` does not interleave that code.

Usage

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.

Default

The default is `--interleave=none`.

Related references

D1.20 --disassemble on page D1-750
D1.23 --emit=option[,option,…] on page D1-753
D1.56 --source_directory=path on page D1-789
D1.59 --text on page D1-793

D1.41 --linkview, --no_linkview

Controls the section-level view from the ELF image.

Usage

`--no_linkview` discards the section-level view and retains only the segment-level view (load time view).

Discarding the section-level view eliminates:

- 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.

Note

This option is deprecated.

Restrictions

The following restrictions apply:

- You must use `--elf` with `--linkview` and `--no_linkview`.

Example

To get ELF format output for `image.axf`, enter:

```
fromelf --no_linkview --elf image.axf --output=image_nlk.axf
```

Related references

D1.22 --elf on page D1-752
D1.47 --privacy (fromelf) on page D1-780
D1.57 --strip=option[,option,…] on page D1-790
C1.109 --privacy (armlink) on page C1-458

D1.42 --localize=option[,option,…]

Converts the selected symbols to local symbols.

Syntax

`--localize=option[,option,…]`

Where `option` is 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:

- 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 references

D1.22 --elf on page D1-752

D1.31 --hide=option[,option,…] on page D1-763

D1.43 --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.

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.

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.

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 references

D1.1 --base [[object_file::]load_region_ID=]num on page D1-725
D1.44 --m32combined on page D1-777
D1.46 --output=destination on page D1-779

D1.44 --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.

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.

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.

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 references

D1.1 --base [[object_file::]load_region_ID=]num on page D1-725

D1.43 --m32 on page D1-776

D1.46 --output=destination on page D1-779

D1.45 --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`*

Where *`section_name`* is 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.

Examples

The following examples show how to use `--only`:

- To display only the symbol table, `.symtab`, from the section-by-section output, enter:

  ```
  fromelf --only=.symtab --text -s test.axf
  ```
- To display all `ER`*`n`* sections, enter:

  ```
  fromelf --only=ER? test.axf
  ```
- To display the `HEAP` section and all symbol and string table sections, enter:

  ```
  fromelf --only=HEAP --only=.*tab --text -s -t test.axf
  ```

Related references

D1.59 --text on page D1-793

D1.46 --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`*

Where *`destination`* can be either a file or a directory. For example:

`--output=foo`

is the name of an output file

`--output=foo/`

is the name of an output directory.

Usage

Usage 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 references

D1.2 --bin on page D1-727

D1.22 --elf on page D1-752

D1.59 --text on page D1-793

D1.47 --privacy (fromelf)

Modifies the output file to protect your code in images and objects that are delivered to third parties.

Usage

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 references

D1.57 --strip=option[,option,...] on page D1-790

C1.79 --locals, --no_locals on page C1-425

C1.109 --privacy (armlink) on page C1-458

D1.48 --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.

Usage

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.

Examples

A structure called `foo` is defined in two headers for example, `one.h` and `two.h`.

Using `fromelf --fieldoffsets`, the linker might define the following symbols:

- `foo.a`, `foo.b`, and `foo.c`.
- `foo.x`, `foo.y`, and `foo.z`.

Using `fromelf --qualify --fieldoffsets`, the linker defines the following symbols:

- `oneh_foo.a`, `oneh_foo.b` and `oneh_foo.c`.
- `twoh_foo.x`, `twoh_foo.y` and `twoh_foo.z`.

Related references

D1.26 --fieldoffsets on page D1-757

D1.49 --relax_section=option[,option,…]

Changes the severity of a compare report for the specified sections to warnings rather than errors.

Restrictions

You must use `--compare` with this option.

Syntax

`--relax_section=`*`option`*`[,`*`option`*`,…]`

Where *`option`* is 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:

- 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.

Related references

D1.8 --compare=option[,option,…] on page D1-734
D1.35 --ignore_section=option[,option,…] on page D1-767
D1.50 --relax_symbol=option[,option,…] on page D1-783

D1.50 --relax_symbol=option[,option,…]

Changes the severity of a compare report for the specified symbols to warnings rather than errors.

Restrictions

You must use `--compare` with this option.

Syntax

`--relax_symbol=`*`option`*`[,`*`option`*`,…]`

Where *`option`* is 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.

Related references

D1.8 --compare=option[,option,…] on page D1-734

D1.36 --ignore_symbol=option[,option,…] on page D1-768

D1.49 --relax_section=option[,option,…] on page D1-782

D1.51 --rename=option[,option,…]

Renames the specified symbol in an output ELF object.

Restrictions

You must use `--elf` and `--output` with this option.

Syntax

`--rename=`*`option`*`[,`*`option`*`,…]`

Where *`option`* is 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.

Example

This example renames the `clock` symbol in the `timer.axf` image to `myclock`, and creates a new file called `mytimer.axf`:

```
fromelf --elf --rename=clock=myclock --output=mytimer.axf timer.axf
```

Related references

D1.22 --elf on page D1-752

D1.46 --output=destination on page D1-779

D1.52 --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`*

Where *`select_options`* is 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.

Usage

Use this option with either `--fieldoffsets` or `--text -a`.

Example

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 produces the output:

```
|structure.f1|                        EQU     0       ;  int16_t
|structure.f11|                       EQU     0x6     ;  int16_t
|structure.f111|                      EQU     0xC     ;  int16_t

    END
```

Related references

D1.26 --fieldoffsets on page D1-757

D1.59 --text on page D1-793

D1.53 --show=option[,option,…]

Changes the symbol visibility property of the selected symbols, to mark them with default visibility.

Syntax

`--show=option[,option,…]`

Where `option` is 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 references

D1.22 --elf on page D1-752

D1.31 --hide=option[,option,...] on page D1-763

D1.54 --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,…]`

Where *`option`* is 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 references

D1.22 --elf on page D1-752

D1.55 --show_cmdline (fromelf)

Outputs the command line used by the `armasm`, `armlink`, `fromelf`, and `armar` tools.

Usage

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 references

D1.62 --via=file (fromelf) on page D1-797

D1.56 --source_directory=path

Explicitly specifies the directory of the source code.

Syntax

`--source_directory=path`

Usage

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 references

D1.40 --interleave=option on page D1-773

D1.57 --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`*`,…]`
Where *`option`* is 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.

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

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 concepts

C5.1 About mapping symbols on page C5-578

Related references

D1.22 --elf on page D1-752

D1.41 --linkview, --no_linkview on page D1-774

D1.47 --privacy (fromelf) on page D1-780

C1.79 --locals, --no_locals on page C1-425

C1.109 --privacy (armlink) on page C1-458

D1.58 --symbolversions, --no_symbolversions

Turns off the decoding of symbol version tables.

Restrictions

If you use `--elf` with this option, you must also use `--output`.

Related concepts

C8.6 Symbol versioning on page C8-702

Related information

Base Platform ABI for the Arm Architecture

D1.59 --text

Prints image information in text format. You can decode an ELF image or ELF object file using this option.

Syntax

`--text [options]`

Where *`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 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.

`-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.

Usage

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.

Examples

The following examples show how to use `--text`:

- 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
  ```
- To list to `stdout` all the global and static data variables and all the structure field addresses, enter:

  ```
  fromelf -a --select=* infile.axf
  ```
- 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
  ```
- To produce a text file containing addresses of the nested structures only, enter:

  ```
  fromelf --text -a --select=*.*.* --output=structaddress.txt infile.axf
  ```
- 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
  ```
- To output only the `.symtab` section information in `infile.axf`, enter:

  ```
  fromelf --only .symtab -s --output=symtab.txt infile.axf
  ```

Related references

D1.11 --cpu=name (fromelf) on page D1-738
D1.23 --emit=option[,option,…] on page D1-753
D1.24 --expandarrays on page D1-755
D1.38 --info=topic[,topic,…] (fromelf) on page D1-770
D1.40 --interleave=option on page D1-773
D1.45 --only=section_name on page D1-778
D1.46 --output=destination on page D1-779
D1.52 --select=select_options on page D1-785
D1.64 -w on page D1-799
D1.20 --disassemble on page D1-750

Related information

Using fromelf to find where a symbol is placed in an executable ELF image
Getting Image Details

D1.60 --version_number (fromelf)

Displays the version of fromelf that you are using.

Usage

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 references

D1.30 --help (fromelf) on page D1-762

D1.63 --vsn (fromelf) on page D1-798

D1.61 --vhx

Produces Byte oriented (Verilog Memory Model) hexadecimal format output.

Usage

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 `--widthxbanks` option.

Restrictions

The following restrictions apply:

- You cannot use this option with object files.
- You must use `--output` with this 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.

Examples

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
```

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 references

D1.46 --output=destination on page D1-779

D1.66 --widthxbanks on page D1-801

D1.62 --via=file (fromelf)

Reads an additional list of input filenames and tool options from *`filename`*.

Syntax

`--via=`*`filename`*

Where *`filename`* is the name of a via file containing options to be included on the command line.

Usage

You can enter multiple `--via` options on the `armasm`, `armlink`, `fromelf`, and `armar` command lines. You can also include the `--via` options within a via file.

Related concepts

C.1 Overview of via files on page Appx-C-1172

Related references

C.2 Via file syntax rules on page Appx-C-1173

D1.63 --vsn (fromelf)

Displays the version information and the license details.

Note

`--vsn` is intended to report the version information for manual inspection. The `Component` line indicates the release of Arm Compiler 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`.

Example

```
> fromelf --vsn
Product: ARM Compiler N.n
Component: ARM Compiler N.n
Tool: fromelf [tool_id]
license_type
Software supplied by: ARM Limited
```

Related references

D1.30 --help (fromelf) on page D1-762

D1.60 --version_number (fromelf) on page D1-795

D1.64 -w

Causes some text output information that usually appears on multiple lines to be displayed on a single line.

Usage

This 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 references

D1.59 --text on page D1-793

D1.65 --wide64bit

Causes all addresses to be displayed with a width of 64 bits.

Usage

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 references

D1.39 input_file (fromelf) on page D1-771

D1.66 --widthxbanks

Outputs multiple files for multiple memory banks.

Syntax

`--`*`widthxbanks`*

Where:

`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 are:

```
--8x1
--8x2
--8x4
--16x1
--16x2
--32x1
--32x2
--64x1
```

Usage

`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
…
```

Restrictions

You must use `--output` with this option.

Related references

D1.2 --bin on page D1-727

D1.46 --output=destination on page D1-779

D1.61 --vhx on page D1-796

Part E

armar Reference

Chapter E1
armar Command-line Options

Describes the command-line options of the Arm librarian, `armar`.

It contains the following sections:

- *E1.1 archive* on page E1-807.
- *E1.2 -a pos_name* on page E1-808.
- *E1.3 -b pos_name* on page E1-809.
- *E1.4 -c (armar)* on page E1-810.
- *E1.5 -C (armar)* on page E1-811.
- *E1.6 --create* on page E1-812.
- *E1.7 -d* on page E1-813.
- *E1.8 --debug_symbols* on page E1-814.
- *E1.9 --diag_error=tag[,tag,…] (armar)* on page E1-815.
- *E1.10 --diag_remark=tag[,tag,…] (armar)* on page E1-816.
- *E1.11 --diag_style={arm|ide|gnu} (armar)* on page E1-817.
- *E1.12 --diag_suppress=tag[,tag,…] (armar)* on page E1-818.
- *E1.13 --diag_warning=tag[,tag,…] (armar)* on page E1-819.
- *E1.14 --entries* on page E1-820.
- *E1.15 file_list* on page E1-821.
- *E1.16 --help (armar)* on page E1-822.
- *E1.17 -i pos_name* on page E1-823.
- *E1.18 -m pos_name (armar)* on page E1-824.
- *E1.19 -n* on page E1-825.
- *E1.20 --new_files_only* on page E1-826.
- *E1.21 -p* on page E1-827.
- *E1.22 -r* on page E1-828.
- *E1.23 -s* on page E1-829.

- *E1.24 --show_cmdline (armar)* on page E1-830.
- *E1.25 --sizes* on page E1-831.
- *E1.26 -t* on page E1-832.
- *E1.27 -T* on page E1-833.
- *E1.28 -u (armar)* on page E1-834.
- *E1.29 -v (armar)* on page E1-835.
- *E1.30 --version_number (armar)* on page E1-836.
- *E1.31 --via=filename (armar)* on page E1-837.
- *E1.32 --vsn (armar)* on page E1-838.
- *E1.33 -x (armar)* on page E1-839.
- *E1.34 --zs* on page E1-840.
- *E1.35 --zt* on page E1-841.

E1.1 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 references

E1.15 file_list on page E1-821

E1.2 -a pos_name

Places new files in the library after the specified library member.

Syntax

`-a=`*`pos_name`*

Where *`pos_name`* is the name of a file in the library.

Usage

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
```

Related references

E1.3 -b pos_name on page E1-809
E1.17 -i pos_name on page E1-823
E1.18 -m pos_name (armar) on page E1-824
E1.22 -r on page E1-828

E1.3 -b pos_name

Places new files in the library before the specified library member.

Syntax

`-b=pos_name`

Where `pos_name` is the name of a file in the library.

Usage

This option takes precedence if you include `-a` on the same command line.

Related references

E1.2 -a pos_name on page E1-808
E1.17 -i pos_name on page E1-823
E1.18 -m pos_name (armar) on page E1-824
E1.22 -r on page E1-828

E1.4 -c (armar)

Suppresses the diagnostic message normally written to `stderr` when a library is created.

E1.5 -C (armar)

Instructs the librarian not to replace existing files with like-named files when performing extractions.

Usage

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 references

E1.27 -T on page E1-833
E1.33 -x (armar) on page E1-839

E1.6 --create

Creates a new library containing only the files specified in *`file_list`*. If the library already exists, its previous contents are discarded.

Usage

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`.

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 references

E1.15 file_list on page E1-821

E1.7 -d

Deletes one or more files specified in `file_list` from the library.

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 references

E1.15 file_list on page E1-821

E1.8 --debug_symbols

By default, debug symbols are excluded from an archive. Use `--debug_symbols` to include debug symbols in the archive.

Related information

About the Librarian

E1.9 --diag_error=tag[,tag,…] (armar)

Sets diagnostic messages that have a specific tag to Error severity.

Syntax

`--diag_error=tag[,tag,…]`

Where `tag` can be:

- A diagnostic message number to set to error severity. This is the four-digit number, *nnnn*, with the tool letter prefix, but without the letter suffix indicating the severity.
- `warning`, to treat all warnings as errors.

Related references

E1.10 --diag_remark=tag[,tag,...] (armar) on page E1-816

E1.11 --diag_style={arm|ide|gnu} (armar) on page E1-817

E1.12 --diag_suppress=tag[,tag,...] (armar) on page E1-818

E1.13 --diag_warning=tag[,tag,...] (armar) on page E1-819

E1.10 --diag_remark=tag[,tag,…] (armar)

Sets diagnostic messages that have a specific tag to Remark severity.

Syntax

`--diag_remark=tag[,tag,…]`

Where `tag` is a comma-separated list of diagnostic message numbers. This is the four-digit number, `nnnn`, with the tool letter prefix, but without the letter suffix indicating the severity.

Related references

E1.9 --diag_error=tag[,tag,...] (armar) on page E1-815
E1.11 --diag_style={arm|ide|gnu} (armar) on page E1-817
E1.12 --diag_suppress=tag[,tag,...] (armar) on page E1-818
E1.13 --diag_warning=tag[,tag,...] (armar) on page E1-819

E1.11 --diag_style={arm|ide|gnu} (armar)

Specifies the display style for diagnostic messages.

Syntax

`--diag_style=`*`string`*

Where *`string`* is 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`.

Usage

`--diag_style=gnu` matches the format reported by the GNU Compiler, `gcc`.

`--diag_style=ide` matches the format reported by Microsoft Visual Studio.

Default

The default is `--diag_style=arm`.

Related references

E1.9 --diag_error=tag[,tag,…] (armar) on page E1-815
E1.10 --diag_remark=tag[,tag,…] (armar) on page E1-816
E1.12 --diag_suppress=tag[,tag,…] (armar) on page E1-818
E1.13 --diag_warning=tag[,tag,…] (armar) on page E1-819

E1.12 --diag_suppress=tag[,tag,…] (armar)

Suppresses diagnostic messages that have a specific tag.

Syntax

`--diag_suppress=tag[,tag,…]`

Where *`tag`* can be:

- A diagnostic message number to be suppressed. This is the four-digit number, *nnnn*, with the tool letter prefix, but without the letter suffix indicating the severity.
- `error`, to suppress all errors that can be downgraded.
- `warning`, to suppress all warnings.

Related references

E1.9 --diag_error=tag[,tag,...] (armar) on page E1-815

E1.10 --diag_remark=tag[,tag,...] (armar) on page E1-816

E1.11 --diag_style={arm|ide|gnu} (armar) on page E1-817

E1.13 --diag_warning=tag[,tag,...] (armar) on page E1-819

E1.13 --diag_warning=tag[,tag,…] (armar)

Sets diagnostic messages that have a specific tag to Warning severity.

Syntax

`--diag_warning=tag[,tag,…]`

Where `tag` can be:

- A diagnostic message number to set to warning severity. This is the four-digit number, *nnnn*, with the tool letter prefix, but without the letter suffix indicating the severity.
- `error`, to set all errors that can be downgraded to warnings.

Related references

E1.9 --diag_error=tag[,tag,...] (armar) on page E1-815
E1.10 --diag_remark=tag[,tag,...] (armar) on page E1-816
E1.11 --diag_style={arm|ide|gnu} (armar) on page E1-817
E1.12 --diag_suppress=tag[,tag,...] (armar) on page E1-818

E1.14 --entries

Lists all object files in the library that have an entry point. You can use the `armasm` `ENTRY` directive to specify an entry point in legacy `armasm` syntax assembler code.

Usage

The format for the listing is:

```
ENTRY at offset num in section name of member
```

Example

The following example lists the entry point of each object file in `myasm.a`:

```
> 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 references

E1.25 --sizes on page E1-831

E1.35 --zt on page E1-841

F6.26 ENTRY on page F6-1049

Related information

Miscellaneous directives

E1.15 file_list

A space-separated list of ELF-compliant files, such as ELF objects and ELF libraries.

Usage

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.

E1.16 --help (armar)

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.

Related references

E1.30 --version_number (armar) on page E1-836

E1.32 --vsn (armar) on page E1-838

E1.17 -i pos_name

Places new files in the library before the specified library member.

Syntax

`-i pos_name`

Where *`pos_name`* is the name of a file in the library.

This is equivalent to `-b` *`pos_name`*

Related references

E1.2 -a pos_name on page E1-808
E1.3 -b pos_name on page E1-809
E1.18 -m pos_name (armar) on page E1-824
E1.22 -r on page E1-828

E1.18 -m pos_name (armar)

Moves files in a library to a specified position.

Syntax

`-m=`*`pos_name`*

Where *`pos_name`* is the name of a file in the library.

Usage

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 references

E1.2 -a pos_name on page E1-808
E1.3 -b pos_name on page E1-809
E1.17 -i pos_name on page E1-823

E1.19 -n

Suppresses the creation of a symbol table in the library.

Usage

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 references
E1.23 -s on page E1-829

E1.20 --new_files_only

Updates an object file in the archive only if the new object has a later timestamp.

Usage

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 references

E1.22 -r on page E1-828
E1.28 -u (armar) on page E1-834

E1.21 -p

Prints the contents of source files in a library to `stdout`.

Note

The files must be text files.

Example

To display the contents of `file1.c` in `mylib.a`, enter:

```
armar -p mylib.a file1.c
```

Related references

E1.26 -t on page E1-832

E1.22 -r

Replaces, or adds, files in the specified library.

Usage

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.

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 references

E1.2 -a pos_name on page E1-808
E1.3 -b pos_name on page E1-809
E1.17 -i pos_name on page E1-823
E1.28 -u (armar) on page E1-834
E1.15 file_list on page E1-821

E1.23 -s

Creates a symbol table in the library.

Usage

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.

Example

To create a symbol table in a library that was created using the `-n` option, enter:

```
armar -s mylib.a
```

Related references

E1.19 -n on page E1-825

E1.34 --zs on page E1-840

E1.24 --show_cmdline (armar)

Outputs the command line used by the `armasm`, `armlink`, `fromelf`, and `armar` tools.

Usage

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 references

E1.31 --via=filename (armar) on page E1-837

E1.25 --sizes

Lists the `Code`, `RO Data`, `RW Data`, `ZI Data`, and `Debug` sizes of each member in the library.

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 references

E1.14 --entries on page E1-820

E1.35 --zt on page E1-841

E1.26 -t

Prints a table of contents for the library.

Usage

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
```

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 references

E1.29 -v (armar) on page E1-835

E1.15 file_list on page E1-821

E1.27 -T

Enables truncation of filenames when extracted files have library names that are longer than the file system can support.

Usage

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 references

E1.5 -C (armar) on page E1-811
E1.33 -x (armar) on page E1-839

E1.28 -u (armar)

Updates older files in the specified archive.

Usage

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 references

E1.20 --new_files_only on page E1-826

E1.22 -r on page E1-828

E1.29 -v (armar)

Gives verbose output.

Usage

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 references

E1.7 -d on page E1-813
E1.21 -p on page E1-827
E1.22 -r on page E1-828
E1.26 -t on page E1-832
E1.33 -x (armar) on page E1-839

E1.30 --version_number (armar)

Displays the version of Arm Compiler tool that you are using.

Usage

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 references

E1.16 --help (armar) on page E1-822

E1.32 --vsn (armar) on page E1-838

E1.31 --via=filename (armar)

Reads an additional list of input filenames and tool options from `filename`.

Syntax

`--via=filename`

Where `filename` is the name of a via file containing options to be included on the command line.

Usage

You can enter multiple `--via` options on the `armasm`, `armlink`, `fromelf`, and `armar` command lines. You can also include the `--via` options within a via file.

Related concepts

C.1 Overview of via files on page Appx-C-1172

Related references

C.2 Via file syntax rules on page Appx-C-1173

E1.32 --vsn (armar)

Displays the version information and the license details.

Note

`--vsn` is intended to report the version information for manual inspection. The `Component` line indicates the release of Arm Compiler 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`.

Example

Example output:

```
> armar --vsn
Product: ARM Compiler N.n
Component: ARM Compiler N.n
Tool: armar [tool_id]
```

Related references

E1.16 --help (armar) on page E1-822

E1.30 --version_number (armar) on page E1-836

E1.33 -x (armar)

Extracts the files specified in *`file_list`* from the library to the current directory.

Usage

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 references

E1.5 -C (armar) on page E1-811

E1.27 -T on page E1-833

E1.15 file_list on page E1-821

E1.34 --zs

Displays the symbol table for all files in the library.

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 references

E1.19 -n on page E1-825

E1.23 -s on page E1-829

E1.35 --zt

Lists both the member sizes and entry points for all files in the library.

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 references

E1.14 --entries on page E1-820

E1.25 --sizes on page E1-831

Part F
armasm Legacy Assembler Reference

Chapter F1
armasm Command-line Options

Describes the `armasm` command-line syntax and command-line options.

It contains the following sections:

- *F1.1 --16* on page F1-847.
- *F1.2 --32* on page F1-848.
- *F1.3 --apcs=qualifier…qualifier* on page F1-849.
- *F1.4 --arm* on page F1-851.
- *F1.5 --arm_only* on page F1-852.
- *F1.6 --bi* on page F1-853.
- *F1.7 --bigend* on page F1-854.
- *F1.8 --brief_diagnostics, --no_brief_diagnostics* on page F1-855.
- *F1.9 --checkreglist* on page F1-856.
- *F1.10 --cpreproc* on page F1-857.
- *F1.11 --cpreproc_opts=option[,option,…]* on page F1-858.
- *F1.12 --cpu=list (armasm)* on page F1-859.
- *F1.13 --cpu=name (armasm)* on page F1-860.
- *F1.14 --debug* on page F1-863.
- *F1.15 --depend=dependfile* on page F1-864.
- *F1.16 --depend_format=string* on page F1-865.
- *F1.17 --diag_error=tag[,tag,…] (armasm)* on page F1-866.
- *F1.18 --diag_remark=tag[,tag,…] (armasm)* on page F1-867.
- *F1.19 --diag_style={arm|ide|gnu} (armasm)* on page F1-868.
- *F1.20 --diag_suppress=tag[,tag,…] (armasm)* on page F1-869.
- *F1.21 --diag_warning=tag[,tag,…] (armasm)* on page F1-870.
- *F1.22 --dllexport_all* on page F1-871.
- *F1.23 --dwarf2* on page F1-872.

- *F1.24 --dwarf3* on page F1-873.
- *F1.25 --errors=errorfile* on page F1-874.
- *F1.26 --exceptions, --no_exceptions* on page F1-875.
- *F1.27 --exceptions_unwind, --no_exceptions_unwind* on page F1-876.
- *F1.28 --execstack, --no_execstack* on page F1-877.
- *F1.29 --execute_only* on page F1-878.
- *F1.30 --fpmode=model* on page F1-879.
- *F1.31 --fpu=list (armasm)* on page F1-880.
- *F1.32 --fpu=name (armasm)* on page F1-881.
- *F1.33 -g (armasm)* on page F1-882.
- *F1.34 --help (armasm)* on page F1-883.
- *F1.35 -idir[,dir, ...]* on page F1-884.
- *F1.36 --keep (armasm)* on page F1-885.
- *F1.37 --length=n* on page F1-886.
- *F1.38 --li* on page F1-887.
- *F1.39 --library_type=lib* on page F1-888.
- *F1.40 --list=file* on page F1-889.
- *F1.41 --list=* on page F1-890.
- *F1.42 --littleend* on page F1-891.
- *F1.43 -m (armasm)* on page F1-892.
- *F1.44 --maxcache=n* on page F1-893.
- *F1.45 --md* on page F1-894.
- *F1.46 --no_code_gen* on page F1-895.
- *F1.47 --no_esc* on page F1-896.
- *F1.48 --no_hide_all* on page F1-897.
- *F1.49 --no_regs* on page F1-898.
- *F1.50 --no_terse* on page F1-899.
- *F1.51 --no_warn* on page F1-900.
- *F1.52 -o filename (armasm)* on page F1-901.
- *F1.53 --pd* on page F1-902.
- *F1.54 --predefine "directive"* on page F1-903.
- *F1.55 --reduce_paths, --no_reduce_paths* on page F1-904.
- *F1.56 --regnames* on page F1-905.
- *F1.57 --report-if-not-wysiwyg* on page F1-906.
- *F1.58 --show_cmdline (armasm)* on page F1-907.
- *F1.59 --thumb* on page F1-908.
- *F1.60 --unaligned_access, --no_unaligned_access* on page F1-909.
- *F1.61 --unsafe* on page F1-910.
- *F1.62 --untyped_local_labels* on page F1-911.
- *F1.63 --version_number (armasm)* on page F1-912.
- *F1.64 --via=filename (armasm)* on page F1-913.
- *F1.65 --vsn (armasm)* on page F1-914.
- *F1.66 --width=n* on page F1-915.
- *F1.67 --xref* on page F1-916.

F1.1 --16

Instructs `armasm` to interpret instructions as T32 instructions using the pre-UAL T32 syntax.

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.

Note

Not supported for AArch64 state.

Related references

F1.59 --thumb on page F1-908
F6.12 CODE16 directive on page F6-1035

F1.2 --32

A synonym for the `--arm` command-line option.

———— **Note** ————

Not supported for AArch64 state.

————————————

Related references

F1.4 --arm on page F1-851

F1.3 --apcs=qualifier…qualifier

Controls interworking and position independence when generating code.

Syntax

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

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

- At least one qualifier present.
- No spaces or commas separating individual qualifiers in the list.

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 Armv7-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.

Usage

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)

F1.4 --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.

Note

Not supported for AArch64 state.

Related references

F1.2 --32 on page F1-848
F1.5 --arm_only on page F1-852
F6.8 ARM or CODE32 directive on page F6-1031

F1.5 --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.

——— **Note** ———

Not supported for AArch64 state.

———————————

Related references

F1.4 --arm on page F1-851

F1.6 --bi

A synonym for the `--bigend` command-line option.

Related references

F1.7 --bigend on page F1-854

F1.42 --littleend on page F1-891

F1.7 --bigend

Generates code suitable for an Arm processor using big-endian memory access.

The default is `--littleend`.

Related references

F1.42 --littleend on page F1-891
F1.6 --bi on page F1-853

F1.8 --brief_diagnostics, --no_brief_diagnostics

Enables and disables the output of brief diagnostic messages.

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 references

F1.17 --diag_error=tag[,tag,…] (armasm) on page F1-866

F1.21 --diag_warning=tag[,tag,…] (armasm) on page F1-870

F1.9 --checkreglist

Instructs the `armasm` to check `RLIST`, `LDM`, and `STM` register lists to ensure that all registers are provided in increasing register number order.

When this option is used, `armasm` gives a warning if the registers are not listed in order.

——— **Note** ———

In AArch32 state, this option is deprecated. Use `--diag_warning 1206` instead. In AArch64 state, this option is not supported..

———————————

Related references

F1.21 --diag_warning=tag[,tag,…] (armasm) on page F1-870

F1.10 --cpreproc

Instructs `armasm` to call `armclang` to preprocess the input file before assembling it.

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 concepts

F4.14 Using the C preprocessor on page F4-980

Related references

F1.11 --cpreproc_opts=option[,option,…] on page F1-858

B1.88 -x (armclang) on page B1-171

Related information

Command-line options for preprocessing assembly source code

F1.11 --cpreproc_opts=option[,option,…]

Enables `armasm` to pass options to `armclang` when using the C preprocessor.

Syntax

`--cpreproc_opts=`*`option[,option,…]`*

Where `option[,option,…]` is 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` `--cpu` option.

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 concepts

F4.14 Using the C preprocessor on page F4-980

Related references

F1.10 --cpreproc on page F1-857

B1.49 -march on page B1-110

B1.56 -mcpu on page B1-125

B1.78 --target on page B1-161

B1.88 -x (armclang) on page B1-171

Related information

Command-line options for preprocessing assembly source code

Mandatory armclang options

F1.12 --cpu=list (armasm)

Lists the architecture and processor names that are supported by the `--cpu=name` option.

Syntax

`--cpu=list`

Related references

F1.13 --cpu=name (armasm) on page F1-860

F1.13 --cpu=name (armasm)

Enables code generation for the selected Arm processor or architecture.

Syntax

`--cpu=`*`name`*

Where *`name`* is 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.

Note

`armasm` does not support architectures later than Armv8.3.

Table F1-1 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 Security Extensions 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.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.

Table F1-1 Supported Arm architectures (continued)

Architecture name	Description
`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.
`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.

Note

- The full list of supported architectures and processors depends on your license.

Default

There is no default option for `--cpu`.

Usage

The following general points apply to processor and architecture options:

Processors

- Selecting the processor selects the appropriate architecture, *Floating-Point Unit* (FPU), and memory organization.
- If you specify a processor for the `--cpu` option, the generated code is optimized for that processor. This 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.

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.

Restrictions

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

Example

```
armasm --cpu=Cortex-A17 inputfile.s
```

Related references

F1.3 --apcs=qualifier…qualifier on page F1-849

F1.12 --cpu=list (armasm) on page F1-859

F1.32 --fpu=name (armasm) on page F1-881

F1.59 --thumb on page F1-908

F1.61 --unsafe on page F1-910

Related information

Arm Architecture Reference Manual

F1.14 --debug

Instructs the assembler to generate DWARF debug tables.

`--debug` is a synonym for `-g`. The default is DWARF 3.

Note

Local symbols are not preserved with `--debug`. You must specify `--keep` if you want to preserve the local symbols to aid debugging.

Related references

F1.23 --dwarf2 on page F1-872
F1.24 --dwarf3 on page F1-873
F1.36 --keep (armasm) on page F1-885
F1.33 -g (armasm) on page F1-882

F1.15 --depend=dependfile

Writes makefile dependency lines to a file.

Source file dependency lists are suitable for use with make utilities.

Related references

F1.45 --md on page F1-894

F1.16 --depend_format=string on page F1-865

F1.16 --depend_format=string

Specifies the format of output dependency files, for compatibility with some UNIX make programs.

Syntax

`--depend_format=`*`string`*

Where *`string`* is one of:

`unix`

generates dependency file entries using UNIX-style path separators.

`unix_escaped`

is the same as `unix`, but escapes spaces with `\`.

`unix_quoted`

is the same as `unix`, but surrounds path names with double quotes.

Related references

F1.15 --depend=dependfile on page F1-864

F1.17 --diag_error=tag[,tag,…] (armasm)

Sets diagnostic messages that have a specific tag to Error severity.

Syntax

`--diag_error=tag[,tag,…]`

Where *tag* can be:

- A diagnostic message number to set to error severity. This is the four-digit number, *nnnn*, with the tool letter prefix, but without the letter suffix indicating the severity.
- `warning`, to treat all warnings as errors.

Usage

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 F1-2 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.

Related references

F1.8 --brief_diagnostics, --no_brief_diagnostics on page F1-855
F1.18 --diag_remark=tag[,tag,...] (armasm) on page F1-867
F1.20 --diag_suppress=tag[,tag,...] (armasm) on page F1-869
F1.21 --diag_warning=tag[,tag,...] (armasm) on page F1-870

F1.18 --diag_remark=tag[,tag,…] (armasm)

Sets diagnostic messages that have a specific tag to Remark severity.

Syntax

`--diag_remark=tag[,tag,…]`

Where *`tag`* is a comma-separated list of diagnostic message numbers. This is the four-digit number, *`nnnn`*, with the tool letter prefix, but without the letter suffix indicating the severity.

Usage

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 references

F1.8 --brief_diagnostics, --no_brief_diagnostics on page F1-855
F1.17 --diag_error=tag[,tag,...] (armasm) on page F1-866
F1.20 --diag_suppress=tag[,tag,...] (armasm) on page F1-869
F1.21 --diag_warning=tag[,tag,...] (armasm) on page F1-870

F1.19 --diag_style={arm|ide|gnu} (armasm)

Specifies the display style for diagnostic messages.

Syntax

`--diag_style=`*`string`*

Where *`string`* is 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`.

Usage

`--diag_style=gnu` matches the format reported by the GNU Compiler, `gcc`.

`--diag_style=ide` matches the format reported by Microsoft Visual Studio.

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

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

Default

The default is `--diag_style=arm`.

Related references

F1.8 --brief_diagnostics, --no_brief_diagnostics on page F1-855

F1.20 --diag_suppress=tag[,tag,…] (armasm)

Suppresses diagnostic messages that have a specific tag.

Syntax

`--diag_suppress=tag[,tag,…]`
Where *tag* can be:

- A diagnostic message number to be suppressed. This is the four-digit number, *nnnn*, with the tool letter prefix, but without the letter suffix indicating the severity.
- `error`, to suppress all errors that can be downgraded.
- `warning`, to suppress all warnings.

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 references

F1.8 --brief_diagnostics, --no_brief_diagnostics on page F1-855
F1.17 --diag_error=tag[,tag,…] (armasm) on page F1-866
F1.18 --diag_remark=tag[,tag,…] (armasm) on page F1-867
F1.20 --diag_suppress=tag[,tag,…] (armasm) on page F1-869
F1.21 --diag_warning=tag[,tag,…] (armasm) on page F1-870

F1.21 --diag_warning=tag[,tag,…] (armasm)

Sets diagnostic messages that have a specific tag to Warning severity.

Syntax

`--diag_warning=`*`tag[,tag,…]`*

Where *`tag`* can be:

- A diagnostic message number to set to warning severity. This is the four-digit number, *`nnnn`*, with the tool letter prefix, but without the letter suffix indicating the severity.
- `error`, to set all errors that can be downgraded to warnings.

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 references

F1.8 --brief_diagnostics, --no_brief_diagnostics on page F1-855

F1.17 --diag_error=tag[,tag,...] (armasm) on page F1-866

F1.18 --diag_remark=tag[,tag,...] (armasm) on page F1-867

F1.20 --diag_suppress=tag[,tag,...] (armasm) on page F1-869

F1.22 --dllexport_all

Controls symbol visibility when building DLLs.

This option gives all exported global symbols `STV_PROTECTED` visibility in ELF rather than `STV_HIDDEN`, unless overridden by source directives.

Related references

F6.28 EXPORT or GLOBAL on page F6-1051

F1.23 --dwarf2

Uses DWARF 2 debug table format.

Note

Not supported for AArch64 state.

This option can be used with `--debug`, to instruct `armasm` to generate DWARF 2 debug tables.

Related references

F1.14 --debug on page F1-863
F1.24 --dwarf3 on page F1-873

F1.24 --dwarf3

Uses DWARF 3 debug table format.

This option can be used with `--debug`, to instruct the assembler to generate DWARF 3 debug tables. This is the default if `--debug` is specified.

Related references

F1.14 --debug on page F1-863
F1.23 --dwarf2 on page F1-872

F1.25 --errors=errorfile

Redirects the output of diagnostic messages from stderr to the specified errors file.

F1.26 --exceptions, --no_exceptions

Enables or disables exception handling.

Note

Not supported for AArch64 state.

These options instruct `armasm` to switch on or off exception table generation for all functions defined by `FUNCTION` (or `PROC`) and `ENDFUNC` (or `ENDP`) directives.

`--no_exceptions` causes no tables to be generated. It is the default.

Related references

F1.27 --exceptions_unwind, --no_exceptions_unwind on page F1-876
F6.40 FRAME UNWIND ON on page F6-1064
F6.41 FRAME UNWIND OFF on page F6-1065
F6.42 FUNCTION or PROC on page F6-1066
F6.25 ENDFUNC or ENDP on page F6-1048

F1.27 --exceptions_unwind, --no_exceptions_unwind

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

Note

Not supported for AArch64 state.

The default is `--exceptions_unwind`.

For finer control, use the `FRAME UNWIND ON` and `FRAME UNWIND OFF` directives.

Related references

F1.26 --exceptions, --no_exceptions on page F1-875
F6.40 FRAME UNWIND ON on page F6-1064
F6.41 FRAME UNWIND OFF on page F6-1065
F6.42 FUNCTION or PROC on page F6-1066
F6.25 ENDFUNC or ENDP on page F6-1048

F1.28 --execstack, --no_execstack

Generates a `.note.GNU-stack` section marking the stack as either executable or non-executable.

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 F1-3 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 references

F6.7 AREA on page F6-1027

F1.29 --execute_only

Adds the `EXECONLY AREA` attribute to all code sections.

Usage

The `EXECONLY AREA` attribute causes the linker to treat the section as execute-only.

It is the user's responsibility to ensure that the code in the section is safe to run in execute-only memory. For example:

- The code must not contain literal pools.
- The code must not attempt to load data from the same, or another, execute-only section.

Restrictions

This option is only supported for:

- Processors that support the Armv8-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.

Note

Arm has only performed limited testing of execute-only code on Armv6-M targets.

F1.30 --fpmode=model

Specifies floating-point standard conformance and sets library attributes and floating-point optimizations.

Syntax

`--fpmode=`*`model`*

Where *`model`* is 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.

`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 does not cause any changes to the code that you write.

Example

```
armasm --cpu=8-A.32 --fpmode ieee_full inputfile.s
```

Related references

F1.32 --fpu=name (armasm) on page F1-881

Related information

IEEE Standards Association

F1.31 --fpu=list (armasm)

Lists the FPU architecture names that are supported by the `--fpu=name` option.

Example

```
armasm --fpu=list
```

Related references

F1.30 --fpmode=model on page F1-879
F1.32 --fpu=name (armasm) on page F1-881

F1.32 --fpu=name (armasm)

Specifies the target FPU architecture.

Syntax

`--fpu=`*`name`*

Where *`name`* is 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.

Note

Software floating-point linkage is not supported for AArch64 state.

Usage

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 references

F1.30 --fpmode=model on page F1-879

F1.33 -g (armasm)

Enables the generation of debug tables.

This option is a synonym for `--debug`.

Related references

F1.14 --debug on page F1-863

F1.34 --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.

Related references

F1.63 --version_number (armasm) on page F1-912
F1.65 --vsn (armasm) on page F1-914

F1.35 -idir[,dir, …]

Adds directories to the source file include path.

Any directories added using this option have to be fully qualified.

Related references

F6.44 GET or INCLUDE on page F6-1068

F1.36 --keep (armasm)

Instructs the assembler to keep named local labels in the symbol table of the object file, for use by the debugger.

Related references

F6.49 KEEP on page F6-1075

F1.37 --length=n

Sets the listing page length.

Length zero means an unpaged listing. The default is 66 lines.

Related references

F1.40 --list=file on page F1-889

F1.38 --li

A synonym for the `--littleend` command-line option.

Related references

F1.42 --littleend on page F1-891

F1.7 --bigend on page F1-854

F1.39 --library_type=lib

Enables the selected library to be used at link time.

Syntax

`--library_type=`*`lib`*

Where *`lib`* is one of:

`standardlib`

Specifies that the full Arm runtime libraries are selected at link time. This is the default.

`microlib`

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

Note

- 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

F1.40 --list=file

Instructs the assembler to output a detailed listing of the assembly language produced by the assembler to a file.

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 references

F1.50 --no_terse on page F1-899
F1.66 --width=n on page F1-915
F1.37 --length=n on page F1-886
F1.67 --xref on page F1-916
F6.56 OPT on page F6-1084

F1.41 --list=

Instructs the assembler to send the detailed assembly language listing to *`inputfile`*`.lst`.

Note

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 references

F1.40 --list=file on page F1-889

F1.42 --littleend

Generates code suitable for an Arm processor using little-endian memory access.

Related references

F1.7 --bigend on page F1-854
F1.38 --li on page F1-887

F1.43 -m (armasm)

Instructs the assembler to write source file dependency lists to `stdout`.

Related references

F1.45 --md on page F1-894

F1.44 --maxcache=n

Sets the maximum source cache size in bytes.

The default is 8MB. `armasm` gives a warning if the size is less than 8MB.

F1.45 --md

Creates makefile dependency lists.

This option instructs the assembler to write source file dependency lists to `inputfile.d`.

Related references

F1.43 -m (armasm) on page F1-892

F1.46 --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.

F1.47 --no_esc

Instructs the assembler to ignore C-style escaped special characters, such as \n and \t.

F1.48 --no_hide_all

Gives all exported and imported global symbols `STV_DEFAULT` visibility in ELF rather than `STV_HIDDEN`, unless overridden using source directives.

You can use the following directives to specify an attribute that overrides the implicit symbol visibility:

- `EXPORT`.
- `EXTERN`.
- `GLOBAL`.
- `IMPORT`.

Related references

F6.28 EXPORT or GLOBAL on page F6-1051

F6.46 IMPORT and EXTERN on page F6-1071

F1.49 --no_regs

Instructs `armasm` not to predefine register names.

Note

This option is deprecated. In AArch32 state, use `--regnames=none` instead.

Related references

F1.56 --regnames on page F1-905

F1.50 --no_terse

Instructs the assembler to show in the list file the lines of assembly code that it has skipped because of conditional assembly.

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 references

F1.40 --list=file on page F1-889

F1.51 --no_warn

Turns off warning messages.

Related references

F1.21 --diag_warning=tag[,tag,…] (armasm) on page F1-870

F1.52 -o filename (armasm)

Specifies the name of the output file.

If this option is not used, the assembler creates an object filename in the form `inputfilename.o`. This option is case-sensitive.

F1.53 --pd

A synonym for the `--predefine` command-line option.

Related references

F1.54 --predefine "directive" on page F1-903

F1.54 --predefine "directive"

Instructs `armasm` to pre-execute one of the `SETA`, `SETL`, or `SETS` directives.

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.

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, Arm recommends you use the compiler to pre-process your assembly code.

Related references

F1.53 --pd on page F1-902

F6.43 GBLA, GBLL, and GBLS on page F6-1067

F6.45 IF, ELSE, ENDIF, and ELIF on page F6-1069

F6.64 SETA, SETL, and SETS on page F6-1094

F1.55 --reduce_paths, --no_reduce_paths

Enables or disables the elimination of redundant path name information in file paths.

Windows systems impose a 260 character limit on file paths. Where relative pathnames exist whose absolute names expand to longer than 260 characters, you can use the `--reduce_paths` option to reduce absolute pathname length by matching up directories with corresponding instances of `..` and eliminating the directory/`..` sequences in pairs.

`--no_reduce_paths` is the default.

Note

Arm recommends that you avoid using long and deeply nested file paths, in preference to minimizing path lengths using the `--reduce_paths` option.

Note

This option is valid for 32-bit Windows systems only.

F1.56 --regnames

Controls the predefinition of register names.

Note

Not supported for AArch64 state.

Syntax

`--regnames=`*`option`*

Where *`option`* is 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`.

Related references

F1.49 --no_regs on page F1-898

F1.56 --regnames on page F1-905

F1.3 --apcs=qualifier…qualifier on page F1-849

F1.57 --report-if-not-wysiwyg

Instructs `armasm` to report when it outputs an encoding that was not directly requested in the source code.

This can happen when `armasm`:
- 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.

Note

Not supported for AArch64 state.

F1.58 --show_cmdline (armasm)

Outputs the command line used by the `armasm`, `armlink`, `fromelf`, and `armar` tools.

Usage

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 references

F1.64 --via=filename (armasm) on page F1-913

F1.59 --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.

—————— **Note** ——————

Not supported for AArch64 state.

————————————————

Related references

F1.4 --arm on page F1-851
F6.66 THUMB directive on page F6-1097

F1.60 --unaligned_access, --no_unaligned_access

Enables or disables unaligned accesses to data on Arm-based processors.

These options instruct the assembler to set an attribute in the object file to enable or disable the use of unaligned accesses.

F1.61 --unsafe

Enables instructions for other architectures to be assembled without error.

Note

Not supported for AArch64 state.

It downgrades error messages to corresponding warning messages. It also suppresses warnings about operator precedence.

Related concepts

F5.20 Binary operators on page F5-1006

Related references

F1.17 --diag_error=tag[,tag,...] (armasm) on page F1-866

F1.21 --diag_warning=tag[,tag,...] (armasm) on page F1-870

F1.62 --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.

Note

Not supported for AArch64 state.

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.

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 concepts

F5.10 Numeric local labels on page F5-996

F1.63 --version_number (armasm)

Displays the version of armasm that you are using.

Usage

The assembler 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.

F1.64 --via=filename (armasm)

Reads an additional list of input filenames and tool options from `filename`.

Syntax

`--via=filename`

Where `filename` is the name of a via file containing options to be included on the command line.

Usage

You can enter multiple `--via` options on the `armasm`, `armlink`, `fromelf`, and `armar` command lines. You can also include the `--via` options within a via file.

Related concepts

C.1 Overview of via files on page Appx-C-1172

Related references

C.2 Via file syntax rules on page Appx-C-1173

F1.65 --vsn (armasm)

Displays the version information and the license details.

Note

`--vsn` is intended to report the version information for manual inspection. The `Component` line indicates the release of Arm Compiler 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`.

Example

```
> armasm --vsn
Product: ARM Compiler N.n
Component: ARM Compiler N.n
Tool: armasm [tool_id]
license_type
Software supplied by: ARM Limited
```

F1.66 --width=n

Sets the listing page width.

The default is 79 characters.

Related references

F1.40 --list=file on page F1-889

F1.67 --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.

The default is off.

Related references

F1.40 --list=file on page F1-889

Chapter F2
Structure of armasm Assembly Language Modules

Describes the structure of `armasm` assembly language source files.

It contains the following sections:

- *F2.1 Syntax of source lines in armasm syntax assembly language* on page F2-918.
- *F2.2 Literals* on page F2-920.
- *F2.3 ELF sections and the AREA directive* on page F2-921.
- *F2.4 An example armasm syntax assembly language module* on page F2-922.

F2.1 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.

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.

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 concepts

F5.6 Labels on page F5-992

F5.10 Numeric local labels on page F5-996

F5.13 String literals on page F5-999

Related references

F2.2 Literals on page F2-920

F5.1 Symbol naming rules on page F5-987

F5.15 Syntax of numeric literals on page F5-1001

F2.2 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 references

F2.1 Syntax of source lines in armasm syntax assembly language on page F2-918

F2.3 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).

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 concepts

F2.4 An example armasm syntax assembly language module on page F2-922

Related references

F6.7 AREA on page F6-1027

Chapter C6 Scatter-loading Features on page C6-595

F2.4 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
```

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`.

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 concepts

F2.3 ELF sections and the AREA directive on page F2-921

Related references

F6.24 END on page F6-1047

F6.26 ENTRY on page F6-1049

Related information

Semihosting for AArch32 and AArch64

Chapter F3
Writing A32/T32 Instructions in armasm Syntax Assembly Language

Describes the use of a few basic A32 and T32 instructions and the use of macros in the armasm syntax assembly language.

It contains the following sections:

- *F3.1 About the Unified Assembler Language* on page F3-927.
- *F3.2 Syntax differences between UAL and A64 assembly language* on page F3-928.
- *F3.3 Register usage in subroutine calls* on page F3-929.
- *F3.4 Load immediate values* on page F3-930.
- *F3.5 Load immediate values using MOV and MVN* on page F3-931.
- *F3.6 Load immediate values using MOV32* on page F3-934.
- *F3.7 Load immediate values using LDR Rd, =const* on page F3-935.
- *F3.8 Literal pools* on page F3-936.
- *F3.9 Load addresses into registers* on page F3-937.
- *F3.10 Load addresses to a register using ADR* on page F3-938.
- *F3.11 Load addresses to a register using ADRL* on page F3-940.
- *F3.12 Load addresses to a register using LDR Rd, =label* on page F3-941.
- *F3.13 Other ways to load and store registers* on page F3-943.
- *F3.14 Load and store multiple register instructions* on page F3-944.
- *F3.15 Load and store multiple register instructions in A32 and T32* on page F3-945.
- *F3.16 Stack implementation using LDM and STM* on page F3-946.
- *F3.17 Stack operations for nested subroutines* on page F3-948.
- *F3.18 Block copy with LDM and STM* on page F3-949.
- *F3.19 Memory accesses* on page F3-951.
- *F3.20 The Read-Modify-Write operation* on page F3-952.

- *F3.21 Optional hash with immediate constants* on page F3-953.
- *F3.22 Use of macros* on page F3-954.
- *F3.23 Test-and-branch macro example* on page F3-955.
- *F3.24 Unsigned integer division macro example* on page F3-956.
- *F3.25 Instruction and directive relocations* on page F3-958.
- *F3.26 Symbol versions* on page F3-960.
- *F3.27 Frame directives* on page F3-961.
- *F3.28 Exception tables and Unwind tables* on page F3-962.

F3.1 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 references

F1.1 --16 on page F1-847
F6.8 ARM or CODE32 directive on page F6-1031
F6.12 CODE16 directive on page F6-1035
F6.66 THUMB directive on page F6-1097
F1.2 --32 on page F1-848
F1.4 --arm on page F1-851
F1.59 --thumb on page F1-908

F3.2 Syntax differences between UAL and A64 assembly 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.

UAL in Armv8 is unchanged from Armv7.

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:

Table F3-1 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`

F3.3 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:

- 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 concepts

F3.17 Stack operations for nested subroutines on page F3-948

Related information

Procedure Call Standard for the Arm Architecture

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

F3.4 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.

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 concepts

F3.5 Load immediate values using MOV and MVN on page F3-931

F3.6 Load immediate values using MOV32 on page F3-934

F3.7 Load immediate values using LDR Rd, =const on page F3-935

Related references

F7.5 LDR pseudo-instruction on page F7-1107

F3.5 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 value shown in the Step column.

Table F3-2 A32 state immediate values (8-bit)

Binary	Decimal	Step	Hexadecimal	MVN value[a]	Notes
`000000000000000000000000abcdefgh`	0-255	1	`0-0xFF`	-1 to -256	-
`0000000000000000000000abcdefgh00`	0-1020	4	`0-0x3FC`	-4 to -1024	-
`00000000000000000000abcdefgh0000`	0-4080	16	`0-0xFF0`	-16 to -4096	-
`000000000000000000abcdefgh000000`	0-16320	64	`0-0x3FC0`	-64 to -16384	-
...	...	...	...	...	-
`abcdefgh000000000000000000000000`	0-255 x 2^{24}	2^{24}	`0-0xFF000000`	1-256 x -2^{24}	-
`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:

Table F3-3 A32 state immediate values in MOV instructions

Binary	Decimal	Step	Hexadecimal	MVN value	Notes
`0000000000000000abcdefghijklmnop`	0-65535	1	`0-0xFFFF`	-	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 value shown in the Step column.

Table F3-4 32-bit T32 immediate values

Binary	Decimal	Step	Hexadecimal	MVN value[a]	Notes
`000000000000000000000000abcdefgh`	0-255	1	`0x0-0xFF`	-1 to -256	-
`00000000000000000000000abcdefgh0`	0-510	2	`0x0-0x1FE`	-2 to -512	-
`0000000000000000000000abcdefgh00`	0-1020	4	`0x0-0x3FC`	-4 to -1024	-
...	...	...	...	...	-
`0abcdefgh00000000000000000000000`	0-255 x 2^{23}	2^{23}	`0x0-0x7F800000`	1-256 x -2^{23}	-
`abcdefgh000000000000000000000000`	0-255 x 2^{24}	2^{24}	`0x0-0xFF000000`	1-256 x -2^{24}	-
`abcdefghabcdefghabcdefghabcdefgh`	(bit pattern)	-	`0xXYXYXYXY`	`0xXYXYXYXY`	-
`00000000abcdefgh00000000abcdefgh`	(bit pattern)	-	`0x00XY00XY`	`0xFFXYFFXY`	-
`abcdefgh00000000abcdefgh00000000`	(bit pattern)	-	`0xXY00XY00`	`0xXYFFXYFF`	-
`00000000000000000000abcdefghijkl`	0-4095	1	`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 F3-5 32-bit T32 immediate values in MOV instructions

Binary	Decimal	Step	Hexadecimal	MVN value	Notes
`0000000000000000abcdefghijklmnop`	0-65535	1	`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 concepts

F3.4 Load immediate values on page F3-930

F3.6 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.
- 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 concepts

F5.5 Register-relative and PC-relative expressions on page F5-991

Related references

F7.6 MOV32 pseudo-instruction on page F7-1109

F3.7 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 concepts

F3.8 Literal pools on page F3-936

Related references

F7.5 LDR pseudo-instruction on page F7-1107

F3.8 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 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 Ox55555555
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 concepts

F3.7 Load immediate values using LDR Rd, =const on page F3-935

Related references

F6.51 LTORG on page F6-1077

F3.9 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 concepts

F3.10 Load addresses to a register using ADR on page F3-938

F3.11 Load addresses to a register using ADRL on page F3-940

F3.6 Load immediate values using MOV32 on page F3-934

F3.12 Load addresses to a register using LDR Rd, =label on page F3-941

F3.10 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.

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:

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 concepts

F3.12 Load addresses to a register using LDR Rd, =label on page F3-941

F3.11 Load addresses to a register using ADRL on page F3-940

F3.11 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` `*rn,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.

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 concepts

F3.10 Load addresses to a register using ADR on page F3-938

F3.12 Load addresses to a register using LDR Rd, =label on page F3-941

F3.12 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
        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
        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 concepts

F3.11 Load addresses to a register using ADRL on page F3-940

F3.7 Load immediate values using LDR Rd, =const on page F3-935

Related references

F7.5 LDR pseudo-instruction on page F7-1107

F6.16 DCB on page F6-1039

F3.13 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 concepts

F3.14 Load and store multiple register instructions on page F3-944
F3.15 Load and store multiple register instructions in A32 and T32 on page F3-945

F3.14 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 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 concepts

F3.15 Load and store multiple register instructions in A32 and T32 on page F3-945
F3.16 Stack implementation using LDM and STM on page F3-946
F3.17 Stack operations for nested subroutines on page F3-948
F3.18 Block copy with LDM and STM on page F3-949

F3.15 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.

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 concepts

F3.14 Load and store multiple register instructions on page F3-944

F3.16 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:

Table F3-6 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 F3-7 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 concepts

F3.14 Load and store multiple register instructions on page F3-944

Related information

Procedure Call Standard for the Arm Architecture

F3.17 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 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 concepts

F3.3 Register usage in subroutine calls on page F3-929

F3.14 Load and store multiple register instructions on page F3-944

Related information

Procedure Call Standard for the Arm Architecture

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

F3.18 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 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.

Related information

What is the fastest way to copy memory on a Cortex-A8?

F3.19 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 concepts

F4.15 Address alignment in A32/T32 code on page F4-982

F3.20 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
```

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.

F3.21 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.

F3.22 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 concepts

F3.23 Test-and-branch macro example on page F3-955

F3.24 Unsigned integer division macro example on page F3-956

Related references

F6.52 MACRO and MEND on page F6-1078

F3.23 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 concepts

F3.22 Use of macros on page F3-954

F3.24 Unsigned integer division macro example on page F3-956

F5.10 Numeric local labels on page F5-996

F3.24 Unsigned integer division macro example

You can use a macro to perform unsigned integer division.

The macro takes the following parameters:

`$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
```

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 concepts

F3.22 Use of macros on page F3-954

F3.23 Test-and-branch macro example on page F3-955

F5.10 Numeric local labels on page F5-996

F3.25 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.

`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 references

F6.7 AREA on page F6-1027

F6.28 EXPORT or GLOBAL on page F6-1051

F6.46 IMPORT and EXTERN on page F6-1071
F6.59 REQUIRE on page F6-1089
F6.58 RELOC on page F6-1088
F6.16 DCB on page F6-1039
F6.17 DCD and DCDU on page F6-1040
F6.23 DCW and DCWU on page F6-1046

Related information

ELF for the Arm Architecture

F3.26 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 references

Chapter C5 Accessing and Managing Symbols with armlink on page C5-577

Related information

Base Platform ABI for the Arm Architecture

F3.27 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.
- The assembler does not validate the information in frame directives against the instructions emitted.

Related concepts

F3.28 Exception tables and Unwind tables on page F3-962

Related references

F6.3 About frame directives on page F6-1021

Related information

Procedure Call Standard for the Arm Architecture

F3.28 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 concepts

F3.27 Frame directives on page F3-961

Related references

F6.3 About frame directives on page F6-1021

F1.26 --exceptions, --no_exceptions on page F1-875

F1.27 --exceptions_unwind, --no_exceptions_unwind on page F1-876

F6.40 FRAME UNWIND ON on page F6-1064

F6.41 FRAME UNWIND OFF on page F6-1065

F6.42 FUNCTION or PROC on page F6-1066

F6.25 ENDFUNC or ENDP on page F6-1048

Related information

Exception Handling ABI for the Arm Architecture

Chapter F4
Using armasm

Describes how to use `armasm`.

It contains the following sections:

- *F4.1 armasm command-line syntax* on page F4-964.
- *F4.2 Specify command-line options with an environment variable* on page F4-965.
- *F4.3 Using stdin to input source code to the assembler* on page F4-966.
- *F4.4 Built-in variables and constants* on page F4-967.
- *F4.5 Identifying versions of armasm in source code* on page F4-971.
- *F4.6 Diagnostic messages* on page F4-972.
- *F4.7 Interlocks diagnostics* on page F4-973.
- *F4.8 Automatic IT block generation in T32 code* on page F4-974.
- *F4.9 T32 branch target alignment* on page F4-975.
- *F4.10 T32 code size diagnostics* on page F4-976.
- *F4.11 A32 and T32 instruction portability diagnostics* on page F4-977.
- *F4.12 T32 instruction width diagnostics* on page F4-978.
- *F4.13 Two pass assembler diagnostics* on page F4-979.
- *F4.14 Using the C preprocessor* on page F4-980.
- *F4.15 Address alignment in A32/T32 code* on page F4-982.
- *F4.16 Address alignment in A64 code* on page F4-983.
- *F4.17 Instruction width selection in T32 code* on page F4-984.

F4.1 armasm command-line syntax

You can use a command line to invoke `armasm`. You must specify an input source file and you can specify various options.

The command for invoking the assembler is:

```
armasm {options} inputfile
```

where:

`options`

are 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`

is 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 means that if the command line contains options that conflict with each other, then the last option found always takes precedence.

F4.2 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 concepts

F4.1 armasm command-line syntax on page F4-964

Related information

Toolchain environment variables

F4.3 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.

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 concepts

F4.1 armasm command-line syntax on page F4-964

Related references

F1.44 --maxcache=n on page F1-893

F4.4 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 F4-1 Built-in variables

`{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 *`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. **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.
`{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 F4-2 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 F4-3 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.
`{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.

Table F4-3 Predefined macros (continued)

Name	Value	Meaning
`{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 `SMLAxy` 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`.

Table F4-3 Predefined macros (continued)

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 concepts

F4.5 Identifying versions of armasm in source code on page F4-971

Related references

F1.13 --cpu=name (armasm) on page F1-860

F1.32 --fpu=name (armasm) on page F1-881

F4.5 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 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 references

F4.4 Built-in variables and constants on page F4-967

F4.6 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 concepts

F4.7 Interlocks diagnostics on page F4-973

F4.8 Automatic IT block generation in T32 code on page F4-974

F4.9 T32 branch target alignment on page F4-975

F4.10 T32 code size diagnostics on page F4-976

F4.11 A32 and T32 instruction portability diagnostics on page F4-977

F4.12 T32 instruction width diagnostics on page F4-978

F4.13 Two pass assembler diagnostics on page F4-979

Related references

F1.17 --diag_error=tag[,tag,…] (armasm) on page F1-866

F4.7 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 concepts

F4.8 Automatic IT block generation in T32 code on page F4-974

F4.9 T32 branch target alignment on page F4-975

F4.12 T32 instruction width diagnostics on page F4-978

F4.6 Diagnostic messages on page F4-972

Related references

F1.21 --diag_warning=tag[,tag,…] (armasm) on page F1-870

F4.8 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 concepts

F4.6 Diagnostic messages on page F4-972

Related references

F1.21 --diag_warning=tag[,tag,…] (armasm) on page F1-870

F4.9 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 concepts

F4.6 Diagnostic messages on page F4-972

Related references

F1.21 --diag_warning=tag[,tag,…] (armasm) on page F1-870

F4.10 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 concepts

F4.17 Instruction width selection in T32 code on page F4-984

F4.6 Diagnostic messages on page F4-972

Related references

F1.21 --diag_warning=tag[,tag,…] (armasm) on page F1-870

F4.11 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 concepts

F4.6 Diagnostic messages on page F4-972

Related references

F1.21 --diag_warning=tag[,tag,…] (armasm) on page F1-870

F4.12 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.

Related concepts

F4.6 Diagnostic messages on page F4-972

Related references

F1.21 --diag_warning=tag[,tag,…] (armasm) on page F1-870

F4.13 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 may cause failure in the second pass.
```

Related concepts

F4.8 Automatic IT block generation in T32 code on page F4-974

F4.9 T32 branch target alignment on page F4-975

F4.12 T32 instruction width diagnostics on page F4-978

F4.6 Diagnostic messages on page F4-972

Related references

F1.21 --diag_warning=tag[,tag,…] (armasm) on page F1-870

F6.4 Directives that can be omitted in pass 2 of the assembler on page F6-1022

Related information

How the assembler works

F4.14 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.

Note

As a minimum, you must specify the `armclang` `--target` option 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 F4-4 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
```

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 references

F1.10 --cpreproc on page F1-857

F1.11 --cpreproc_opts=option[,option,...] on page F1-858

B1.49 -march on page B1-110

B1.56 -mcpu on page B1-125

B1.78 --target on page B1-161

Related information

Specifying a target architecture, processor, and instruction set

Mandatory armclang options

F4.15 Address alignment in A32/T32 code

In Armv7-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 references

F1.60 --unaligned_access, --no_unaligned_access on page F1-909

F4.16 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.

F4.17 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 concepts

F4.10 T32 code size diagnostics on page F4-976

Chapter F5

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.

It contains the following sections:

- *F5.1 Symbol naming rules* on page F5-987.
- *F5.2 Variables* on page F5-988.
- *F5.3 Numeric constants* on page F5-989.
- *F5.4 Assembly time substitution of variables* on page F5-990.
- *F5.5 Register-relative and PC-relative expressions* on page F5-991.
- *F5.6 Labels* on page F5-992.
- *F5.7 Labels for PC-relative addresses* on page F5-993.
- *F5.8 Labels for register-relative addresses* on page F5-994.
- *F5.9 Labels for absolute addresses* on page F5-995.
- *F5.10 Numeric local labels* on page F5-996.
- *F5.11 Syntax of numeric local labels* on page F5-997.
- *F5.12 String expressions* on page F5-998.
- *F5.13 String literals* on page F5-999.
- *F5.14 Numeric expressions* on page F5-1000.
- *F5.15 Syntax of numeric literals* on page F5-1001.
- *F5.16 Syntax of floating-point literals* on page F5-1002.
- *F5.17 Logical expressions* on page F5-1003.
- *F5.18 Logical literals* on page F5-1004.
- *F5.19 Unary operators* on page F5-1005.
- *F5.20 Binary operators* on page F5-1006.

- *F5.21 Multiplicative operators* on page F5-1007.
- *F5.22 String manipulation operators* on page F5-1008.
- *F5.23 Shift operators* on page F5-1009.
- *F5.24 Addition, subtraction, and logical operators* on page F5-1010.
- *F5.25 Relational operators* on page F5-1011.
- *F5.26 Boolean operators* on page F5-1012.
- *F5.27 Operator precedence* on page F5-1013.
- *F5.28 Difference between operator precedence in assembly language and C* on page F5-1014.

F5.1 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 concepts

F5.10 Numeric local labels on page F5-996

Related references

F4.4 Built-in variables and constants on page F4-967

F5.2 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 concepts

F5.14 Numeric expressions on page F5-1000

F5.12 String expressions on page F5-998

F5.3 Numeric constants on page F5-989

F5.17 Logical expressions on page F5-1003

Related references

F6.43 GBLA, GBLL, and GBLS on page F6-1067

F6.50 LCLA, LCLL, and LCLS on page F6-1076

F6.64 SETA, SETL, and SETS on page F6-1094

F5.3 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 the range -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 concepts

F5.14 Numeric expressions on page F5-1000

Related references

F5.15 Syntax of numeric literals on page F5-1001

F6.27 EQU on page F6-1050

F5.4 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.

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 references

F2.1 Syntax of source lines in armasm syntax assembly language on page F2-918

F5.1 Symbol naming rules on page F5-987

F5.5 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.
- 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 concepts

F5.6 Labels on page F5-992

Related references

F6.53 MAP on page F6-1081

F5.6 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 concepts

F5.7 Labels for PC-relative addresses on page F5-993

F5.8 Labels for register-relative addresses on page F5-994

F5.9 Labels for absolute addresses on page F5-995

Related references

F2.1 Syntax of source lines in armasm syntax assembly language on page F2-918

F6.28 EXPORT or GLOBAL on page F6-1051

F5.7 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.

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 references

F6.7 AREA on page F6-1027
F6.16 DCB on page F6-1039
F6.17 DCD and DCDU on page F6-1040
F6.19 DCFD and DCFDU on page F6-1042
F6.20 DCFS and DCFSU on page F6-1043
F6.21 DCI on page F6-1044
F6.22 DCQ and DCQU on page F6-1045
F6.23 DCW and DCWU on page F6-1046

F5.8 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 references

F6.18 DCDO on page F6-1041

F6.27 EQU on page F6-1050

F6.53 MAP on page F6-1081

F6.65 SPACE or FILL on page F6-1096

F5.9 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 concepts

F5.6 Labels on page F5-992

F5.7 Labels for PC-relative addresses on page F5-993

F5.8 Labels for register-relative addresses on page F5-994

Related references

F6.27 EQU on page F6-1050

F5.10 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.

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 concepts

F5.6 Labels on page F5-992

Related references

F2.1 Syntax of source lines in armasm syntax assembly language on page F2-918

F5.11 Syntax of numeric local labels on page F5-997

F6.52 MACRO and MEND on page F6-1078

F6.49 KEEP on page F6-1075

F6.63 ROUT on page F6-1093

F5.11 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.

`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 concepts

F5.10 Numeric local labels on page F5-996

Related references

F6.63 ROUT on page F6-1093

F5.12 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 concepts

F5.13 String literals on page F5-999

F5.19 Unary operators on page F5-1005

F5.2 Variables on page F5-988

Related references

F5.22 String manipulation operators on page F5-1008

F6.64 SETA, SETL, and SETS on page F6-1094

F5.13 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 references

F2.1 Syntax of source lines in armasm syntax assembly language on page F2-918

F1.47 --no_esc on page F1-896

F5.14 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 *F6.64 SETA, SETL, and SETS* on page F6-1094 (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.

Example

```
a   SETA    256*256            ; 256*256 is a numeric expression
    MOV     r1,#(a*22)         ; (a*22) is a numeric expression
```

Related concepts

F5.20 Binary operators on page F5-1006

F5.2 Variables on page F5-988

F5.3 Numeric constants on page F5-989

Related references

F5.15 Syntax of numeric literals on page F5-1001

F6.64 SETA, SETL, and SETS on page F6-1094

F5.15 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.

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 concepts

F5.3 Numeric constants on page F5-989

F5.16 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:

- `{-}digitsE{-}digits`
- `{-}{digits}.digits`
- `{-}{digits}.digitsE{-}digits`
- `0xhexdigits`
- `&hexdigits`
- `0f_hexdigits`
- `0d_hexdigits`

where:

`digits`

Are 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`

Are 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.

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 0.00012201070785522461.

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 concepts

F5.3 Numeric constants on page F5-989

Related references

F5.15 Syntax of numeric literals on page F5-1001

F5.17 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 references

F5.26 Boolean operators on page F5-1012

F5.25 Relational operators on page F5-1011

F5.18 Logical literals

Logical or Boolean literals can have one of two values, `{TRUE}` or `{FALSE}`.

Related concepts

F5.13 String literals on page F5-999

Related references

F5.15 Syntax of numeric literals on page F5-1001

F5.19 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 F5-1 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 F5-2 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`}.
`: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 concepts

F5.20 Binary operators on page F5-1006

F5.20 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 concepts

F5.28 Difference between operator precedence in assembly language and C on page F5-1014

Related references

F5.21 Multiplicative operators on page F5-1007

F5.22 String manipulation operators on page F5-1008

F5.23 Shift operators on page F5-1009

F5.24 Addition, subtraction, and logical operators on page F5-1010

F5.25 Relational operators on page F5-1011

F5.26 Boolean operators on page F5-1012

F5.21 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 F5-3 Multiplicative operators

Operator	Alias	Usage	Explanation
`*`		`A*B`	Multiply
`/`		`A/B`	Divide
`:MOD:`	`%`	`A:MOD:B`	A modulo B

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 concepts

F5.20 Binary operators on page F5-1006

F5.5 Register-relative and PC-relative expressions on page F5-991

F5.14 Numeric expressions on page F5-1000

Related references

F5.15 Syntax of numeric literals on page F5-1001

F5.22 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 F5-4 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 concepts

F5.12 String expressions on page F5-998

F5.14 Numeric expressions on page F5-1000

F5.23 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:

Table F5-5 Shift operators

Operator	Alias	Usage	Explanation
`:ROL:`		`A:ROL:B`	Rotate A left by B bits
`:ROR:`		`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 concepts

F5.20 Binary operators on page F5-1006

F5.24 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 F5-6 Addition, subtraction, and logical operators

Operator	Alias	Usage	Explanation
`+`		`A+B`	Add A to B
`-`		`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:`		`A:OR:B`	Bitwise OR of A and B

The use of `|` as an alias for `:OR:` is deprecated.

Related concepts

F5.20 Binary operators on page F5-1006

F5.25 Relational operators

Relational operators act on two operands of the same type to produce a logical value.

The operands can be one of:

- Numeric.
- 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 F5-7 Relational operators

Operator	Alias	Usage	Explanation
`=`	`==`	`A=B`	A equal to B
`>`		`A>B`	A greater than B
`>=`		`A>=B`	A greater than or equal to B
`<`		`A<B`	A less than B
`<=`		`A<=B`	A less than or equal to B
`/=`	`<>` `!=`	`A/=B`	A not equal to B

Related concepts

F5.20 Binary operators on page F5-1006

F5.26 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 F5-8 Boolean operators

Operator	Alias	Usage	Explanation
`:LAND:`	`&&`	`A:LAND:B`	Logical AND of A and B
`:LEOR:`		`A:LEOR:B`	Logical Exclusive OR of A and B
`:LOR:`	`||`	`A:LOR:B`	Logical OR of A and B

Related concepts

F5.20 Binary operators on page F5-1006

F5.27 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 concepts

F5.19 Unary operators on page F5-1005

F5.20 Binary operators on page F5-1006

F5.28 Difference between operator precedence in assembly language and C on page F5-1014

Related references

F5.21 Multiplicative operators on page F5-1007

F5.22 String manipulation operators on page F5-1008

F5.23 Shift operators on page F5-1009

F5.24 Addition, subtraction, and logical operators on page F5-1010

F5.25 Relational operators on page F5-1011

F5.26 Boolean operators on page F5-1012

F5.28 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.
- 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 F5-9 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 following table shows the order of precedence of operators in C.

Table F5-10 Operator precedence in C

C precedence
unary operators
`* / %`
`+ -` (as binary operators)
`<< >>`
`< <= > >=`
`== !=`
`&`
`^`
`|`
`&&`
`||`

Related concepts

F5.20 Binary operators on page F5-1006

Related references

F5.27 Operator precedence on page F5-1013

Chapter F6
armasm Directives Reference

Describes the directives that are provided by the Arm assembler, `armasm`.

It contains the following sections:

- *F6.1 Alphabetical list of directives armasm assembly language directives* on page F6-1019.
- *F6.2 About armasm assembly language control directives* on page F6-1020.
- *F6.3 About frame directives* on page F6-1021.
- *F6.4 Directives that can be omitted in pass 2 of the assembler* on page F6-1022.
- *F6.5 ALIAS* on page F6-1024.
- *F6.6 ALIGN* on page F6-1025.
- *F6.7 AREA* on page F6-1027.
- *F6.8 ARM or CODE32 directive* on page F6-1031.
- *F6.9 ASSERT* on page F6-1032.
- *F6.10 ATTR* on page F6-1033.
- *F6.11 CN* on page F6-1034.
- *F6.12 CODE16 directive* on page F6-1035.
- *F6.13 COMMON* on page F6-1036.
- *F6.14 CP* on page F6-1037.
- *F6.15 DATA* on page F6-1038.
- *F6.16 DCB* on page F6-1039.
- *F6.17 DCD and DCDU* on page F6-1040.
- *F6.18 DCDO* on page F6-1041.
- *F6.19 DCFD and DCFDU* on page F6-1042.
- *F6.20 DCFS and DCFSU* on page F6-1043.
- *F6.21 DCI* on page F6-1044.
- *F6.22 DCQ and DCQU* on page F6-1045.
- *F6.23 DCW and DCWU* on page F6-1046.

- *F6.24 END* on page F6-1047.
- *F6.25 ENDFUNC or ENDP* on page F6-1048.
- *F6.26 ENTRY* on page F6-1049.
- *F6.27 EQU* on page F6-1050.
- *F6.28 EXPORT or GLOBAL* on page F6-1051.
- *F6.29 EXPORTAS* on page F6-1053.
- *F6.30 FIELD* on page F6-1054.
- *F6.31 FRAME ADDRESS* on page F6-1055.
- *F6.32 FRAME POP* on page F6-1056.
- *F6.33 FRAME PUSH* on page F6-1057.
- *F6.34 FRAME REGISTER* on page F6-1058.
- *F6.35 FRAME RESTORE* on page F6-1059.
- *F6.36 FRAME RETURN ADDRESS* on page F6-1060.
- *F6.37 FRAME SAVE* on page F6-1061.
- *F6.38 FRAME STATE REMEMBER* on page F6-1062.
- *F6.39 FRAME STATE RESTORE* on page F6-1063.
- *F6.40 FRAME UNWIND ON* on page F6-1064.
- *F6.41 FRAME UNWIND OFF* on page F6-1065.
- *F6.42 FUNCTION or PROC* on page F6-1066.
- *F6.43 GBLA, GBLL, and GBLS* on page F6-1067.
- *F6.44 GET or INCLUDE* on page F6-1068.
- *F6.45 IF, ELSE, ENDIF, and ELIF* on page F6-1069.
- *F6.46 IMPORT and EXTERN* on page F6-1071.
- *F6.47 INCBIN* on page F6-1073.
- *F6.48 INFO* on page F6-1074.
- *F6.49 KEEP* on page F6-1075.
- *F6.50 LCLA, LCLL, and LCLS* on page F6-1076.
- *F6.51 LTORG* on page F6-1077.
- *F6.52 MACRO and MEND* on page F6-1078.
- *F6.53 MAP* on page F6-1081.
- *F6.54 MEXIT* on page F6-1082.
- *F6.55 NOFP* on page F6-1083.
- *F6.56 OPT* on page F6-1084.
- *F6.57 QN, DN, and SN* on page F6-1086.
- *F6.58 RELOC* on page F6-1088.
- *F6.59 REQUIRE* on page F6-1089.
- *F6.60 REQUIRE8 and PRESERVE8* on page F6-1090.
- *F6.61 RLIST* on page F6-1091.
- *F6.62 RN* on page F6-1092.
- *F6.63 ROUT* on page F6-1093.
- *F6.64 SETA, SETL, and SETS* on page F6-1094.
- *F6.65 SPACE or FILL* on page F6-1096.
- *F6.66 THUMB directive* on page F6-1097.
- *F6.67 TTL and SUBT* on page F6-1098.
- *F6.68 WHILE and WEND* on page F6-1099.
- *F6.69 WN and XN* on page F6-1100.

F6.1 Alphabetical list of directives armasm assembly language directives

The Arm assembler, `armasm`, provides various directives.

The following table lists them:

Table F6-1 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`	

F6.2 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`.
- `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 references

F6.52 MACRO and MEND on page F6-1078

F6.54 MEXIT on page F6-1082

F6.45 IF, ELSE, ENDIF, and ELIF on page F6-1069

F6.68 WHILE and WEND on page F6-1099

F6.3 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.
- 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 references

F6.31 FRAME ADDRESS on page F6-1055
F6.32 FRAME POP on page F6-1056
F6.33 FRAME PUSH on page F6-1057
F6.34 FRAME REGISTER on page F6-1058
F6.35 FRAME RESTORE on page F6-1059
F6.36 FRAME RETURN ADDRESS on page F6-1060
F6.37 FRAME SAVE on page F6-1061
F6.38 FRAME STATE REMEMBER on page F6-1062
F6.39 FRAME STATE RESTORE on page F6-1063
F6.40 FRAME UNWIND ON on page F6-1064
F6.41 FRAME UNWIND OFF on page F6-1065
F6.42 FUNCTION or PROC on page F6-1066
F6.25 ENDFUNC or ENDP on page F6-1048

F6.4 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`.
- `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 concepts

F4.13 Two pass assembler diagnostics on page F4-979

Related information

How the assembler works

F6.5 ALIAS

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 references

F6.28 EXPORT or GLOBAL on page F6-1051

F6.6 ALIGN

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:

- 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.
```

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 references

F6.7 AREA on page F6-1027

F6.7 AREA

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 are manipulated by the linker.

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 produced by the C compiler, or for code sections 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 to 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 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.

`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. It 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 as is required by the largest common section of each name.

`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`.

`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 the required alignment cannot be accomplished by padding with `NOP` instructions. `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. It must be defined by the source file, or a file included by the source file. 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. It 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 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`.

`READONLY`

Indicates that this section must not be written to. This is the default for Code areas.

`READWRITE`

Indicates that this section can be read from and written to. This 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 usually best placed in separate sections.

The scope of numeric local labels is defined by `AREA` directives, optionally subdivided by `ROUT` directives.

There must be at least one `AREA` directive for an assembly.

Note

`armasm` emits `R_ARM_TARGET1` relocations for the `DCD` and `DCDU` directives if the directive uses PC-relative expressions and is in any of the `PREINIT_ARRAY`, `FINI_ARRAY`, or `INIT_ARRAY` ELF sections. 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 the platform ABI permits this.

Example

The following example defines a read-only code section named `Example`:

```
    AREA    Example,CODE,READONLY   ; An example code section.
            ; code
```

Related concepts

F2.3 ELF sections and the AREA directive on page F2-921

Related references

F6.6 ALIGN on page F6-1025

F6.58 RELOC on page F6-1088

F6.17 DCD and DCDU on page F6-1040

Chapter C3 Image Structure and Generation on page C3-527

F6.8 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
```

Related references

F6.12 CODE16 directive on page F6-1035

F6.66 THUMB directive on page F6-1097

Related information

Arm Architecture Reference Manual

F6.9 ASSERT

The `ASSERT` directive generates an error message during assembly if a given assertion is false.

Syntax

`ASSERT` *`logical-expression`*

where:

`logical-expression`

is an assertion that can evaluate to either `{TRUE}` or `{FALSE}`.

Usage

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 references

F6.48 INFO on page F6-1074

F6.10 ATTR

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.

Syntax

`ATTR FILESCOPE`

`ATTR SCOPE` *`name`*

`ATTR` *`settype tagid, value`*

where:

`name`

is a section name or symbol name.

`settype`

can be any of:

- `SETVALUE`.
- `SETSTRING`.
- `SETCOMPATWITHVALUE`.
- `SETCOMPATWITHSTRING`.

`tagid`

is an attribute tag name (or its numerical value) defined in the ABI for the Arm Architecture.

`value`

depends on *`settype`*:

- is a 32-bit integer value when *`settype`* is `SETVALUE` or `SETCOMPATWITHVALUE`.
- is a nul-terminated string when *`settype`* is `SETSTRING` or `SETCOMPATWITHSTRING`.

Usage

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.

Examples

```
    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

F6.11 CN

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
```

F6.12 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`

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 references

F6.8 ARM or CODE32 directive on page F6-1031
F6.66 THUMB directive on page F6-1097

F6.13 COMMON

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.

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
```

F6.14 CP

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.

Example

```
dmu    CP  6       ; defines dmu as a symbol for
                   ; coprocessor 6
```

F6.15 DATA

The `DATA` directive is no longer required. It is ignored by the assembler.

F6.16 DCB

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 concepts

F5.14 Numeric expressions on page F5-1000

Related references

F6.17 DCD and DCDU on page F6-1040

F6.22 DCQ and DCQU on page F6-1045

F6.23 DCW and DCWU on page F6-1046

F6.65 SPACE or FILL on page F6-1096

F6.6 ALIGN on page F6-1025

F6.17 DCD and DCDU

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 concepts

F5.14 Numeric expressions on page F5-1000

Related references

F6.16 DCB on page F6-1039

F6.22 DCQ and DCQU on page F6-1045

F6.23 DCW and DCWU on page F6-1046

F6.65 SPACE or FILL on page F6-1096

F6.21 DCI on page F6-1044

F6.18 DCDO

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.
```

F6.19 DCFD and DCFDU

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.
- Minimum 2.22507385850720138e-308.

Examples

```
        DCFD    1E308,-4E-100
        DCFDU   10000,-.1,3.1E26
```

Related references

F6.20 DCFS and DCFSU on page F6-1043

F5.16 Syntax of floating-point literals on page F5-1002

F6.20 DCFS and DCFSU

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 references

F6.19 DCFD and DCFDU on page F6-1042

F5.16 Syntax of floating-point literals on page F5-1002

F6.21 DCI

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 concepts

F5.14 Numeric expressions on page F5-1000

Related references

F6.17 DCD and DCDU on page F6-1040

F6.23 DCW and DCWU on page F6-1046

F6.22 DCQ and DCQU

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.
```

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 concepts

F5.14 Numeric expressions on page F5-1000

Related references

F6.16 DCB on page F6-1039

F6.17 DCD and DCDU on page F6-1040

F6.23 DCW and DCWU on page F6-1046

F6.65 SPACE or FILL on page F6-1096

F6.23 DCW and DCWU

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 concepts

F5.14 Numeric expressions on page F5-1000

Related references

F6.16 DCB on page F6-1039

F6.17 DCD and DCDU on page F6-1040

F6.22 DCQ and DCQU on page F6-1045

F6.65 SPACE or FILL on page F6-1096

F6.24 END

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 references

F6.44 GET or INCLUDE on page F6-1068

F6.25 ENDFUNC or ENDP

The `ENDFUNC` directive marks the end of an AAPCS-conforming function. `ENDP` is a synonym for `ENDFUNC`.

Related references

F6.42 FUNCTION or PROC on page F6-1066

F6.26 ENTRY

The `ENTRY` directive declares an entry point to a program.

Syntax

`ENTRY`

Usage

A program must have an entry point. You can specify an entry point in the following ways:

- Using the `ENTRY` directive in assembly language source code.
- Providing a `main()` function in C or C++ source code.
- Using the `armlink --entry` command-line option.

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 --entry` option 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 concepts

C3.1.5 Image entry points on page C3-533

Related references

C1.41 --entry=location on page C1-382

F6.27 EQU

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:

- `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 references

F6.49 KEEP on page F6-1075

F6.28 EXPORT or GLOBAL on page F6-1051

F6.28 EXPORT or GLOBAL

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`*`}]`

where:

`symbol`

is the symbol name to export. The symbol name is case-sensitive. If *`symbol`* is omitted, all symbols are exported.

`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`

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`.

`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 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`.
- 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.

Usage

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.

Examples

```
        AREA    Example,CODE,READONLY
        EXPORT  DoAdd           ; Export the function name
                                ; to be used by external modules.
DoAdd   ADD     r0,r0,r1
```

Symbol visibility can be overridden 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.
```

The following examples show the use of the `SIZE` attribute:

```
        EXPORT symA [SIZE=4]
        EXPORT symA [DATA, SIZE=4]
```

Related references

F6.46 IMPORT and EXTERN on page F6-1071

Related information

ELF for the Arm Architecture

F6.29 EXPORTAS

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.

`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 references

F6.28 EXPORT or GLOBAL on page F6-1051

F6.30 FIELD

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`.

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 references

F6.53 MAP on page F6-1081

F6.4 Directives that can be omitted in pass 2 of the assembler on page F6-1022

Related information

How the assembler works

F6.31 FRAME ADDRESS

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.

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 references

F6.32 FRAME POP on page F6-1056
F6.33 FRAME PUSH on page F6-1057

F6.32 FRAME POP

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.

`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 references

F6.31 FRAME ADDRESS on page F6-1055
F6.35 FRAME RESTORE on page F6-1059

F6.33 FRAME PUSH

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.

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 references

F6.31 FRAME ADDRESS on page F6-1055

F6.37 FRAME SAVE on page F6-1061

F6.34 FRAME REGISTER

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.

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.

F6.35 FRAME RESTORE

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 references

F6.32 FRAME POP on page F6-1056

F6.36 FRAME RETURN ADDRESS

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:

`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.

F6.37 FRAME SAVE

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`.

Related references

F6.33 FRAME PUSH on page F6-1057

F6.38 FRAME STATE REMEMBER

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 references

F6.39 FRAME STATE RESTORE on page F6-1063

F6.42 FUNCTION or PROC on page F6-1066

F6.39 FRAME STATE RESTORE

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
```

Usage

You can only use `FRAME STATE RESTORE` within functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives.

Related references

F6.38 FRAME STATE REMEMBER on page F6-1062

F6.42 FUNCTION or PROC on page F6-1066

F6.40 FRAME UNWIND ON

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 references

F1.26 --exceptions, --no_exceptions on page F1-875

F1.27 --exceptions_unwind, --no_exceptions_unwind on page F1-876

F6.41 FRAME UNWIND OFF

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 references

F1.26 --exceptions, --no_exceptions on page F1-875

F1.27 --exceptions_unwind, --no_exceptions_unwind on page F1-876

F6.42 FUNCTION or PROC

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
       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 references

F6.39 FRAME STATE RESTORE on page F6-1063

F6.31 FRAME ADDRESS on page F6-1055

F6.6 ALIGN on page F6-1025

F6.43 GBLA, GBLL, and GBLS

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 `0xFF`, and then uses it later in a `SPACE` directive:

```
            GBLA    objectsize     ; declare the variable name
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 references

F6.50 LCLA, LCLL, and LCLS on page F6-1076
F6.64 SETA, SETL, and SETS on page F6-1094
F1.54 --predefine "directive" on page F1-903

F6.44 GET or INCLUDE

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 MS-DOS format.

Usage

`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
    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 references

F6.47 INCBIN on page F6-1073

F6.2 About armasm assembly language control directives on page F6-1020

F6.45 IF, ELSE, ENDIF, and ELIF

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
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
        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 references

F5.25 Relational operators on page F5-1011

F6.2 About armasm assembly language control directives on page F6-1020

F6.46 IMPORT and EXTERN

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}]`

where:

`directive`

can be either:

`IMPORT`

imports the symbol unconditionally.

`EXTERN`

imports the symbol only if it is referred to in the current assembly.

`symbol`

is 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`.

`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.

Usage

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

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.
    LDR     r0,=__CPP_INITIALIZE    ; If not linked, address is zeroed.
    CMP     r0,#0                   ; Test if zero.
    BEQ     nocplusplus             ; Branch on the result.
```

The following examples show the use of the `SIZE` attribute:

```
    EXTERN symA [SIZE=4]
    EXTERN symA [DATA, SIZE=4]
```

Related references

F6.28 EXPORT or GLOBAL on page F6-1051

Related information

ELF for the Arm Architecture

F6.47 INCBIN

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 MS-DOS format.

Usage

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.
```

F6.48 INFO

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 concepts

F5.12 String expressions on page F5-998

F5.14 Numeric expressions on page F5-1000

Related references

F6.9 ASSERT on page F6-1032

F6.49 KEEP

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 concepts

F5.10 Numeric local labels on page F5-996

Related references

F6.53 MAP on page F6-1081

F6.50 LCLA, LCLL, and LCLS

The `LCLA`, `LCLL`, and `LCLS` directives declare and initialize local variables.

Syntax

`lclx variable`

where:

`lclx`

is one of `LCLA`, `LCLL`, or `LCLS`.

`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 references

F6.43 GBLA, GBLL, and GBLS on page F6-1067

F6.64 SETA, SETL, and SETS on page F6-1094

F6.52 MACRO and MEND on page F6-1078

F6.51 LTORG

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.

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.
```

F6.52 MACRO and MEND

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"
```

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
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
        END
```

Related concepts

F3.22 Use of macros on page F3-954

F5.4 Assembly time substitution of variables on page F5-990

Related references

F6.54 MEXIT on page F6-1082

F6.43 GBLA, GBLL, and GBLS on page F6-1067

F6.50 LCLA, LCLL, and LCLS on page F6-1076

F6.53 MAP

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
        MAP     0xff,r9
```

Related references

F6.30 FIELD on page F6-1054

F6.4 Directives that can be omitted in pass 2 of the assembler on page F6-1022

Related information

How the assembler works

F6.54 MEXIT

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 references

F6.52 MACRO and MEND on page F6-1078

F6.55 NOFP

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.

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.

F6.56 OPT

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 F6-2 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.

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 references

F1.40 --list=file on page F1-889

F6.57 QN, DN, and SN

The `QN`, `DN`, and `SN` directives define names for Advanced SIMD and floating-point registers.

Syntax

`name directive expr``{.`*`type`*`}{[`*`x`*`]}`

where:

`directive`

is `QN`, `DN`, or `SN`.

`name`

is 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.

`type`

is any Advanced SIMD or floating-point datatype.

`[`*`x`*`]`

is only available for Advanced SIMD code. `[`*`x`*`]` is a scalar index into a register.

`type` and `[`*`x`*`]` are *Extended notation*.

Usage

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.

Examples

```
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
```

Extended notation examples

```
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]
```

F6.58 RELOC

The `RELOC` directive explicitly encodes an ELF relocation in an object file.

Syntax

`RELOC` *`n, symbol`*

`RELOC` *`n`*

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

F6.59 REQUIRE

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.

F6.60 REQUIRE8 and PRESERVE8

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.

Note

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.

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 references

F1.21 --diag_warning=tag[,tag,…] (armasm) on page F1-870

Related information

Eight-byte Stack Alignment

F6.61 RLIST

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 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}
```

F6.62 RN

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
```

F6.63 ROUT

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.

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 concepts

F5.10 Numeric local labels on page F5-996

Related references

F6.7 AREA on page F6-1027

F6.64 SETA, SETL, and SETS

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`.

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 concepts

F5.12 String expressions on page F5-998

F5.14 Numeric expressions on page F5-1000

F5.17 Logical expressions on page F5-1003

Related references

F6.43 GBLA, GBLL, and GBLS on page F6-1067

F6.50 LCLA, LCLL, and LCLS on page F6-1076

F1.54 --predefine "directive" on page F1-903

F6.65 SPACE or FILL

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 concepts

F5.14 Numeric expressions on page F5-1000

Related references

F6.6 ALIGN on page F6-1025

F6.16 DCB on page F6-1039

F6.17 DCD and DCDU on page F6-1040

F6.22 DCQ and DCQU on page F6-1045

F6.23 DCW and DCWU on page F6-1046

F6.66 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 references

F6.8 ARM or CODE32 directive on page F6-1031

F6.12 CODE16 directive on page F6-1035

F6.67 TTL and SUBT

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*`

`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.
```

F6.68 WHILE and WEND

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
```

where:

`logical-expression`

is an expression that can evaluate to either `{TRUE}` or `{FALSE}`.

Usage

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.

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 concepts

F5.17 Logical expressions on page F5-1003

Related references

F6.2 About armasm assembly language control directives on page F6-1020

F6.69 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
regname     XN 21  ; defines regname for register x21
```

Chapter F7
armasm-Specific A32 and T32 Instruction Set Features

Describes the additional support that `armasm` provides for the Arm instruction set.

It contains the following sections:

- *F7.1 armasm support for the CSDB instruction* on page F7-1102.
- *F7.2 A32 and T32 pseudo-instruction summary* on page F7-1103.
- *F7.3 ADRL pseudo-instruction* on page F7-1104.
- *F7.4 CPY pseudo-instruction* on page F7-1106.
- *F7.5 LDR pseudo-instruction* on page F7-1107.
- *F7.6 MOV32 pseudo-instruction* on page F7-1109.
- *F7.7 NEG pseudo-instruction* on page F7-1110.
- *F7.8 UND pseudo-instruction* on page F7-1111.

F7.1 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
                    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

CSDB instruction

F7.2 A32 and T32 pseudo-instruction summary

An overview of the pseudo-instructions available in the A32 and T32 instruction sets.

Table F7-1 Summary of pseudo-instructions

Mnemonic	Brief description	See
`ADRL` pseudo-instruction	Load program or register-relative address (medium range)	*F7.3 ADRL pseudo-instruction* on page F7-1104
`CPY` pseudo-instruction	Copy	*F7.4 CPY pseudo-instruction* on page F7-1106
`LDR` pseudo-instruction	Load Register pseudo-instruction	*F7.5 LDR pseudo-instruction* on page F7-1107
`MOV32` pseudo-instruction	Move 32-bit immediate to register	*F7.6 MOV32 pseudo-instruction* on page F7-1109
`NEG` pseudo-instruction	Negate	*F7.7 NEG pseudo-instruction* on page F7-1110
`UND` pseudo-instruction	Generate an architecturally undefined instruction.	*F7.8 UND pseudo-instruction* on page F7-1111

F7.3 ADRL pseudo-instruction

Load a PC-relative or register-relative address into a register.

Syntax

`ADRL{cond} Rd,label`

where:

`cond`
is an optional condition code.

`Rd`
is the register to load.

`label`
is a PC-relative or register-relative expression.

Usage

`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.

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.

Note

`ADRL` is not available in Armv6-M and Armv8-M Baseline.

Related concepts

F5.5 Register-relative and PC-relative expressions on page F5-991

F3.4 Load immediate values on page F3-930

Related references

F7.5 LDR pseudo-instruction on page F7-1107

Related information

Arm Architecture Reference Manual

F7.4 CPY pseudo-instruction

Copy a value from one register to another.

Syntax

`CPY{cond} Rd, Rm`

where:

`cond`
is an optional condition code.

`Rd`
is the destination register.

`Rm`
is 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

F7.5 LDR pseudo-instruction

Load a register with either a 32-bit immediate value or an address.

Note

This describes the `LDR` pseudo-instruction only, and not the `LDR` instruction.

Syntax

`LDR{cond}{.W} Rt, =expr`

`LDR{cond}{.W} Rt, =label_expr`

where:

cond

is an optional condition code.

.W

is an optional instruction width specifier.

Rt

is the register to be loaded.

expr

evaluates to a numeric value.

label_expr

is a PC-relative or external expression of an address in the form of a label plus or minus a numeric value.

Usage

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.
  — 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]
                             ;     ...
                             ;     litpool DCD 0xfff
        LDR     r2,=place    ; loads the address of
                             ; place into R2
                             ; =>  LDR r2,[pc,offset_to_litpool]
                             ;     ...
                             ;     litpool DCD place
```

Related concepts

F5.3 Numeric constants on page F5-989

F5.5 Register-relative and PC-relative expressions on page F5-991

F5.10 Numeric local labels on page F5-996

Related references

F1.62 --untyped_local_labels on page F1-911

F7.6 MOV32 pseudo-instruction on page F7-1109

F7.6 MOV32 pseudo-instruction

Load a register with either a 32-bit immediate value or any address.

Syntax

`MOV32{cond} Rd, expr`

where:

cond
is an optional condition code.

Rd
is 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.

Usage

`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.
- 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

F7.7 NEG pseudo-instruction

Negate the value in a register.

Syntax

`NEG{cond} Rd, Rm`

where:

`cond`
is an optional condition code.

`Rd`
is the destination register.

`Rm`
is 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.

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

F7.8 UND pseudo-instruction

Generate an architecturally undefined instruction.

Syntax

`UND{cond}{.W} {#expr}`

where:

`cond`

is an optional condition code.

`.W`

is 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 F7-2 Range and encoding of expr

Instruction	Encoding	Number of bits for *expr*	Range
A32	0xV7FYYYFY	16	0-65535
T32 32-bit encoding	0xF7FYAYFY	12	0-4095
T32 16-bit encoding	0xDEYY	8	0-255

Usage

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

Part G

Appendixes

Appendix A
Standard C Implementation Definition

Provides information required by the ISO C standard for conforming C implementations.

It contains the following sections:

- *A.1 Implementation definition* on page Appx-A-1116.
- *A.2 Translation* on page Appx-A-1117.
- *A.3 Translation limits* on page Appx-A-1118.
- *A.4 Environment* on page Appx-A-1120.
- *A.5 Identifiers* on page Appx-A-1122.
- *A.6 Characters* on page Appx-A-1123.
- *A.7 Integers* on page Appx-A-1125.
- *A.8 Floating-point* on page Appx-A-1126.
- *A.9 Arrays and pointers* on page Appx-A-1127.
- *A.10 Hints* on page Appx-A-1128.
- *A.11 Structures, unions, enumerations, and bitfields* on page Appx-A-1129.
- *A.12 Qualifiers* on page Appx-A-1130.
- *A.13 Preprocessing directives* on page Appx-A-1131.
- *A.14 Library functions* on page Appx-A-1133.
- *A.15 Architecture* on page Appx-A-1138.

A.1 Implementation definition

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 references

A.2 Translation on page Appx-A-1117
A.3 Translation limits on page Appx-A-1118
A.4 Environment on page Appx-A-1120
A.5 Identifiers on page Appx-A-1122
A.6 Characters on page Appx-A-1123
A.7 Integers on page Appx-A-1125
A.8 Floating-point on page Appx-A-1126
A.9 Arrays and pointers on page Appx-A-1127
A.10 Hints on page Appx-A-1128
A.11 Structures, unions, enumerations, and bitfields on page Appx-A-1129
A.12 Qualifiers on page Appx-A-1130
A.13 Preprocessing directives on page Appx-A-1131
A.14 Library functions on page Appx-A-1133
A.15 Architecture on page Appx-A-1138

A.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 `-Wflag`, 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.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 6 imposes no limit, other than that imposed by the available memory.

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`*
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 references

B1.8 -fbracket-depth=N on page B1-61

A.15 Architecture on page Appx-A-1138

A.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.

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()`.

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.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.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.

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`**.

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, all but the last such character are ignored.

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.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.

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.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`** and **`long double`** values are represented by IEEE double-precision values.

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 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 does not define non-standard values for `FLT_EVAL_METHOD`.

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 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 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 even".

Whether and how floating expressions are contracted when not disallowed by the `FP_CONTRACT` pragma (6.5).

If `-ffp-mode=fast`, `-ffast-math`, or `-ffp-contract=fast` options are in effect, a floating-point expression can be contracted.

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 default state of `FP_CONTRACT` pragma is `OFF`.

A.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.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.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` bit-field is treated as a signed `int` bit-field or as an unsigned `int` bit-field (6.7.2, 6.7.2.1).

Plain `int` bitfields are signed.

Allowable bit-field types other than `_Bool`, signed `int`, and `unsigned int` (6.7.2.1).

Enumeration types, `long` and `long long` (signed and unsigned) are allowed as bitfield types.

Whether atomic types are permitted for bit-fields (6.7.2.1).

Atomic types are not permitted for bitfields.

Whether a bit-field can straddle a storage-unit boundary (6.7.2.1).

A bitfield cannot straddle a storage-unit boundary.

The order of allocation of bit-fields within a unit (6.7.2.1).

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.

The alignment of non-bit-field 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.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.13 Preprocessing directives

Describes implementation-defined aspects of the Arm C compiler and C library relating to preprocessing directives, as required by the ISO C standard.

The 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.

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 via 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 should 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 *Chapter B5 Compiler-specific Pragmas* on page B5-247.

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.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).

The Arm Compiler 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`.

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`

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.

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.

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.

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 doesn't, 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].

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 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.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 is empty, 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`	`53`
`FLT_DECIMAL_DIG`	
`DBL_DECIMAL_DIG`	
`LDBL_DECIMAL_DIG`	
`DECIMAL_DIG`	`17`
`FLT_DIG`	`6`
`DBL_DIG`	`15`
`LDBL_DIG`	`15`
`FLT_MIN_EXP`	`(-125)`
`DBL_MIN_EXP`	`(-1021)`
`LDBL_MIN_EXP`	`(-1021)`
`FLT_MIN_10_EXP`	`(-37)`
`DBL_MIN_10_EXP`	`(-307)`
`LDBL_MIN_10_EXP`	`(-307)`
`FLT_MAX_EXP`	`128`
`DBL_MAX_EXP`	`1024`
`LDBL_MAX_EXP`	`1024`
`FLT_MAX_10_EXP`	`38`
`DBL_MAX_10_EXP`	`308`
`LDBL_MAX_10_EXP`	`308`
`FLT_MAX`	`3.40282347e+38F`
`DBL_MAX`	`1.79769313486231571e+308`
`LDBL_MAX`	`1.79769313486231571e+308L`

(continued)

Macro name	Value
FLT_EPSILON	1.19209290e-7F
DBL_EPSILON	2.2204460492503131e-16
LDBL_EPSILON	2.2204460492503131e-16L
FLT_MIN	1.175494351e-38F
DBL_MIN	2.22507385850720138e-308
LDBL_MIN	2.22507385850720138e-308L
FLT_TRUE_MIN	
DBL_TRUE_MIN	
LDBL_TRUE_MIN	

The values of the macros in `<limits.h>` are:

Macro name	Value
CHAR_BIT	8
SCHAR_MIN	(-128)
SCHAR_MAX	127
UCHAR_MAX	255
CHAR_MIN	0
CHAR_MAX	255
MB_LEN_MAX	6
SHRT_MIN	(-0x8000)
SHRT_MAX	0x7fff
USHRT_MAX	65535
INT_MIN	(~0x7fffffff)
INT_MAX	0x7fffffff
UINT_MAX	0xffffffffU
LONG_MIN	(~0x7fffffffL)
LONG_MIN (64-bit)	(~0x7fffffffffffffffL)
LONG_MAX	0x7fffffffL
LONG_MAX (64-bit)	0x7fffffffffffffffL
ULONG_MAX	0xffffffffUL
ULONG_MAX (64-bit)	0xffffffffffffffffUL
LLONG_MIN	(~0x7fffffffffffffffLL)
LLONG_MAX	0x7fffffffffffffffLL
ULLONG_MAX	0xffffffffffffffffULL

The values of the macros in `<stdint.h>` are:

Macro name	Value
`INT8_MIN`	`-128`
`INT8_MAX`	`127`
`UINT8_MAX`	`255`
`INT16_MIN`	`-32768`
`INT16_MAX`	`32767`
`UINT16_MAX`	`65535`
`INT32_MIN`	`(~0x7fffffff)`
`INT32_MAX`	`2147483647`
`UINT32_MAX`	`4294967295u`
`INT64_MIN`	`(~0x7fffffffffffffffLL)`
`INT32_MAX`	`2147483647`
`UINT32_MAX`	`4294967295u`
`INT64_MIN (64-bit)`	`(~0x7fffffffffffffffL)`
`INT64_MAX (64-bit)`	`(9223372036854775807L)`
`UINT64_MAX (64-bit)`	`(18446744073709551615uL)`
`INT_LEAST8_MIN`	`-128`
`INT_LEAST8_MAX`	`127`
`UINT_LEAST8_MAX`	`255`
`INT_LEAST16_MIN`	`-32768`
`INT_LEAST16_MAX`	`32767`
`UINT_LEAST16_MAX`	`65535`
`INT_LEAST32_MIN`	`(~0x7fffffff)`
`INT_LEAST32_MAX`	`2147483647`
`UINT_LEAST32_MAX`	`4294967295u`
`INT_LEAST64_MIN`	`(~0x7fffffffffffffffLL)`
`INT_LEAST64_MAX`	`(9223372036854775807LL)`
`UINT_LEAST64_MAX`	`(18446744073709551615uLL)`
`INT_LEAST64_MIN (64-bit)`	`(~0x7fffffffffffffffL)`
`INT_LEAST64_MAX (64-bit)`	`(9223372036854775807L)`
`UINT_LEAST64_MAX (64-bit)`	`(18446744073709551615uL)`
`INT_FAST8_MIN`	`(~0x7fffffff)`
`INT_FAST8_MAX`	`2147483647`
`UINT_FAST8_MAX`	`4294967295u`
`INT_FAST16_MIN`	`(~0x7fffffff)`
`INT_FAST16_MAX`	`2147483647`
`UINT_FAST16_MAX`	`4294967295u`

(continued)

Macro name	**Value**
`INT_FAST32_MIN`	`(~0x7fffffff)`
`INT_FAST32_MAX`	`2147483647`
`UINT_FAST32_MAX`	`4294967295u`
`INT_FAST64_MIN`	`(~0x7fffffffffffffffLL)`
`INT_FAST64_MAX`	`(9223372036854775807LL)`
`UINT_FAST64_MAX`	`(18446744073709551615uLL)`
`INT_FAST64_MIN (64-bit)`	`(~0x7fffffffffffffffL)`
`INT_FAST64_MAX (64-bit)`	`(9223372036854775807L)`
`UINT_FAST64_MAX (64-bit)`	`(18446744073709551615uL)`
`INTPTR_MIN`	`(~0x7fffffff)`
`INTPTR_MIN (64-bit)`	`(~0x7fffffffffffffffLL)`
`INTPTR_MAX`	`2147483647`
`INTPTR_MAX (64-bit)`	`(9223372036854775807LL)`
`UINTPTR_MAX`	`4294967295u`
`UINTPTR_MAX (64-bit)`	`(18446744073709551615uLL)`
`INTMAX_MIN`	`(~0x7fffffffffffffffll)`
`INTMAX_MAX`	`(9223372036854775807ll)`
`UINTMAX_MAX`	`(18446744073709551615ull)`
`PTRDIFF_MIN`	`(~0x7fffffff)`
`PTRDIFF_MIN (64-bit)`	`(~0x7fffffffffffffffLL)`
`PTRDIFF_MAX`	`2147483647`
`PTRDIFF_MAX (64-bit)`	`(9223372036854775807LL)`
`SIG_ATOMIC_MIN`	`(~0x7fffffff)`
`SIG_ATOMIC_MAX`	`2147483647`
`SIZE_MAX`	`4294967295u`
`SIZE_MAX (64-bit)`	`(18446744073709551615uLL)`
`WCHAR_MIN`	`0`
`WCHAR_MAX`	`0xffffffffU`
`WINT_MIN`	`(~0x7fffffff)`
`WINT_MAX`	`2147483647`

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)	16	16

Appendix B
Standard C++ Implementation Definition

Provides information required by the ISO C++ Standard for conforming C++ implementations.

It contains the following sections:

- *B.1 Implementation definition* on page Appx-B-1146.
- *B.2 General* on page Appx-B-1147.
- *B.3 Lexical conventions* on page Appx-B-1148.
- *B.4 Basic concepts* on page Appx-B-1149.
- *B.5 Standard conversions* on page Appx-B-1150.
- *B.6 Expressions* on page Appx-B-1151.
- *B.7 Declarations* on page Appx-B-1153.
- *B.8 Declarators* on page Appx-B-1154.
- *B.9 Templates* on page Appx-B-1155.
- *B.10 Exception handling* on page Appx-B-1156.
- *B.11 Preprocessing directives* on page Appx-B-1157.
- *B.12 Library introduction* on page Appx-B-1158.
- *B.13 Language support library* on page Appx-B-1159.
- *B.14 General utilities library* on page Appx-B-1160.
- *B.15 Strings library* on page Appx-B-1161.
- *B.16 Localization library* on page Appx-B-1162.
- *B.17 Containers library* on page Appx-B-1163.
- *B.18 Input/output library* on page Appx-B-1164.
- *B.19 Regular expressions library* on page Appx-B-1165.
- *B.20 Atomic operations library* on page Appx-B-1166.
- *B.21 Thread support library* on page Appx-B-1167.
- *B.22 Implementation quantities* on page Appx-B-1168.

B.1 Implementation definition

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 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.

B.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 `-Wflag`, only for messages that can be suppressed.

Libraries in a freestanding implementation (1.4).

Arm Compiler 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 references

B1.85 -W on page B1-168

B.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.

B.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 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`** and **`long double`** values are represented by IEEE double-precision values.

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.

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

B.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.

B.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 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)	16

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.

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.

B.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 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 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.

B.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.

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.

B.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.

B.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.

B.11 Preprocessing directives

Describes implementation-defined aspects of the Arm C++ compiler and C++ library relating to preprocessing directives, as required by the ISO C++ Standard.

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:
  - 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 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.

B.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.

B.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 may have to call `std::_Exit` instead of `std::exit`.

B.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`.

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.

B.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.

B.16 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.

B.17 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.

B.18 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.

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.

B.19 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
};
```

B.20 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.

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

B.21 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
```

B.22 Implementation quantities

Describes limits in C++ implementations.

Note

This topic includes descriptions of [COMMUNITY] features. See *Support level definitions* on page A1-39.

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`

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.

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`.

Appendix C
Via File Syntax

Describes the syntax of via files accepted by the `armasm`, `armlink`, `fromelf`, and `armar` tools.

It contains the following sections:

- *C.1 Overview of via files* on page Appx-C-1172.
- *C.2 Via file syntax rules* on page Appx-C-1173.

C.1 Overview of via files

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 references

C.2 Via file syntax rules on page Appx-C-1173

F1.64 --via=filename (armasm) on page F1-913

C1.159 --via=filename (armlink) on page C1-510

D1.62 --via=file (fromelf) on page D1-797

E1.31 --via=filename (armar) on page E1-837

C.2 Via file syntax rules

Via files must conform to some syntax rules.

- A via file is a text file containing a sequence of words. Each word in the text file is converted into an argument string and passed to the tool.
- Words are separated by whitespace, or the end of a line, except in delimited strings, for example:

  `--bigend --reduce_paths` (two words)

  `--bigend--reduce_paths` (one word)
- The end of a line is treated as whitespace, for example:

  ```
  --bigend
  --reduce_paths
  ```

  This is equivalent to:

  `--bigend --reduce_paths`
- 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:

  `--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 concepts

C.1 Overview of via files on page Appx-C-1172

Related references

F1.64 --via=filename (armasm) on page F1-913

C1.159 --via=filename (armlink) on page C1-510

D1.62 --via=file (fromelf) on page D1-797

E1.31 --via=filename (armar) on page E1-837