

ARM[®] Compiler

Version 6.00

armlink User Guide



ARM Compiler

armlink User Guide

Copyright © 2014 ARM. All rights reserved.

Release Information

The following changes have been made to this book.

Change History			
Date	Issue	Confidentiality	Change
14 March 2014	A	Non-Confidential	ARM Compiler v6.00 Release

Proprietary Notice

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

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

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

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

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

Confidentiality Status

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

Product Status

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

Web Address

<http://www.arm.com>

Contents

ARM Compiler armlink User Guide

Chapter 1	Conventions and feedback	
Chapter 2	Overview of the linker	
2.1	About the linker	2-2
2.2	Linker command-line syntax	2-4
2.3	Linker command-line options listed in groups	2-5
2.4	What the linker can accept as input	2-9
2.5	What the linker outputs	2-10
2.6	What the linker does when constructing an executable image	2-11
Chapter 3	Linking models supported by armlink	
3.1	Overview of linking models	3-2
3.2	Bare-metal linking model	3-3
3.3	Partial linking model	3-4
3.4	Base Platform Application Binary Interface (BPABI) linking model	3-5
3.5	Base Platform linking model	3-6
Chapter 4	Image structure and generation	
4.1	The image structure	4-3
4.2	Input sections, output sections, regions, and Program Segments	4-5
4.3	Load view and execution view of an image	4-6
4.4	Methods of specifying an image memory map with the linker	4-7
4.5	Types of simple image	4-9
4.6	Type 1 image, one load region and contiguous execution regions	4-10
4.7	Type 2 image, one load region and non-contiguous execution regions	4-12
4.8	Type 3 image, two load regions and non-contiguous execution regions	4-14
4.9	Image entry points	4-16
4.10	About specifying an initial entry point	4-17
4.11	Section placement with the linker	4-18

4.12	Placing sections with FIRST and LAST attributes	4-20
4.13	Section alignment with the linker	4-21
4.14	Demand paging	4-22
4.15	About ordering execution regions containing T32 code	4-24
4.16	Overview of veneers	4-25
4.17	Veneer sharing	4-26
4.18	Veneer types	4-27
4.19	Generation of position independent to absolute veneers	4-28
4.20	Reuse of veneers when scatter-loading	4-29
4.21	About weak references and definitions	4-30
4.22	How the linker performs library searching, selection, and scanning	4-33
4.23	Controlling how the linker searches for the ARM standard libraries	4-34
4.24	Specifying user libraries when linking	4-36
4.25	How the linker resolves references	4-37
4.26	Use of the strict family of options in the linker	4-38
Chapter 5	Using linker optimizations	
5.1	Elimination of common debug sections	5-2
5.2	Elimination of common groups or sections	5-3
5.3	Elimination of unused sections	5-4
5.4	Elimination of unused virtual functions	5-5
5.5	Optimization with RW data compression	5-6
5.6	How the linker chooses a compressor	5-7
5.7	Overriding the compression algorithm used by the linker	5-8
5.8	How compression is applied	5-9
5.9	Working with RW data compression	5-10
5.10	Inlining functions with the linker	5-11
5.11	Factors that influence function inlining	5-12
5.12	Handling branches that optimize to a NOP	5-14
5.13	About reordering of tail calling sections	5-15
5.14	Restrictions on reordering of tail calling sections	5-16
5.15	About merging comment sections	5-17
Chapter 6	Getting information about images when linking	
6.1	Linker options for getting information about images	6-2
6.2	Identifying the source of some link errors	6-3
6.3	Example of using the --info linker option	6-4
6.4	How to find where a symbol is placed when linking	6-6
Chapter 7	Accessing and managing symbols with armlink	
7.1	About mapping symbols	7-3
7.2	Accessing linker-defined symbols	7-4
7.3	Region-related symbols	7-5
7.4	Image\$\$ execution region symbols	7-6
7.5	Load\$\$ execution region symbols	7-7
7.6	Load\$\$LR\$\$ load region symbols	7-9
7.7	Region name values when not scatter-loading	7-10
7.8	Linker defined symbols and scatter files	7-11
7.9	Importing linker-defined symbols in C and C++	7-12
7.10	Importing linker-defined symbols in ARM assembly language	7-13
7.11	Section-related symbols	7-14
7.12	Image symbols	7-15
7.13	Input section symbols	7-16
7.14	Accessing symbols in another image	7-17
7.15	Creating a symdefs file	7-18
7.16	Outputting a subset of the global symbols	7-19
7.17	Reading a symdefs file	7-20
7.18	Symdefs file format	7-21
7.19	What is a steering file?	7-23
7.20	Specifying steering files on the linker command-line	7-24

7.21	Steering file command summary	7-25
7.22	Steering file format	7-26
7.23	About hiding and renaming global symbols with a steering file	7-27
7.24	About using \$Super\$\$ and \$Sub\$\$ to patch symbol definitions	7-28

Chapter 8

Using scatter files

8.1	About scatter-loading	8-3
8.2	When to use scatter-loading	8-4
8.3	Scatter-loading command-line option	8-5
8.4	Images with a simple memory map	8-6
8.5	Images with a complex memory map	8-8
8.6	Linker-defined symbols that are not defined when scatter-loading	8-10
8.7	Specifying stack and heap using the scatter file	8-11
8.8	What is a root region?	8-12
8.9	Creating root execution regions	8-13
8.10	Using the FIXED attribute to create root regions	8-16
8.11	Methods of placing functions and data at specific addresses	8-17
8.12	Placing a variable at a specific address without scatter-loading	8-18
8.13	Placing a variable in a named section with scatter-loading	8-20
8.14	Placing a variable at a specific address with scatter-loading	8-22
8.15	Explicit placement of a named section with scatter-loading	8-24
8.16	Placement of unassigned sections with the .ANY module selector	8-25
8.17	Prioritization of .ANY sections	8-27
8.18	Command-line options for controlling the placement of input sections for multiple .ANY selectors	8-28
8.19	Specifying the maximum size permitted for placing unassigned sections	8-29
8.20	Examples of using placement algorithms for .ANY sections	8-31
8.21	Example of next_fit algorithm showing behavior of full regions, selectors, and priority ..	8-33
8.22	Examples of using sorting algorithms for .ANY sections	8-35
8.23	Selecting veneer input sections in scatter-loading descriptions	8-37
8.24	Placement of code and data with __attribute__((section("name")))	8-38
8.25	Placement of sections at a specific address with __attribute__((section(".ARM.__at_address")))	8-40
8.26	Restrictions on placing __at sections	8-41
8.27	Automatic placement of __at sections	8-42
8.28	Manual placement of __at sections	8-44
8.29	Placement of a key in flash memory with __at	8-46
8.30	Placement of sections with overlays	8-48
8.31	About placing ARM C and C++ library code	8-51
8.32	Example of placing code in a root region	8-52
8.33	Example of placing ARM C library code	8-53
8.34	Example of placing ARM C++ library code	8-54
8.35	Reserving an empty region	8-55
8.36	About creating regions on page boundaries	8-57
8.37	Overalignment of execution regions and input sections	8-59
8.38	Expression evaluation in scatter files	8-60
8.39	Using expression evaluation in a scatter file to avoid padding	8-61
8.40	Equivalent scatter-loading descriptions for simple images	8-62
8.41	Type 1 image, one load region and contiguous execution regions	8-63
8.42	Type 2 image, one load region and non-contiguous execution regions	8-65
8.43	Type 3 image, two load regions and non-contiguous execution regions	8-67
8.44	Scatter file to ELF mapping	8-69

Chapter 9

Base Platform Application Binary Interface Support

9.1	About the Base Platform Application Binary Interface (BPABI)	9-3
9.2	Platforms supported by the BPABI	9-4
9.3	Concepts common to all BPABI models	9-5
9.4	About importing and exporting symbols for BPABI models	9-6
9.5	Symbol visibility for BPABI models	9-7

9.6	Automatic import and export for BPABI models	9-8
9.7	Manual import and export for BPABI models	9-9
9.8	Symbol versioning for BPABI models	9-10
9.9	RW compression for BPABI models	9-11
9.10	Linker options for bare metal and DLL-like models	9-12
9.11	Bare metal and DLL-like memory model	9-13
9.12	Mandatory symbol versioning in the BPABI DLL-like model	9-14
9.13	Automatic dynamic symbol table rules in the BPABI DLL-like model	9-15
9.14	Addressing modes in the BPABI DLL-like model	9-16
9.15	C++ initialization in the BPABI DLL-like model	9-17
9.16	About symbol versioning	9-18
9.17	Symbol versioning script file	9-19
9.18	Example of creating versioned symbols	9-20
9.19	About embedded symbols	9-21
9.20	Linker options for enabling implicit symbol versioning	9-22
9.21	Related linker command-line options for the BPABI DLL-like model	9-23

Chapter 10

Features of the Base Platform linking model

10.1	Restrictions on the use of scatter files with the Base Platform model	10-2
10.2	Example scatter file for the Base Platform linking model	10-5
10.3	Placement of PLT sequences with the Base Platform model	10-7

Chapter 1

Conventions and feedback

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

Typographical conventions

The following typographical conventions are used:

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

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

`monospace` *italic*

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

`monospace` **bold**

Denotes language keywords when used outside example code.

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

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

Feedback on this product

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

- your name and company

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

Feedback on content

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

- the title
- the number, ARM DUI 0803A
- if viewing online, the topic names to which your comments apply
- if viewing a PDF version of a document, the page numbers to which your comments apply
- a concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

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

Other information

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

Chapter 2

Overview of the linker

The following topics give an overview of the ARM linker, `armlink`:

Concepts

- [About the linker on page 2-2.](#)
- [What the linker can accept as input on page 2-9.](#)
- [What the linker outputs on page 2-10.](#)
- [What the linker does when constructing an executable image on page 2-11.](#)

Reference

- [Linker command-line syntax on page 2-4.](#)
- [Linker command-line options listed in groups on page 2-5.](#)

2.1 About the linker

The linker, `arm1ink`, combines the contents of one or more object files with selected parts of one or more object libraries to produce:

- An ARM ELF image.
- A partially linked ELF object that can be used as input in a subsequent link step.
- ELF files that can be demand-paged efficiently.
- A shared object, compatible with the *Base Platform Application Binary Interface* (BPABI) or a BPABI executable file.

The linker can:

- Link A32 code and T32 code.
- Link A64 code.
- Generate interworking veneers to switch between A32 and T32 states when required.
- Generate inline veneers or long branch veneers, where required, to extend the range of branch instructions.
- Automatically select the appropriate standard C or C++ library variants to link with, based on the build attributes of the objects it is linking.
- Enable you to specify the locations of code and data within the system memory map, using either a command-line option or a scatter file.
- Perform Read/Write data compression to minimize ROM size.
- Perform unused section elimination to reduce the size of your output image.
- Control the generation of debug information in the output file.
- Generate a static callgraph and list the stack usage.
- Control the contents of the symbol table in output images.
- Show the sizes of code and data in the output.
- Accept GNU ld scripts when linking code for AArch32 state, with restrictions.

2.1.1 See also

Concepts

- [Demand paging on page 4-22.](#)
- [Chapter 3 Linking models supported by armlink.](#)
- [Chapter 4 Image structure and generation.](#)
- [Chapter 5 Using linker optimizations.](#)
- [Chapter 6 Getting information about images when linking.](#)
- [Chapter 7 Accessing and managing symbols with armlink.](#)
- [Chapter 8 Using scatter files.](#)
- [Chapter 9 Base Platform Application Binary Interface Support.](#)
- [Chapter 10 Features of the Base Platform linking model.](#)

Getting Started Guide:

- *ARM Compiler and virtual address space*
<http://infocenter.arm.com/help/topic/com.arm.doc.dui0741-/chr1372757164575.html>.

Other information

Base Platform ABI for the ARM Architecture

<http://infocenter.arm.com/help/topic/com.arm.doc.ihl0037-/index.html>.

2.2 Linker command-line syntax

The command for invoking the linker is:

```
armlink [options] [input-file-list]
```

options Linker command-line options.

input-file-list

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

2.2.1 See also

Reference

- [Linker command-line options listed in groups on page 2-5.](#)
- armlink Reference Guide:*
- [input-file-list on page 2-68.](#)
 - [Chapter 2 Linker command-line options.](#)

2.3 Linker command-line options listed in groups

See the following command-line options in the *armlink Reference Guide*:

Controlling library files and paths

- [--force_scanlib](#) on page 2-55.
- [--libpath](#) on page 2-77.
- [--library](#) on page 2-78.
- [--library_type](#) on page 2-79.
- [--scanlib](#), [--no_scanlib](#) on page 2-109.
- [--thumb2_library](#), [--no_thumb2_library](#) on page 2-133.
- [--userlibpath](#) on page 2-139.

Controlling the linking of object files

- [--match=crossmangled](#) on page 2-87.
- [--strict](#) on page 2-120.
- [--strict_ph](#), [--no_strict_ph](#) on page 2-123.
- [--strict_relocations](#), [--no_strict_relocations](#) on page 2-124.
- [--unresolved](#) on page 2-137.

Controlling the output

- [--base_platform](#) on page 2-12.
- [--bpabi](#) on page 2-16.
- [--combreloc](#), [--no_combreloc](#) on page 2-25.
- [--dll](#) on page 2-38.
- [--output](#) on page 2-92.
- [--partial](#) on page 2-97.
- [--reloc](#) on page 2-103.

Specifying the image memory map

- [--autoat](#), [--no_autoat](#) on page 2-11.
- [--ro_base](#) on page 2-106.
- [--rosplit](#) on page 2-107.
- [--rw_base](#) on page 2-108.
- [--scatter](#) on page 2-110.
- [--split](#) on page 2-118.
- [--zi_base](#) on page 2-152.

Controlling debug information in an image

- [--bestdebug](#), [--no_bestdebug](#) on page 2-13.
- [--compress_debug](#), [--no_compress_debug](#) on page 2-27.
- [--debug](#), [--no_debug](#) on page 2-32.
- [--emit_debug_overlay_relocs](#) on page 2-42.
- [--emit_debug_overlay_section](#) on page 2-43.
- [--emit_non_debug_relocs](#) on page 2-44.

Controlling the content of an image

- [--any_contingency](#) on page 2-5.
- [--any_placement](#) on page 2-6.
- [--any_sort_order](#) on page 2-8.
- [--api](#), [--no_api](#) on page 2-9.

- `--arm_only` on page 2-10.
- `--blx_arm_thumb`, `--no_blx_arm_thumb` on page 2-14.
- `--blx_thumb_arm`, `--no_blx_thumb_arm` on page 2-15.
- `--branchnop`, `--no_branchnop` on page 2-17.
- `--comment_section`, `--no_comment_section` on page 2-26.
- `--cppinit`, `--no_cppinit` on page 2-28.
- `--cpu` on page 2-29.
- `--datacompressor` on page 2-31.
- `--dynamic_linker` on page 2-39.
- `--edit` on page 2-41.
- `--emit_relocs` on page 2-45.
- `--entry` on page 2-46.
- `--exceptions`, `--no_exceptions` on page 2-48.
- `--export_all`, `--no_export_all` on page 2-49.
- `--export_dynamic`, `--no_export_dynamic` on page 2-50.
- `--filtercomment`, `--no_filtercomment` on page 2-51.
- `--fini` on page 2-52.
- `--first` on page 2-53.
- `--force_explicit_attr` on page 2-54.
- `--force_so_throw`, `--no_force_so_throw` on page 2-56.
- `--fpu` on page 2-57.
- `--gnu_linker_defined_syms` on page 2-58.
- `--import_unresolved`, `--no_import_unresolved` on page 2-60.
- `--init=symbol` on page 2-65.
- `--inline`, `--no_inline` on page 2-66.
- `--keep` on page 2-70.
- `--keep_protected_symbols` on page 2-72.
- `--largeregions`, `--no_largeregions` on page 2-73.
- `--last` on page 2-75.
- `--locals`, `--no_locals` on page 2-84.
- `--max_visibility` on page 2-89.
- `--merge`, `--no_merge` on page 2-90.
- `--muldefweak`, `--no_muldefweak` on page 2-91.
- `--override_visibility` on page 2-93.
- `--pad` on page 2-94.
- `--paged` on page 2-95.
- `--pagesize` on page 2-96.
- `--pltgot` on page 2-99.
- `--pltgot_opts` on page 2-100.
- `--privacy` on page 2-101.
- `--ref_cpp_init`, `--no_ref_cpp_init` on page 2-102.
- `--remove`, `--no_remove` on page 2-105.
- `--sort` on page 2-116.
- `--startup`, `--no_startup` on page 2-119.
- `--strict_flags`, `--no_strict_flags` on page 2-122.
- `--symver_script` on page 2-130.
- `--symver_soname` on page 2-131.

- [--tailreorder](#), [--no_tailreorder](#) on page 2-132.
- [--tiebreaker](#) on page 2-134.
- [--undefined](#) on page 2-135.
- [--undefined_and_export](#) on page 2-136.
- [--use_definition_visibility](#) on page 2-138.
- [--vfemode](#) on page 2-146.

Controlling veneer generation

- [--crosser_veneershare](#), [--no_crosser_veneershare](#) on page 2-30.
- [--inlineveneer](#), [--no_inlineveneer](#) on page 2-67.
- [--max_veneer_passes](#) on page 2-88.
- [--piveneer](#), [--no_piveneer](#) on page 2-98.
- [--veneerinject](#), [--no_veneerinject](#) on page 2-140.
- [--veneer_inject_type](#) on page 2-141.
- [--veneer_pool_size](#) on page 2-142.
- [--veneershare](#), [--no_veneershare](#) on page 2-143.

Controlling the extraction and presentation of image information

- [--callgraph](#), [--no_callgraph](#) on page 2-18.
- [--callgraph_file](#) on page 2-20.
- [--callgraph_output](#) on page 2-21.
- [--cgfile](#) on page 2-22.
- [--cgsymbol](#) on page 2-23.
- [--cgundefined](#) on page 2-24.
- [--info](#) on page 2-61.
- [--info_lib_prefix](#) on page 2-64.
- [--list_mapping_symbols](#), [--no_list_mapping_symbols](#) on page 2-82.
- [--load_addr_map_info](#), [--no_load_addr_map_info](#) on page 2-83.
- [--mangled](#), [--unmangled](#) on page 2-85.
- [--map](#), [--no_map](#) on page 2-86.
- [--section_index_display](#) on page 2-111.
- [--symbols](#), [--no_symbols](#) on page 2-128.
- [--symdefs](#) on page 2-129.
- [--xref](#), [--no_xref](#) on page 2-149.
- [--xrefdbg](#), [--no_xrefdbg](#) on page 2-150.
- [--xref{from|to}](#) on page 2-151.

———— Note ————

With the exception of `--callgraph`, the linker prints the information you request on the standard output stream, `stdout`, by default. You can redirect the information to a text file using the `--list` command-line option.

Controlling diagnostic messages

- [--diag_error](#) on page 2-33.
- [--diag_remark](#) on page 2-34.
- [--diag_style](#) on page 2-35.
- [--diag_suppress](#) on page 2-36.
- [--diag_warning](#) on page 2-37.
- [--errors](#) on page 2-47.

- [--list](#) on page 2-81.
- [--remarks](#) on page 2-104.
- [--show_full_path](#) on page 2-113.
- [--show_parent_lib](#) on page 2-114.
- [--show_sec_idx](#) on page 2-115.
- [--strict_enum_size](#), [--no_strict_enum_size](#) on page 2-121.
- [--strict_symbols](#), [--no_strict_symbols](#) on page 2-125.
- [--strict_visibility](#), [--no_strict_visibility](#) on page 2-126.
- [--strict_wchar_size](#), [--no_strict_wchar_size](#) on page 2-127.
- [--verbose](#) on page 2-144.

Controlling alignment in legacy images

- [--legacyalign](#), [--no_legacyalign](#) on page 2-76.

Miscellaneous

- [--eager_load_debug](#), [--no_eager_load_debug](#) on page 2-40.
- [--licretry](#) on page 2-80.
- [--show_cmdline](#) on page 2-112.
- [--version_number](#) on page 2-145.
- [--via=filename](#) on page 2-147.
- [--vsn](#) on page 2-148.

2.4 What the linker can accept as input

Input to `armlink` consists of one or more object files in ARM ELF. This format is described in the following documents:

- *ELF for the ARM 64-bit Architecture (AArch64) (ARM IHI 0056)*.
- *ELF for the ARM Architecture (ARM IHI 0044)*, for AArch32.

Optionally, the following files can be used as input to `armlink`:

- One or more libraries created by the librarian, `armar`.
- A symbol definitions file.
- A scatter file.
- A steering file.

2.4.1 See also

Tasks

- [Chapter 8 Using scatter files.](#)

armar User Guide:

- [Creating a new object library on page 3-2.](#)

Reference

- [Accessing symbols in another image on page 7-17.](#)

armlink Reference Guide:

- [Chapter 3 Linker steering file command reference.](#)
- [Chapter 4 Formal syntax of the scatter file.](#)

Other information

- *ELF for the ARM 64-bit Architecture (AArch64) (ARM IHI 0056)*
<http://infocenter.arm.com/help/topic/com.arm.doc.ih0056-/index.html>.
- *ELF for the ARM Architecture (ARM IHI 0044)*
<http://infocenter.arm.com/help/topic/com.arm.doc.ih0044-/index.html>.

2.5 What the linker outputs

Output from `arm1ink` can be:

- An ELF executable image.
- An ELF shared object.
- A partially-linked ELF object.
- A relocatable ELF object.

You can use `fromelf` to convert an ELF executable image to other file formats, or to display, process, and protect the content of an ELF executable image.

2.5.1 See also

Concepts

- [Partial linking model](#) on page 3-4.
- [Section placement with the linker](#) on page 4-18.
- [The image structure](#) on page 4-3.

fromelf User Guide:

- [Chapter 2 Overview of the fromelf image converter](#).

2.6 What the linker does when constructing an executable image

When you use the linker to construct an executable image, it:

- Resolves symbolic references between the input object files.
- Extracts object modules from libraries to satisfy otherwise unsatisfied symbolic references.
- Sorts input sections according to their attributes and names, and merges sections with similar attributes and names into contiguous chunks.
- Removes unused sections.
- Eliminates duplicate common groups and common code, data, and debug sections.
- Organizes object fragments into memory regions according to the grouping and placement information provided.
- Assigns addresses to relocatable values.
- Generates an executable image.

2.6.1 See also

Tasks

- [Elimination of common debug sections on page 5-2.](#)
- [Elimination of unused sections on page 5-4.](#)

Concepts

- [The image structure on page 4-3.](#)

Chapter 3

Linking models supported by armlink

The following topics describe the linking models supported by the ARM linker, armlink:

Concepts

- [Overview of linking models on page 3-2.](#)
- [Bare-metal linking model on page 3-3.](#)
- [Partial linking model on page 3-4.](#)
- [Base Platform Application Binary Interface \(BPABI\) linking model on page 3-5.](#)
- [Base Platform linking model on page 3-6.](#)

3.1 Overview of linking models

A linking model is a group of command-line options and memory maps that control the behavior of the linker.

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.

Partial linking

This model produces a platform-independent object suitable for input to the linker in a subsequent link step. It can be used as an intermediate step in the development process and 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 BPABI specification.

———— **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.

Related options in each model can be combined to tighten control over the output.

———— **Note** —————

ARM Compiler 6 is a bare metal compiler only. Other linking models are not supported.

3.1.1 See also

Concepts

- [Bare-metal linking model on page 3-3.](#)
- [Partial linking model on page 3-4.](#)
- [Base Platform Application Binary Interface \(BPABI\) linking model on page 3-5.](#)
- [Base Platform linking model on page 3-6.](#)

Reference

- [Chapter 9 Base Platform Application Binary Interface Support.](#)

Other information

- [Base Platform ABI for the ARM Architecture](#)
<http://infocenter.arm.com/help/topic/com.arm.doc.ih0037-/index.html>.

3.2 Bare-metal linking model

The bare-metal model focuses on the conventional embedded market where the whole program, possibly including a *Real-Time Operating System* (RTOS), is linked in one pass. Very few assumptions can be made by the linker about the memory map of a bare metal system. Therefore, you must use the scatter-loading mechanism if you want more precise control.

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. This can be achieved using a dynamic model.

With this type of 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.
- Identify the shared libraries that are required by the ELF file using a steering file.

You can use the following options with this model:

- `--edit=file_list`
- `--scatter=file`.

You can use the following options when scatter-loading is not used:

- `--reloc`.
- `--ro_base=address`.
- `--rosplit`.
- `--rw_base=address`.
- `--split`.
- `--zi_base`.

3.2.1 See also

Concepts

- [Methods of specifying an image memory map with the linker on page 4-7.](#)

Reference

armlink Reference Guide:

- [--edit on page 2-41.](#)
- [--reloc on page 2-103.](#)
- [--ro_base on page 2-106.](#)
- [--rosplit on page 2-107.](#)
- [--rw_base on page 2-108.](#)
- [--scatter on page 2-110.](#)
- [--split on page 2-118.](#)
- [--zi_base on page 2-152.](#)
- [Chapter 3 Linker steering file command reference.](#)

3.3 Partial linking model

Partial linking:

- Eliminates duplicate copies of debug sections.
- Merges the symbol tables into one.
- Leaves unresolved references unresolved.
- Merges common data (COMDAT) groups.
- Generates an object that can be used as input to a subsequent link step.

A single output file is produced 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 output file can have only one entry point.

To link with this model, use the `--partial` command-line option. Other linker command-line options supported by this model are:

- `--edit=`*file_list*.
- `--exceptions_tables=`*action*.

Note

If you use partial linking, you cannot refer to the component objects by name in a scatter file. Therefore, you might have to update your scatter file.

3.3.1 See also

Concepts

- [What is a steering file? on page 7-23.](#)

Reference

- [Steering file format on page 7-26.](#)

armlink Reference Guide:

- [--edit on page 2-41.](#)
- [--partial on page 2-97.](#)
- [Chapter 3 Linker steering file command reference.](#)

3.4 Base Platform Application Binary Interface (BPABI) linking model

The *Base Platform Application Binary Interface* (BPABI) is a meta-standard for third parties to generate their own platform-specific image formats. This means that the BPABI model produces as much information as possible without focusing on any specific platform.

Note

BPABI is not supported for AArch64 state.

Be aware of the following:

- You cannot use scatter-loading. However, the Base Platform linking model is an extension to the BPABI model that supports scatter-loading.
- The model assumes that shared objects cannot throw a C++ exception.
- 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.

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`.

3.4.1 See also

Concepts

- [Base Platform linking model on page 3-6.](#)
- [About symbol versioning on page 9-18.](#)

Reference

armlink Reference Guide:

- [--bpabi on page 2-16.](#)
- [--dll on page 2-38.](#)
- [--force_so_throw, --no_force_so_throw on page 2-56.](#)
- [--pltgot on page 2-99.](#)
- [--ro_base on page 2-106.](#)
- [--rosplit on page 2-107.](#)
- [--rw_base on page 2-108.](#)

Other information

- *Base Platform ABI for the ARM Architecture*
<http://infocenter.arm.com/help/topic/com.arm.doc.ih0037-/index.html>.

3.5 Base Platform linking model

Base Platform 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. It 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`, in addition to the following options:
 - `--dll`.
 - `--force_so_throw`, `--no_force_so_throw`.
 - `--pltgot=type` is restricted to types `none` or `direct`.
 - `--ro_base=address`.
 - `--rosplit`.
 - `--rw_base=address`.
- 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 do not use a scatter file, the linker can ensure that the *Procedure Linkage Table* (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.

Each load region containing code might require a 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.

To ensure calls between relocated load regions at run-time:

- Use the `--pltgot=direct` option to turn on PLT generation.
- Use the `--pltgot_opts=crosslr` option to add entries in the PLT for calls between 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 assumes that shared objects cannot throw a C++ exception.

- 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.

3.5.1 See also

Concepts

- [Base Platform Application Binary Interface \(BPABI\) linking model](#) on page 3-5.
- [Methods of specifying an image memory map with the linker](#) on page 4-7.
- [About symbol versioning](#) on page 9-18.
- [Restrictions on the use of scatter files with the Base Platform model](#) on page 10-2.
- [Example scatter file for the Base Platform linking model](#) on page 10-5.

Reference

armlink Reference Guide:

- [--base_platform](#) on page 2-12.
- [--dll](#) on page 2-38.
- [--force_so_throw, --no_force_so_throw](#) on page 2-56.
- [--pltgot](#) on page 2-99.
- [--pltgot_opts](#) on page 2-100.
- [--ro_base](#) on page 2-106.
- [--rosplit](#) on page 2-107.
- [--rw_base](#) on page 2-108.
- [--scatter](#) on page 2-110.

Chapter 4

Image structure and generation

The following topics describe the image structure and the functionality available in the ARM linker, `arm1ink`, to generate images:

Tasks

- *Controlling how the linker searches for the ARM standard libraries on page 4-34.*
- *Specifying user libraries when linking on page 4-36.*

Concepts

- *The image structure on page 4-3.*
- *Input sections, output sections, regions, and Program Segments on page 4-5.*
- *Load view and execution view of an image on page 4-6.*
- *Methods of specifying an image memory map with the linker on page 4-7.*
- *Types of simple image on page 4-9.*
- *Type 1 image, one load region and contiguous execution regions on page 4-10.*
- *Type 2 image, one load region and non-contiguous execution regions on page 4-12.*
- *Type 3 image, two load regions and non-contiguous execution regions on page 4-14.*
- *Image entry points on page 4-16.*
- *About specifying an initial entry point on page 4-17.*
- *Section placement with the linker on page 4-18.*
- *Placing sections with FIRST and LAST attributes on page 4-20.*
- *Section alignment with the linker on page 4-21.*
- *Demand paging on page 4-22.*

- *About ordering execution regions containing T32 code on page 4-24.*
- *Overview of veneers on page 4-25.*
- *Veneer sharing on page 4-26.*
- *Veneer types on page 4-27.*
- *Generation of position independent to absolute veneers on page 4-28.*
- *Reuse of veneers when scatter-loading on page 4-29.*
- *About weak references and definitions on page 4-30.*
- *How the linker performs library searching, selection, and scanning on page 4-33.*
- *How the linker resolves references on page 4-37.*
- *Use of the strict family of options in the linker on page 4-38.*

4.1 The image structure

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.

Each link stage has a different view of the image:

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.
- An executable file that holds a program suitable for execution.
- A shared object file that holds code and data in the following contexts:
 - The linker processes the file with other relocatable and shared object files to create another object file.
 - The dynamic linker combines the file with an executable file and other shared objects to create a process image.

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 corresponds to up to three output sections:
 - RO section.
 - RW section.
 - ZI section.

One or more execution regions make up a load region.

———— Note ————

With `arm1link`, 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:

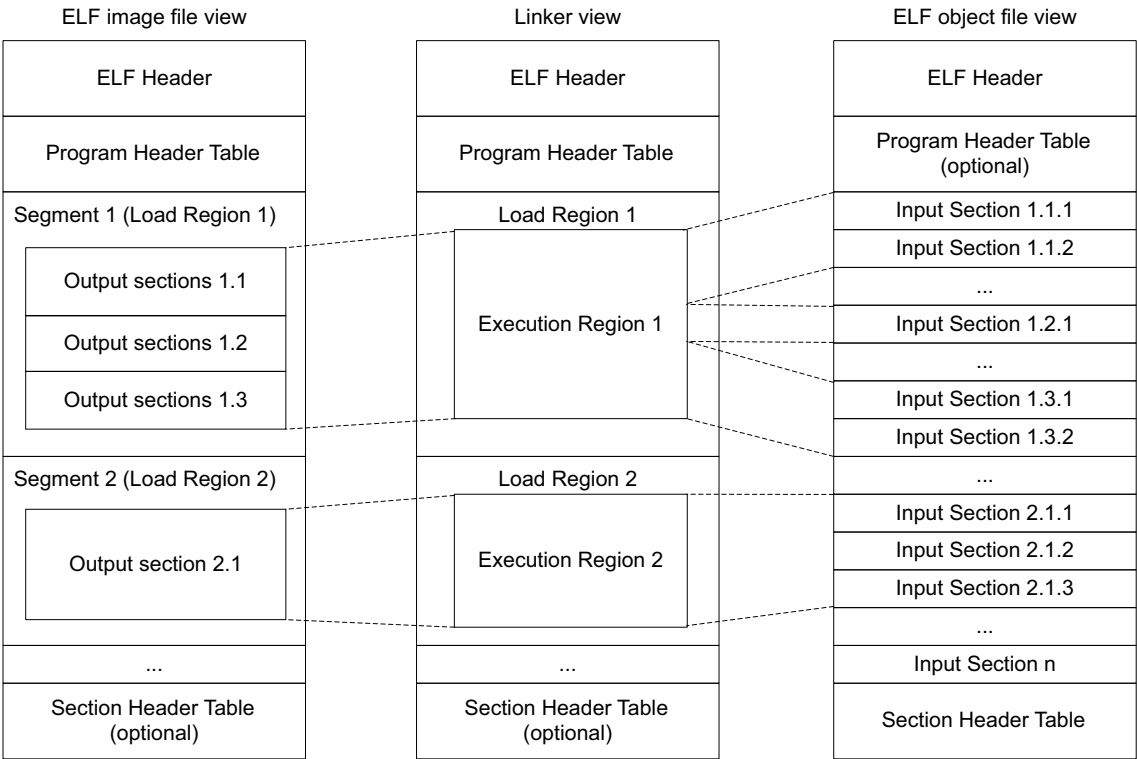


Figure 4-1 Relationship between sections, regions, and segments

4.1.1 See also

Concepts

- [Input sections, output sections, regions, and Program Segments on page 4-5.](#)
- [Load view and execution view of an image on page 4-6.](#)

4.2 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 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, 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 is a contiguous sequence of one, two, or three output sections depending on the contents of the number of sections with different attributes. The output sections in a region are sorted according to their attributes. The RO output section is first, then 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.

Program Segment

A Program Segment corresponds to a load region and contains output sections. Program Segments hold information such as text and data.

———— Note ————

With `armlink`, the maximum size of a Program Segment is 2GB.

4.2.1 See also

Concepts

- [The image structure on page 4-3.](#)
- [Methods of specifying an image memory map with the linker on page 4-7.](#)
- [Section placement with the linker on page 4-18.](#)

4.3 Load view and execution view of an image

Image regions are placed in the system memory map at load time. 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:

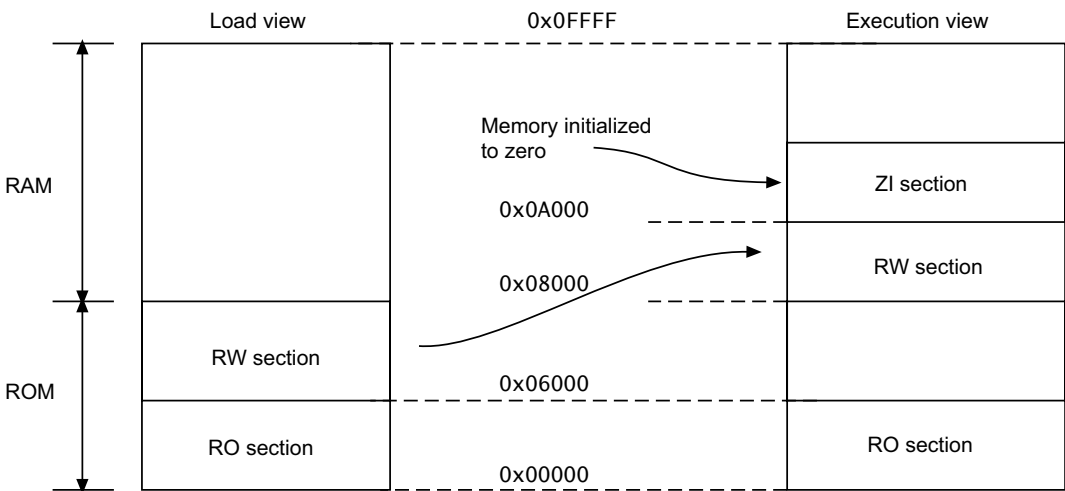


Figure 4-2 Load and execution memory maps

The following table compares the load and execution views:

Table 4-1 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 region in the load address space.	Execution region	A region in the execution address space.

4.3.1 See also

Concepts

- [The image structure on page 4-3.](#)
- [Input sections, output sections, regions, and Program Segments on page 4-5.](#)
- [Methods of specifying an image memory map with the linker on page 4-7.](#)
- [Section placement with the linker on page 4-18.](#)

4.4 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. To construct 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 maps.

Depending on the complexity of the memory maps of the image, there are two ways to pass this information to `armlink`:

Using command-line options

The following options can be used for simple cases where an image has only one or two load regions and up to three execution regions:

- `--first`.
- `--last`.
- `--ro_base`.
- `--rw_base`.
- `--split`.
- `--rosplit`.
- `--zi_base`.

These options provide a simplified notation that gives the same settings as a scatter-loading description for a simple image.

Using a scatter file

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 —————

The PI scatter-loading keyword is not supported in AArch64 state.

————— Note —————

You cannot use `--scatter` with the other memory map related command-line options.

4.4.1 Comparison of scatter file with the equivalent command-line options

The following table shows a scatter file and the equivalent command-line options you can use to get the same result.

Table 4-2 Comparison of scatter file and equivalent command-line options

Scatter file	Equivalent command-line options
LR1 0x0000 0x20000	
{	
ER1 0x0 0x2000	--ro_base=0x0
{	
init.o (INIT, +FIRST)	--first=init.o(init)
*(+R0)	
}	
RAM 0x400000	--rw_base=0x400000
{	
*(+RW)	
}	
RAM 0x405000	--zi_base=0x405000
{	
*(+ZI)	
}	
}	

4.4.2 See also

Tasks

- [Chapter 8 Using scatter files.](#)

Concepts

- [The image structure on page 4-3.](#)
- [Input sections, output sections, regions, and Program Segments on page 4-5.](#)
- [Load view and execution view of an image on page 4-6.](#)
- [Types of simple image on page 4-9.](#)

Reference

armlink Reference Guide:

- [--first on page 2-53.](#)
- [--last on page 2-75.](#)
- [--ro_base on page 2-106.](#)
- [--rosplit on page 2-107.](#)
- [--rw_base on page 2-108.](#)
- [--scatter on page 2-110.](#)
- [--split on page 2-118.](#)
- [--zi_base on page 2-152.](#)

4.5 Types of simple image

A simple image consists of a number of input sections of type RO, RW, and ZI. These input sections are collated to form the RO, RW, and ZI output sections. Depending on how the output sections are arranged within load and execution regions, there are three basic types of simple image:

- Type 1** One region in load view, three contiguous regions in execution view. Use the `--ro_base` option to create this type of image.
- Type 2** One region in load view, three 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, three non-contiguous regions in execution view. Use the `--ro_base`, `--rw_base`, and `--split` options to create this type of image.

In all the simple image types:

- The first execution region contains the RO output section.
- The second execution region contains the RW output section (if present).
- The third execution region contains the ZI output section (if present).

These execution regions are referred to as the RO, the RW, and the ZI 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.

4.5.1 See also

Concepts

- [Type 1 image, one load region and contiguous execution regions](#) on page 4-10.
- [Type 2 image, one load region and non-contiguous execution regions](#) on page 4-12.
- [Type 3 image, two load regions and non-contiguous execution regions](#) on page 4-14.
- [Equivalent scatter-loading descriptions for simple images](#) on page 8-62.

Reference

armlink Reference Guide:

- [--ro_base](#) on page 2-106.
- [--rosplit](#) on page 2-107.
- [--rw_base](#) on page 2-108.
- [--scatter](#) on page 2-110.
- [--split](#) on page 2-118.
- [--zi_base](#) on page 2-152.

4.6 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 three execution regions placed contiguously in the memory map. This approach is suitable for systems that load programs into RAM, for example, an OS bootloader or a desktop system.

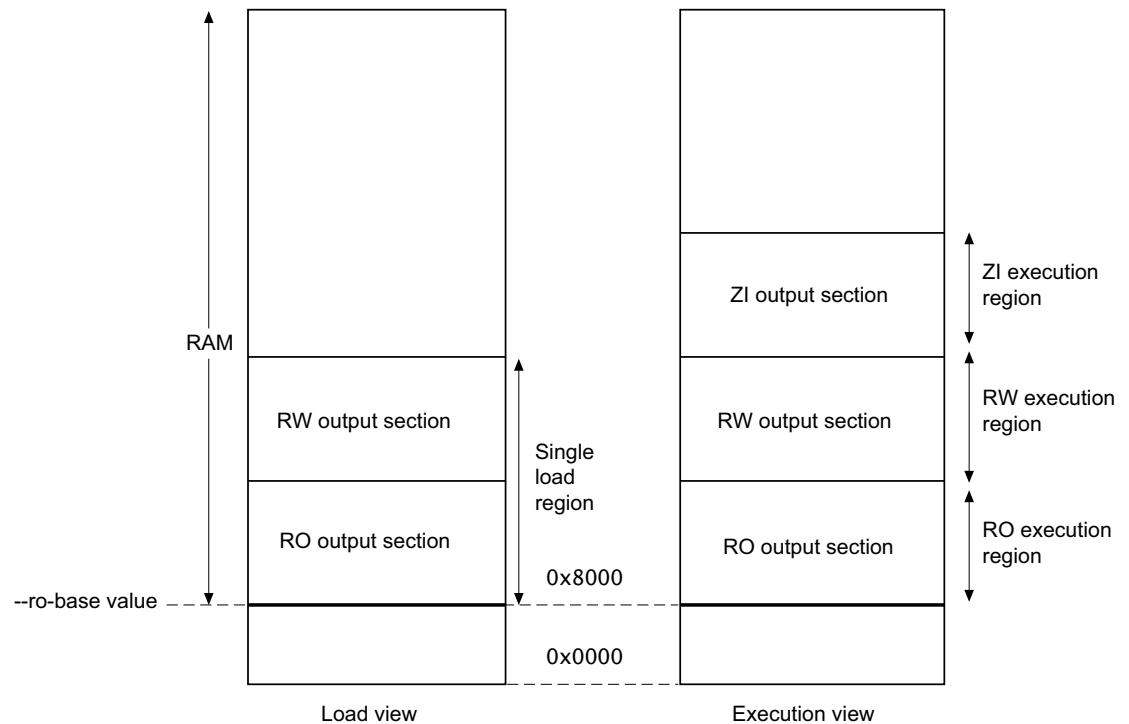


Figure 4-3 Simple type 1 image

Use the following command for images of this type:

```
armlink --ro_base 0x8000
```

———— **Note** ————

0x8000 is the default address, so you do not have to specify --ro_base for the example.

4.6.1 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.

4.6.2 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 execution 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.

4.6.3 See also

Concepts

- [The image structure](#) on page 4-3.
- [Input sections, output sections, regions, and Program Segments](#) on page 4-5.
- [Load view and execution view of an image](#) on page 4-6.

Reference

armlink Reference Guide:

- [--ro_base](#) on page 2-106.
- [--zi_base](#) on page 2-152.

4.7 Type 2 image, 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:

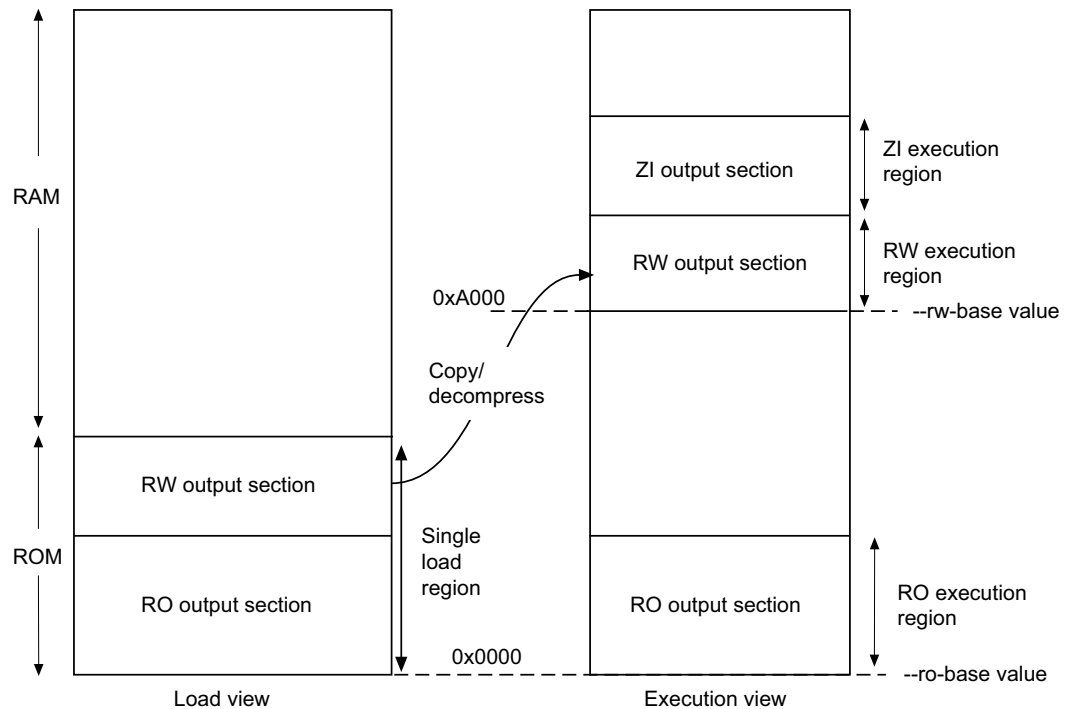


Figure 4-4 Simple type 2 image

Use the following command for images of this type:

```
armlink --ro_base 0x0 --rw_base 0xA000
```

4.7.1 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.

4.7.2 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 exec_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.

4.7.3 See also

Concepts

- [The image structure on page 4-3.](#)
- [Input sections, output sections, regions, and Program Segments on page 4-5.](#)
- [Load view and execution view of an image on page 4-6.](#)
- [Type 1 image, one load region and contiguous execution regions on page 4-10.](#)

Reference

armlink Reference Guide:

- [--ro_base on page 2-106.](#)
- [--rw_base on page 2-108.](#)
- [--zi_base on page 2-152.](#)

4.8 Type 3 image, two 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 two root load regions.

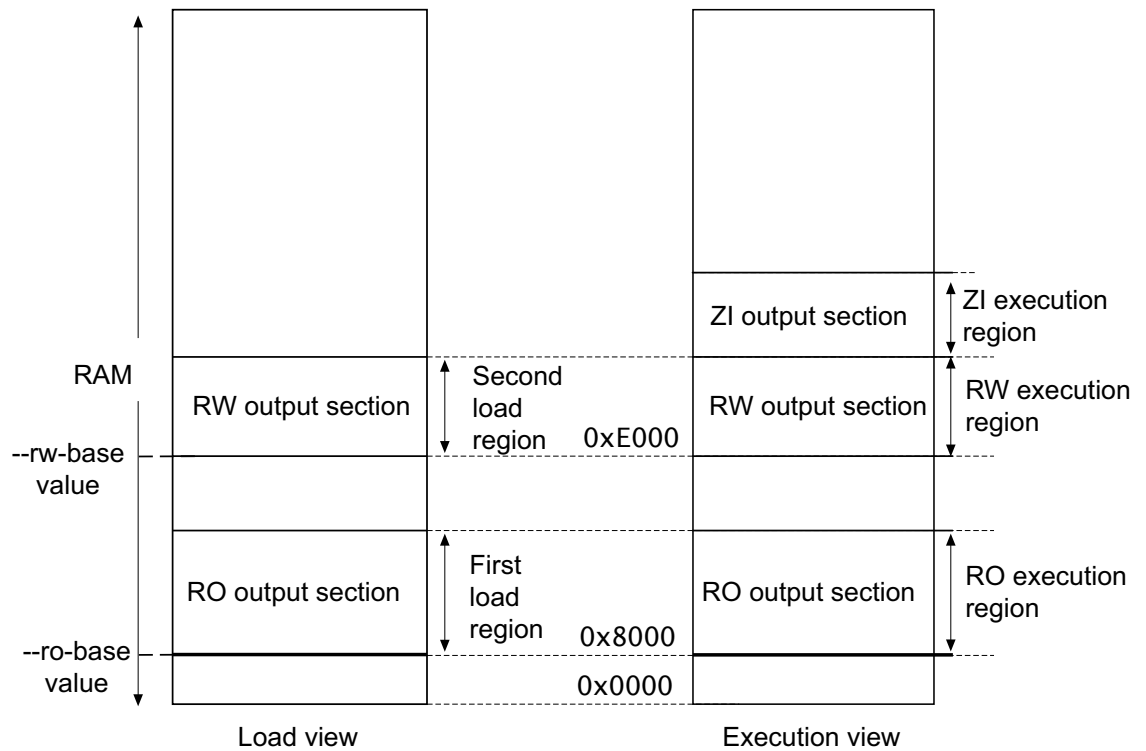


Figure 4-5 Simple type 3 image

Use the following command for images of this type:

```
armlink --split --ro_base 0x8000 --rw_base 0xE000
```

4.8.1 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.

4.8.2 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 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. Both RO and RW are root regions.

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`.

4.8.3 See also

Concepts

- [The image structure on page 4-3.](#)
- [Input sections, output sections, regions, and Program Segments on page 4-5.](#)
- [Load view and execution view of an image on page 4-6.](#)
- [Type 2 image, one load region and non-contiguous execution regions on page 4-12.](#)

Reference

armlink Reference Guide:

- [--ro_base on page 2-106.](#)
- [--rw_base on page 2-108.](#)
- [--split on page 2-118.](#)

4.9 Image entry points

An entry point in an image is a location where program execution can start. 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.

4.9.1 See also

Tasks

- [About specifying an initial entry point on page 4-17.](#)

Reference

armlink Reference Guide:

- [--entry on page 2-46.](#)

armasm Reference Guide:

- [ENTRY on page 10-35.](#)

4.10 About specifying an initial entry point

You must specify at least one initial entry point for a program otherwise the linker produces a warning. Not every source file has to have an entry point. Multiple entry points in a single source file are not permitted.

For 32-bit embedded applications with ROM at zero use `--entry 0x0`, or optionally `0xFFFF0000` for CPUs that are using high vectors.

Note

High vectors are not supported in AArch64 state.

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 (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 has been specified by using the `ENTRY` directive.
 - No entry point has been specified by using the `ENTRY` directive.

4.10.1 See also

Concepts

- [What is a root region?](#) on page 8-12.

Reference

armlink Reference Guide:

- [--entry](#) on page 2-46.

armasm Reference Guide:

- [ENTRY](#) on page 10-35.

4.11 Section placement with the linker

By default, the linker places input sections in the following order when generating an image:

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 their relative positions in the input file if they have the same attributes and section names, except where overridden by `FIRST` or `LAST`.

Portions of the image are collected together into a minimum number of contiguous regions.

Note

The sorting order is unaffected by ordering within scatter files or object file names.

These rules mean that the positions of input sections with identical attributes and names included from libraries is not predictable. If you require more precise positioning, specify the individual modules explicitly in a scatter file, and include the modules in the input file list for the `armlink` command.

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.

By default, the linker creates an image consisting of an RO output section, an RW output section, and optionally a ZI output section. The RO output section can be protected at run-time on systems that have memory management hardware. RO sections can also be placed into ROM in the target.

Alternative sorting orders are available with the `--sort=algorithm` command-line option. The linker might change the *algorithm* to minimise the amount of veneers generated if no algorithm is chosen.

4.11.1 Handling unassigned sections

The linker might not be able to place some input sections in any execution region. When this happens, the linker 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.

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 is used when placing any unassigned sections that have identical properties.

4.11.2 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.

4.11.3 See also

Tasks

- [Placing sections with FIRST and LAST attributes on page 4-20.](#)
- [Methods of placing functions and data at specific addresses on page 8-17.](#)
- [Explicit placement of a named section with scatter-loading on page 8-24.](#)
- [Placement of unassigned sections with the .ANY module selector on page 8-25.](#)

Concepts

- [The image structure on page 4-3.](#)
- [Input sections, output sections, regions, and Program Segments on page 4-5.](#)
- [Load view and execution view of an image on page 4-6.](#)
- [About ordering execution regions containing T32 code on page 4-24.](#)
- [Overview of veneers on page 4-25.](#)
- [Section alignment with the linker on page 4-21.](#)

Reference

armlink Reference Guide:

- [--sort on page 2-116.](#)
- [Syntax of an input section description on page 4-24.](#)

4.12 Placing sections with 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 file to mark the first and last input sections in an execution region if the placement order is important.
- However, `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.

4.12.1 See also

Concepts

- [The image structure on page 4-3.](#)
- [Input sections, output sections, regions, and Program Segments on page 4-5.](#)
- [Load view and execution view of an image on page 4-6.](#)
- [Section placement with the linker on page 4-18.](#)
- [About scatter-loading on page 8-3.](#)

Reference

armlink Reference Guide:

- [--first on page 2-53.](#)
- [--last on page 2-75.](#)
- [Syntax of an input section description on page 4-24.](#)

4.13 Section alignment with the linker

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.

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.

If you require strict conformance with the ELF specification then use the `--no_legacyalign` option. The linker faults the base address of a region if it is not aligned so padding might be inserted to ensure compliance. When `--no_legacyalign` is used the region alignment is the maximum alignment of any input section contained by the region.

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.

4.13.1 See also

Tasks

- [About creating regions on page boundaries on page 8-57.](#)

Reference

armlink Reference Guide:

- [--legacyalign, --no_legacyalign on page 2-76.](#)
- [Load region attributes on page 4-8.](#)
- [Execution region attributes on page 4-13.](#)
- [Example of aligning a base address in execution space but still tightly packed in load space on page 4-43.](#)

4.14 Demand paging

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 on the 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 `--paged` is on:

- The linker automatically generates the Program Headers from the execution region base addresses. The usual situation where one load region generates one Program Header no longer holds.
- 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.

4.14.1 Example

This is an example of a demand paged scatter file:

```
LR1 GetPageSize() + SizeOfHeaders()
{
    ER_RO +0
    {
        *(+R0)
    }
    ER_RW +GetPageSize()
    {
        *(+RW)
    }
    ER_ZI +0
    {
        *(+ZI)
    }
}
```

4.14.2 See also

Concepts

- [About scatter-loading on page 8-3](#)

Reference

armlink Reference Guide:

- [--paged on page 2-95.](#)
- [--pagesize on page 2-96.](#)

- *--pagesize* on page 2-96.
- *--scatter* on page 2-110.
- *GetPageSize()* function on page 4-45.
- *SizeOfHeaders()* function on page 4-46.

4.15 About ordering execution regions containing T32 code

The branch range for 16-bit encoded T32 instructions is 4MB. When an execution region contains T32 code that exceeds 4MB, `armlink` attempts to order sections that are at a similar average call depth and to place the most commonly called sections centrally. This helps to minimize the number of veneers generated.

The branch range for 32-bit T32 instructions is 16MB. Section re-ordering is only required if that limit is exceeded.

To disable section re-ordering, use the `--no_largeregions` command-line option.

4.15.1 See also

Concepts

- [Section placement with the linker on page 4-18.](#)
- [Overview of veneers on page 4-25.](#)

Reference

armlink Reference Guide:

- [--largeregions, --no_largeregions on page 2-73.](#)

4.16 Overview of veneers

Veneers are small sections of code generated by the linker and inserted into your program. The BL instruction is PC-relative and has a limited branch range. Therefore, `arm1ink` must generate veneers when a branch involves a destination beyond the branching range of the BL 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. For example:

```
LDR pc,[pc,#-4]
```

This enables the veneer to branch anywhere in the 4 GB 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. For example:

```
LDR    x16,{pc}+8 ; 0x8 ; [0x8] =
BR     x16
```

This enables the veneer to branch anywhere in the 16 EB 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.

`arm1ink` 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.

4.16.1 See also

Concepts

- [Veneer sharing on page 4-26.](#)
- [Veneer types on page 4-27.](#)
- [Generation of position independent to absolute veneers on page 4-28.](#)
- [Reuse of veneers when scatter-loading on page 4-29.](#)

4.17 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.

4.17.1 See also

Concepts

- [Overview of veneers](#) on page 4-25.
- [About scatter-loading](#) on page 8-3.

Reference

armlink Reference Guide:

- [--veneershare, --no_veneershare](#) on page 2-143.
- [Chapter 4 Formal syntax of the scatter file.](#)

4.18 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-T32 interworking veneer has a range of 256 bytes so the function entry point must appear within 256 bytes of the veneer.
 - A T32-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.
 - A short branch veneer is always position-independent.
- Long branch veneer:
 - Can branch anywhere in the 4GB 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.

Veneers are generated to optimize code size. `arm1ink`, therefore, chooses the variant in order of preference:

1. Inline veneer.
2. Short branch veneer.
3. Long veneer.

4.18.1 See also

Concepts

- [Overview of veneers on page 4-25.](#)

Reference

arm1ink Reference Guide:

- [--inlineveneer, --no_inlineveneer on page 2-67.](#)
- [--max_veneer_passes on page 2-88.](#)

4.19 Generation of position independent to absolute veneers

The normal call instruction encodes the address of the target as an offset from the calling address. When calling from *position independent* (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 in AArch64 state.

4.19.1 See also

Concepts

- [Overview of veneers on page 4-25.](#)

Reference

armlink Reference Guide:

- [--max_veneer_passes on page 2-88.](#)
- [--piveneer, --no_piveneer on page 2-98.](#)

4.20 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 keyword for a load region it prevents:
 - Overlapping of load regions.
 - Veneer sharing.
 - String sharing with the --merge option.
- If you use the OVERLAY keyword 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 always placed in the execution region that contains the call requiring the veneer.

4.20.1 See also

Concepts

- [Overview of veneers on page 4-25.](#)

Reference

- [Load region attributes on page 4-8.](#)
- [Address attributes for load and execution regions on page 4-16.](#)

4.21 About weak references and definitions

Weak references and definitions provide additional flexibility in the way the linker includes various functions and variables in a build. These references are typically to library functions.

Weak references

If the linker cannot resolve normal, non-weak, references to symbols included in the link, 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, the section it is resolved to 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.

Function or variable declarations in C source files can be marked with the weak attribute. As with **extern**, this attribute tells the compiler that a function or variable is declared in another source file. Because the definition of this function or variable might not be available to the compiler, it creates a weak reference to be resolved by the linker.

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 is strongly referenced 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

A function definition, or an exported label in assembler, can also be marked as weak, as can a variable definition. In this case, a weak symbol definition is created in the 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 due to multiply-defined symbols.

4.21.1 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:

```
void init_foo(void) __attribute__((weak));
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:

IMPORT init_foo WEAK
AREA init, CODE, readonly
    BL init_foo
    ;Rest of code
END
```

4.21.2 Example of a weak definition

A simple or dummy implementation of a function can be provided 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.

4.21.3 See also

Concepts

- [How the linker performs library searching, selection, and scanning on page 4-33.](#)
- [How the linker resolves references on page 4-37.](#)

Reference

armlink Reference Guide:

- [--keep on page 2-70.](#)
- [--remove, --no_remove on page 2-105.](#)

armasm Reference Guide:

- [NOP on page 3-108.](#)
- [B, BL, BX, BLX, and BXJ on page 3-44.](#)
- [ENTRY on page 10-35.](#)
- [EXPORT or GLOBAL on page 10-37.](#)

4.22 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.

4.22.1 See also

Tasks

- [About weak references and definitions on page 4-30.](#)
- [Controlling how the linker searches for the ARM standard libraries on page 4-34.](#)

Reference

armlink Reference Guide:

- [--keep on page 2-70.](#)
- [--remove, --no_remove on page 2-105.](#)
- [--scanlib, --no_scanlib on page 2-109.](#)

4.23 Controlling how the linker searches for the ARM standard libraries

By default, the linker searches for the ARM standard libraries in `../lib`, relative to the location of the `armlink` executable. You can also control how the linker searches for the ARM standard libraries with the `--libpath` command-line option.

Some libraries are stored in subdirectories. A library might be added in assembly code or by a linker script from a particular subdirectory. For such a library, the names of subdirectories are placed in each object by using a symbol of the form `Lib$$Request$$sub_dir_name`. You must specify the linker command-line option `--force_scanlib` to inform the compiler to look in all subdirectories for system libraries.

4.23.1 Using 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` and `cpplib`.

The linker searches subdirectories given by the symbol `Lib$$Request$$sub_dir_name`, if you include the path separator character on the end of the library path:

- `\` on Windows.
- `/` on Red Hat Linux.

For example, for `--libpath=mylibs\` and the symbol `Lib$$Request$$armlib` the linker searches the directories:

```
mylibs
mylibs\armlib
```

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.

4.23.2 Library search order

The linker searches for libraries in the following order:

1. Relative to the current path.
2. At the location specified with the command-line option `--libpath`.
3. At the location specified in `../lib`.

4.23.3 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 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.

4.23.4 See also

Concepts

- [How the linker performs library searching, selection, and scanning on page 4-33.](#)

Reference

armlink Reference Guide:

- [--libpath on page 2-77.](#)

ARM C and C++ Libraries and Floating-Point Support User Guide:

- [C and C++ library naming conventions on page 2-115.](#)

ARM C and C++ Libraries and Floating-Point Support Reference Guide:

- [Chapter 2 The C and C++ libraries.](#)

4.24 Specifying user libraries when linking

To specify user libraries:

- 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, `libname.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.

4.24.1 See also

Tasks

- [Controlling how the linker searches for the ARM standard libraries on page 4-34.](#)

Reference

armlink Reference Guide:

- [--libpath on page 2-77.](#)
- [--library on page 2-78.](#)
- [--userlibpath on page 2-139.](#)

ARM C and C++ Libraries and Floating-Point Support Reference Guide:

- [Chapter 2 The C and C++ libraries.](#)

4.25 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. There are two separate lists of files that are maintained. The lists are scanned in the following order to resolve all dependencies:

1. List of system libraries found in `../lib`, or the directories specified by `--libpath`. These might also be specified by the `-Jdir[,dir,...]` compiler option.
2. The list of all other files that have been loaded. These might be specified by the `-Idir[,dir,...]` compiler option.

Each list is scanned using the following process:

1. Search all specified directories to select the most compatible library variants.
2. Add the variants to the list of libraries.
3. 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 stage [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 stage [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.
4. If any non-weak reference remains unsatisfied at the end of the scanning operation, generate an error message.

4.25.1 See also

Concepts

- [About weak references and definitions on page 4-30.](#)
- [How the linker performs library searching, selection, and scanning on page 4-33.](#)
- [Controlling how the linker searches for the ARM standard libraries on page 4-34.](#)
- [Specifying user libraries when linking on page 4-36.](#)

Reference

armlink Reference Guide:

- [--libpath on page 2-77.](#)

Errors and Warnings Reference Guide:

- [List of the armlink error and warning messages on page 3-3.](#)

4.26 Use of the strict family of options in the linker

The strict family of 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 family of options are:

- `--strict`.
- `--[no_]strict_enum_size`.
- `--[no_]strict_flags`.
- `--[no_]strict_ph`.
- `--[no_]strict_relocations`.
- `--[no_]strict_symbols`.
- `--[no_]strict_visibility`.
- `--[no_]strict_wchar_size`.

4.26.1 See also

Reference

armlink Reference Guide:

- [--strict on page 2-120](#).
- [--strict_enum_size, --no_strict_enum_size on page 2-121](#).
- [--strict_flags, --no_strict_flags on page 2-122](#).
- [--strict_ph, --no_strict_ph on page 2-123](#).
- [--strict_relocations, --no_strict_relocations on page 2-124](#).
- [--strict_symbols, --no_strict_symbols on page 2-125](#).
- [--strict_visibility, --no_strict_visibility on page 2-126](#).
- [--strict_wchar_size, --no_strict_wchar_size on page 2-127](#).

Chapter 5

Using linker optimizations

The following topics describe the optimizations available in the linker, `arm11ink`:

Tasks

- *Overriding the compression algorithm used by the linker on page 5-8.*
- *Working with RW data compression on page 5-10.*
- *Inlining functions with the linker on page 5-11.*
- *Handling branches that optimize to a NOP on page 5-14.*

Concepts

- *Elimination of common debug sections on page 5-2.*
- *Elimination of common groups or sections on page 5-3.*
- *Elimination of unused sections on page 5-4.*
- *Elimination of unused virtual functions on page 5-5.*
- *Optimization with RW data compression on page 5-6.*
- *How the linker chooses a compressor on page 5-7.*
- *How compression is applied on page 5-9.*
- *Factors that influence function inlining on page 5-12.*
- *About reordering of tail calling sections on page 5-15.*
- *Restrictions on reordering of tail calling sections on page 5-16.*
- *About merging comment sections on page 5-17.*

5.1 Elimination of common debug sections

In DWARF 3, common debug sections are placed in common groups. `arm1ink` discards all but one copy of each group with the same signature.

5.1.1 See also

Concepts

- [Input sections, output sections, regions, and Program Segments](#) on page 4-5.
- [Elimination of common groups or sections](#) on page 5-3.
- [Elimination of unused sections](#) on page 5-4.
- [Elimination of unused virtual functions](#) on page 5-5.

Reference

armasm Reference Guide:

- [--debug](#) on page 2-17.
- [--dwarf2](#) on page 2-26.
- [--dwarf3](#) on page 2-27.

Other information

- *DWARF for the ARM Architecture*
<http://infocenter.arm.com/help/topic/com.arm.doc.ih0040-/index.html>.
- *DWARF for the ARM 64-bit Architecture (AArch64)*
<http://infocenter.arm.com/help/topic/com.arm.doc.ih0057-/index.html>.
- The DWARF Debugging Standard web site <http://www.dwarfstd.org/>.

5.2 Elimination of common groups or sections

The 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 code sections or groups.

It is possible that the separate instances of common code sections, or 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 code section, or group, based on the attributes of the input objects. `armlink` discards the rest.

If the copies are identical, `armlink` retains the first section or group located.

You control this optimization with the following linker options:

- Use the `--bestdebug` option to use the largest common data (COMDAT) group. This is likely to give the best debug view.
- Use the `--no_bestdebug` option to use the smallest COMDAT group. This is likely to give the smallest code size and is the default option.

Because `--no_bestdebug` is the default, the final image is the same regardless of whether or not you generate debug tables during compilation with `--debug`.

5.2.1 See also

Concepts

- [Input sections, output sections, regions, and Program Segments on page 4-5.](#)
- [Elimination of common debug sections on page 5-2.](#)
- [Elimination of unused sections on page 5-4.](#)
- [Elimination of unused virtual functions on page 5-5.](#)

Reference

armlink Reference Guide:

- [--bestdebug, --no_bestdebug on page 2-13.](#)

5.3 Elimination of unused sections

Unused section elimination is the most significant optimization on image size that is performed by the linker. It removes unreachable code and data from the final image.

Unused section elimination 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 there is no entry point specified for an image, use the `--entry` linker option to specify an entry point and permit unused section elimination to work, if it is enabled.

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 under the following conditions:

- If it contains an entry point.
- If it is referred to, directly or indirectly, by a non-weak reference from an input section containing an entry point.
- If it is specified as the first or last input section by the `--first` or `--last` option (or a scatter-loading equivalent).
- If it is marked as unremovable by the `--keep` option.

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 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.

5.3.1 See also

Concepts

- [Input sections, output sections, regions, and Program Segments](#) on page 4-5.
- [About weak references and definitions](#) on page 4-30.
- [Elimination of common debug sections](#) on page 5-2.
- [Elimination of common groups or sections](#) on page 5-3.
- [Elimination of unused virtual functions](#) on page 5-5.

Reference

armlink Reference Guide:

- [--entry](#) on page 2-46.
- [--first](#) on page 2-53.
- [--info](#) on page 2-61.
- [--keep](#) on page 2-70.
- [--last](#) on page 2-75.
- [--remove, --no_remove](#) on page 2-105.

5.4 Elimination of unused virtual functions

Unused section elimination efficiently removes unused functions from C code. In C++ applications, virtual functions and *RunTime Type Information* (RTTI) objects are referenced by pointer tables, known as vtables. Without extra information, the linker cannot determine which vtable entries are accessed at runtime. This means that the standard unused section elimination algorithm used by the linker cannot guarantee to remove unused virtual functions and RTTI objects.

Virtual Function Elimination (VFE) is a refinement of unused section elimination to reduce ROM size in images generated from C++ code. You can use this optimization to eliminate unused virtual functions and RTTI objects from your code.

An input section that contains more than one function can only be eliminated if all the functions are unused. The linker cannot remove unused functions from within a section.

VFE is a collaboration between the ARM compiler and the linker whereby the compiler supplies extra information about unused virtual functions that is then used by the linker. Based on this analysis, the linker is able to remove unused virtual functions and RTTI objects.

Note

For VFE to work, the assembler requires all objects using C++ to have VFE annotations. If the linker finds a C++ mangled symbol name in the symbol table of an object and VFE information is not present, it turns off the optimization.

The compiler places the extra information in sections with names beginning `.arm_vfe`. These sections are ignored by the linker when it is not VFE-aware.

5.4.1 See also

Concepts

- [Elimination of common debug sections on page 5-2.](#)
- [Elimination of common groups or sections on page 5-3.](#)
- [Elimination of unused sections on page 5-4.](#)

Reference

armlink Reference Guide:

- [--vfemode on page 2-146.](#)

5.5 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 in AArch64 state.

5.5.1 See also

Concepts

- [How compression is applied on page 5-9.](#)

Tasks

- [Overriding the compression algorithm used by the linker on page 5-8.](#)
- [Working with RW data compression on page 5-10.](#)

Concepts

- [How the linker chooses a compressor on page 5-7.](#)
- [How compression is applied on page 5-9.](#)

5.6 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, the linker 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.

5.6.1 See also

Concepts

- [Optimization with RW data compression on page 5-6.](#)
- [How compression is applied on page 5-9.](#)

Tasks

- [Overriding the compression algorithm used by the linker on page 5-8.](#)
- [Working with RW data compression on page 5-10.](#)

5.7 Overriding the compression algorithm used by the linker

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.

5.7.1 See also

Concepts

- [Optimization with RW data compression on page 5-6.](#)
- [How the linker chooses a compressor on page 5-7.](#)
- [How compression is applied on page 5-9.](#)

Tasks

- [Working with RW data compression on page 5-10.](#)

Reference

armlink Reference Guide:

- [--datacompressor on page 2-31.](#)

5.8 How compression is applied

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 and, 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, and the length of the phrase.

5.8.1 See also

Concepts

- [Optimization with RW data compression on page 5-6.](#)
- [How the linker chooses a compressor on page 5-7.](#)

Tasks

- [Overriding the compression algorithm used by the linker on page 5-8.](#)
- [Working with RW data compression on page 5-10.](#)

Reference

armlink Reference Guide:

- [--datacompressor on page 2-31.](#)

5.9 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.
- The linker turns off RW compression if there is a reference from a compressed region to a linker-defined symbol that uses a load address.
- 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.

5.9.1 See also

Concepts

- [Optimization with RW data compression on page 5-6.](#)
- [How the linker chooses a compressor on page 5-7.](#)
- [How compression is applied on page 5-9.](#)
- [Load\\$\\$ execution region symbols on page 7-7.](#)
- [Chapter 8 Using scatter files.](#)

Tasks

- [Overriding the compression algorithm used by the linker on page 5-8.](#)

Reference

armlink Reference Guide:

- [--map, --no_map on page 2-86.](#)
- [Chapter 4 Formal syntax of the scatter file.](#)

5.10 Inlining functions with the linker

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.

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

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.

5.10.1 See also

Reference

- [Factors that influence function inlining on page 5-12.](#)
- [Elimination of unused sections on page 5-4.](#)

armlink Reference Guide:

- [--info on page 2-61.](#)
- [--inline, --no_inline on page 2-66.](#)

5.11 Factors that influence function inlining

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 a single 16-bit T32 instruction or 32-bit 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 5-1 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 ARM 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 32-bit T32 *If-Then* (IT) block cannot inline a 16-bit 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.

5.11.1 See also

Concepts

- [Handling branches that optimize to a NOP on page 5-14.](#)

Using the Assembler:

- [Conditional instructions on page 8-2.](#)

Reference

armasm Reference Guide:

- [ADD, SUB, RSB, ADC, SBC, and RSC](#) on page 3-27.
- [B, BL, BX, BLX, and BXJ](#) on page 3-44.
- [CPS](#) on page 3-54.
- [MOV and MVN](#) on page 3-93.
- [MRS \(PSR to general-purpose register\)](#) on page 3-100.
- [MSR \(general-purpose register to PSR\)](#) on page 3-103.
- [IT](#) on page 3-63.

5.12 Handling branches that optimize to a NOP

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.

Note

This feature is not supported in AArch64.

5.12.1 See also

Concepts

- [About reordering of tail calling sections on page 5-15.](#)

Reference

armlink Reference Guide:

- [--branchnop, --no_branchnop on page 2-17.](#)

5.13 About reordering of 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.

You can take advantage of this behavior by using 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.

5.13.1 See also

Concepts

- [Veneer types](#) on page 4-27.
- [Handling branches that optimize to a NOP](#) on page 5-14.
- [Restrictions on reordering of tail calling sections](#) on page 5-16.

Reference

armlink Reference Guide:

- [--info](#) on page 2-61.
- [--tailreorder, --no_tailreorder](#) on page 2-132.

5.14 Restrictions on 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.

5.14.1 See also

Concepts

- [About reordering of tail calling sections on page 5-15.](#)

5.15 About merging comment sections

If input object files have any `.comment` sections that are identical, then the linker merges them to produce the smallest `.common` 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

If you do not want to retain the information in a `.comment` section, then you can use the `--no_comment_section` option to strip this section from the image.

5.15.1 See also

Reference

armlink Reference Guide:

- [--comment_section, --no_comment_section](#) on page 2-26.
- [--filtercomment, --no_filtercomment](#) on page 2-51

Chapter 6

Getting information about images when linking

The following topics describe how to get image information from `armlink`:

Tasks

- [Identifying the source of some link errors on page 6-3.](#)
- [How to find where a symbol is placed when linking on page 6-6.](#)

Concepts

- [Linker options for getting information about images on page 6-2.](#)
- [Example of using the `--info` linker option on page 6-4.](#)

6.1 Linker options for getting information about images

You can use following options to get information about how your image is generated by the linker:

`--info=topic[,topic,...]`

Displays information about various topics.

`--map`

Displays the image memory map, and contains the address and the size of each load region, execution region, and input section in the image, including linker-generated input sections. It also shows how RW data compression is applied.

`--section_index_display`

Use with `--map` to change the display of the index column.

`--show_cmdline`

Outputs the command-line used by the linker.

`--symbols`

Displays a list of each local and global symbol used in the link step, and its value.

`--verbose`

Displays detailed information about the link operation, including the objects that are included and the libraries that contain them.

`--xref`

Displays a list of all cross-references between input sections.

`--xrefdbg`

Displays a list of all cross-references between input debug sections.

The information can be written to a file using the `--list=filename` option.

6.1.1 See also

Concepts

- [Section alignment with the linker on page 4-21.](#)
- [Optimization with RW data compression on page 5-6.](#)
- [Identifying the source of some link errors on page 6-3.](#)
- [Example of using the `--info` linker option on page 6-4.](#)

Reference

armlink Reference Guide:

- [--info on page 2-61.](#)
- [--list on page 2-81.](#)
- [--map, --no_map on page 2-86.](#)
- [--section_index_display on page 2-111.](#)
- [--show_cmdline on page 2-112.](#)
- [--symbols, --no_symbols on page 2-128.](#)
- [--verbose on page 2-144.](#)
- [--xref, --no_xref on page 2-149.](#)
- [--xrefdbg, --no_xrefdbg on page 2-150.](#)

6.2 Identifying the source of some link errors

You can use `--info` inputs to identify the source of some link errors. For example, you can search the output to locate undefined references from library objects or multiply defined symbols caused by retargeting some library functions and not others. Search backwards from the end of this output to find and resolve link errors.

You can also use the `--verbose` option to output similar text with additional information on the linker operations.

6.2.1 See also

Concepts

- [Linker options for getting information about images on page 6-2.](#)

Reference

armlink Reference Guide:

- [--info on page 2-61.](#)
- [--verbose on page 2-144.](#)

6.3 Example of using the `--info linker option`

To display the component sizes when linking enter:

```
armlink --info sizes ...
```

Here, `sizes` gives a list of the Code and Data sizes for each input object and library member in the image. Using this option implies `--info sizes,totals`.

The following example shows the output in tabular format with the totals separated out for easy reading:

Example 6-1 Image component size information

Code (inc. data)	RO Data	RW Data	ZI Data	Debug	
3712	1580	19	44	10200	7436 Object Totals
0	0	16	0	0	(incl. Generated)
0	0	3	0	0	(incl. Padding)
21376	648	805	4	300	10216 Library Totals
0	0	6	0	0	(incl. Padding)
=====					
Code (inc. data)	RO Data	RW Data	ZI Data	Debug	
25088	2228	824	48	10500	17652 Grand Totals
25088	2228	824	48	10500	17652 ELF Image Totals
25088	2228	824	48	0	0 ROM Totals
=====					
Total RO	Size (Code + RO Data)		25912 (25.30kB)		
Total RW	Size (RW Data + ZI Data)		10548 (10.30kB)		
Total ROM	Size (Code + RO Data + RW Data)		25960 (25.35kB)		

In this example:

Code (inc. Data)

Shows how many bytes are occupied by code. In this image, there are 3712 bytes of code. This includes 1580 bytes of inline data (`inc. data`), for example, literal pools, and short strings.

RO Data Shows how many bytes are occupied by read-only data. This is in addition to the inline data included in the `Code (inc. data)` column.

RW Data Shows how many bytes are occupied by read-write data.

ZI Data Shows how many bytes are occupied by zero-initialized data.

Debug Shows how many bytes are occupied by debug data, for example, debug input sections and the symbol and string table.

Object Totals

Shows how many bytes are occupied by objects linked together to generate the image.

(incl. Generated)

armlink might generate image contents, for example, interworking veneers, and input sections such as region tables. If the Object Totals row includes this type of data, it is shown in this row.

In the example, there are 19 bytes of RO data in total, of which 16 bytes is linker-generated RO data.

Library Totals

Shows how many bytes are occupied by library members that have been extracted and added to the image as individual objects.

(incl. Padding)

armlink inserts padding, if required, to force section alignment. If the Object Totals row includes this type of data, it is shown in the associated (incl. Padding) row. Similarly, if the Library Totals row includes this type of data, it is shown in its associated row.

In the example, there are 19 bytes of RO data in the object total, of which 3 bytes is linker-generated padding, and 805 bytes of RO data in the library total, with 6 bytes of padding.

Grand Totals

Shows the true size of the image. In the example, there are 10200 bytes of ZI data (in Object Totals) and 300 of ZI data (in Library Totals) giving a total of 10500 bytes.

ELF Image Totals

If you are using RW data compression (the default) to optimize ROM size, the size of the final image changes and this is reflected in the output from --info. Compare the number of bytes under Grand Totals and ELF Image Totals to see the effect of compression.

In the example, RW data compression is not enabled. If data is compressed, the RW value changes.

ROM Totals

Shows the minimum size of ROM required to contain the image. This does not include ZI data and debug information which is not stored in the ROM.

6.3.1 See also**Concepts**

- [Linker options for getting information about images on page 6-2.](#)

Reference

armlink Reference Guide:

- [--info on page 2-61.](#)

6.4 How to find where a symbol is placed when linking

To find where a symbol is placed in an ELF image file when linking, use the `--keep=section_id` and `--map` options to view the image memory map. For example, if `object(section)` is the section containing the symbol, enter:

```
armlink --cpu=8-A.32 --keep=object(section) --map s.o --output=s.axf
```

The memory map shows where the section containing the symbol is placed.

6.4.1 Example

Do the following:

1. Create the file `s.c` containing the following source code:

```
long array[10] __attribute__((section ("ARRAY")));

int main()
{
    return sizeof(array);
}
```
2. Compile the source:

```
armclang -target armv8a-arm-none-eabi -c s.c -o s.o
```
3. Link the object `s.o`, keeping the `ARRAY` symbol and displaying the image memory map:

```
armlink --cpu=8-A.32 --keep=s.o(ARRAY) --map s.o --output=s.axf
```
4. Locate the `ARRAY` symbol in the output, for example:

```
...
Execution Region ER_RW (Base: 0x000083a8, Size: 0x00000028, Max: 0xffffffff, ABSOLUTE)
```

Base Addr	Size	Type	Attr	Idx	E Section Name	Object
0x000083a8	0x00000028	Data	RW	4	ARRAY	s.o

This shows that the array is placed in execution region `ER_RW`.

6.4.2 See also

Tasks

- [Using fromelf to find where a symbol is placed in an executable ELF image on page 3-13.](#)

Reference

armlink Reference Guide:

- [--keep on page 2-70.](#)
- [--map, --no_map on page 2-86.](#)
- [--output on page 2-92.](#)

Chapter 7

Accessing and managing symbols with armlink

The following topics describe how to access and manage symbols with the linker, armlink:

Tasks

- [About mapping symbols on page 7-3.](#)
- [Accessing linker-defined symbols on page 7-4.](#)
- [Linker defined symbols and scatter files on page 7-11.](#)
- [Importing linker-defined symbols in C and C++ on page 7-12.](#)
- [Importing linker-defined symbols in ARM assembly language on page 7-13.](#)
- [Accessing symbols in another image on page 7-17.](#)
- [Creating a symdefs file on page 7-18.](#)
- [Outputting a subset of the global symbols on page 7-19.](#)
- [Reading a symdefs file on page 7-20.](#)
- [Specifying steering files on the linker command-line on page 7-24.](#)
- [About hiding and renaming global symbols with a steering file on page 7-27.](#)
- [About using \\$Super\\$\\$ and \\$Sub\\$\\$ to patch symbol definitions on page 7-28.](#)

Concepts

- [Region-related symbols on page 7-5.](#)
- [Region name values when not scatter-loading on page 7-10.](#)
- [Section-related symbols on page 7-14.](#)
- [What is a steering file? on page 7-23.](#)

Reference

- [Image\\$\\$ execution region symbols on page 7-6.](#)
- [Load\\$\\$ execution region symbols on page 7-7.](#)

- *Load\$\$LR\$\$ load region symbols on page 7-9.*
- *Image symbols on page 7-15.*
- *Input section symbols on page 7-16.*
- *Symdefs file format on page 7-21.*
- *Steering file command summary on page 7-25.*
- *Steering file format on page 7-26.*

7.1 About mapping symbols

Mapping symbols are generated by armclang and armasm to identify inline transitions between:

- code and data at literal pool boundaries
- A32 code and T32 code, such as A32/T32 interworking veneers.

The mapping symbols are:

\$a	start of a sequence of A32 instructions
\$t	start of a sequence of T32 instructions
\$d	start of a sequence of data items, such as a literal pool.
\$x	start of A64 code.

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)  R0 Data
      x           y      z

```

In this example, the y is marked with \$d, and R0 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.

7.1.1 See also

Concepts

armasm User Guide:

- [Symbol naming rules on page 10-3.](#)

Reference

armlink Reference Guide:

- [--list_mapping_symbols, --no_list_mapping_symbols on page 2-82.](#)
- [--strict_symbols, --no_strict_symbols on page 2-125.](#)

fromelf User Guide:

- [--strip=option\[,option,...\] on page 4-68.](#)
- [--text on page 4-71.](#)

Other information

- *ELF for the ARM Architecture*
<http://infocenter.arm.com/help/topic/com.arm.doc.ihl0044-/index.html>.
- *ELF for the ARM 64-bit Architecture (AArch64)*
<http://infocenter.arm.com/help/topic/com.arm.doc.ihl0056-/index.html>.

7.2 Accessing linker-defined symbols

The linker defines some symbols that contain the character sequence `$$`. These symbols, and all other external names containing the sequence `$$`, 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.

7.2.1 See also

Concepts

- [Importing linker-defined symbols in C and C++ on page 7-12.](#)
- [Importing linker-defined symbols in ARM assembly language on page 7-13.](#)

7.3 Region-related symbols

The linker generates the following types of region-related symbols for each region in the image:

- `Image$$`.
- `Load$$`.
- `Load$$LR$$`.

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.

7.3.1 See also

Concepts

- [Image\\$\\$ execution region symbols on page 7-6.](#)
- [Load\\$\\$ execution region symbols on page 7-7.](#)
- [Load\\$\\$LR\\$\\$ load region symbols on page 7-9.](#)
- [Region name values when not scatter-loading on page 7-10.](#)

7.4 Image\$\$ execution region symbols

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 7-1 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.
Image\$\$region_name\$\$RW\$\$Limit	Address of the byte beyond the end of the RW 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.

7.4.1 See also

Tasks

- [Importing linker-defined symbols in C and C++ on page 7-12.](#)
- [Importing linker-defined symbols in ARM assembly language on page 7-13.](#)

Concepts

- [Region-related symbols on page 7-5.](#)
- [Region name values when not scatter-loading on page 7-10.](#)

7.5 Load\$\$ execution region symbols

The linker performs an extra address assignment and relocation pass for relocations that refer to load addresses after RW compression. This delayed relocation permits more information about load addresses to be used in linker-defined symbols.

Note

Load\$\$*region_name* symbols apply only to execution regions, and Load\$\$LR\$\$*load_region_name* symbols apply only to load regions.

The following table shows the symbols that the linker generates for every Load\$\$ execution region present in the image. All the symbols refer to execution addresses after the C library is initialized.

Table 7-2 Load\$\$ execution region symbols

Symbol	Description
Load\$\$ <i>region_name</i> \$\$Base	Load address of the region.
Load\$\$ <i>region_name</i> \$\$Length	Region length in bytes.
Load\$\$ <i>region_name</i> \$\$Limit	Address of the byte beyond the end of the execution region.
Load\$\$ <i>region_name</i> \$\$RO\$\$Base	Address of the RO output section in this execution region.
Load\$\$ <i>region_name</i> \$\$RO\$\$Length	Length of the RO output section in bytes.
Load\$\$ <i>region_name</i> \$\$RO\$\$Limit	Address of the byte beyond the end of the RO output section in the execution region.
Load\$\$ <i>region_name</i> \$\$RW\$\$Base	Address of the RW output section in this execution region.
Load\$\$ <i>region_name</i> \$\$RW\$\$Length	Length of the RW output section in bytes.
Load\$\$ <i>region_name</i> \$\$RW\$\$Limit	Address of the byte beyond the end of the RW output section in the execution region.
Load\$\$ <i>region_name</i> \$\$ZI\$\$Base	Load address of the ZI output section in this execution region.
Load\$\$ <i>region_name</i> \$\$ZI\$\$Length	Load length of the ZI output section in bytes. The Load Length of ZI is zero unless <i>region_name</i> has the ZEROPAD scatter-loading keyword set. If ZEROPAD is set then: Load Length = Image\$\$ <i>region_name</i> \$\$ZI\$\$Length
Load\$\$ <i>region_name</i> \$\$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.

- The symbols do not include ZI output section because it does not exist before the C library is initialized.
- All relocations from RW compressed execution regions must be performed before compression, because the linker cannot resolve a delayed relocation on compressed data.
- 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.

7.5.1 See also

Concepts

- [Optimization with RW data compression](#) on page 5-6.
- [Region-related symbols](#) on page 7-5.
- [Image\\$\\$ execution region symbols](#) on page 7-6.
- [Load\\$\\$LR\\$\\$ load region symbols](#) on page 7-9.
- [Region name values when not scatter-loading](#) on page 7-10.

Reference:

armlink Reference Guide:

- [Execution region attributes](#) on page 4-13

7.6 Load\$\$LR\$\$ load region symbols

A Load\$\$LR\$\$ load region can contain many execution regions, so there are no separate \$\$R0 and \$\$RW components.

Note

Load\$\$LR\$\$*load_region_name* symbols apply only to load regions, and Load\$\$*region_name* symbols apply only to execution regions.

The following table shows the symbols that the linker generates for every Load\$\$LR\$\$ load region present in the image.

Table 7-3 Load\$\$LR\$\$ load region symbols

Symbol	Description
Load\$\$LR\$\$ <i>load_region_name</i> \$\$Base	address of the load region
Load\$\$LR\$\$ <i>load_region_name</i> \$\$Length	length of the load region
Load\$\$LR\$\$ <i>load_region_name</i> \$\$Limit	address of the byte beyond the end of the load region

7.6.1 See also

Concepts

- [The image structure on page 4-3.](#)
- [Input sections, output sections, regions, and Program Segments on page 4-5.](#)
- [Load view and execution view of an image on page 4-6.](#)
- [Region-related symbols on page 7-5.](#)

7.7 Region name values when not scatter-loading

If you are not using scatter-loading, the linker uses region name values of:

- ER_R0, 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_R0\$\$Base.

Note

- The ZI output sections of an image are not created statically, but are automatically created dynamically at runtime. Therefore, there is no load address symbol for ZI output sections.
 - It is recommended that you use region-related symbols in preference to section-related symbols.
-

7.7.1 See also

Concepts

- [Region-related symbols on page 7-5.](#)
- [Image\\$\\$ execution region symbols on page 7-6.](#)
- [Load\\$\\$ execution region symbols on page 7-7.](#)
- [Section-related symbols on page 7-14.](#)

7.8 Linker defined symbols and scatter files

If you are using scatter-loading, the names from a scatter file are used in the linker defined symbols. The scatter file:

- Names all the 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.

7.8.1 See also

Tasks

- [Chapter 8 Using scatter files.](#)

Reference

armlink Reference Guide:

- [--scatter on page 2-110.](#)

7.9 Importing linker-defined symbols in C and C++

You can import linker-defined symbols into your C or C++ source code either by value or by reference:

Import by value

```
extern unsigned int symbol_name;
```

Import by reference

```
extern void *symbol_name;
```

If you declare a symbol as an int, then you must use the address-of operator (&) to obtain the correct value as shown in these examples:

Example 7-1 Importing a linker-defined symbol

```
extern unsigned int Image$$ZI$$Limit;
config.heap_base = (unsigned int) &Image$$ZI$$Limit;
```

Example 7-2 Importing symbols that define a ZI output section

```
extern unsigned int Image$$ZI$$Length;
extern char Image$$ZI$$Base[];
memset(Image$$ZI$$Base,0,(unsigned int)&Image$$Length);
```

7.9.1 See also

Concepts

- [Image\\$\\$ execution region symbols on page 7-6.](#)

7.10 Importing linker-defined symbols in ARM assembly language

To import linker-defined symbols into your assembly language source code, use the `IMPORT` directive.

7.10.1 32-bit applications

Create a 32-bit data word to hold the value of the symbol, for example:

```
IMPORT |Image$$ZI$$Limit|
...
zi_limit DCD |Image$$ZI$$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 A32 and T32, and the architecture you are using.

7.10.2 64-bit applications

Create a 64-bit data doubleword to hold the value of the symbol, for example:

```
IMPORT |Image$$ZI$$Limit|
...
zi_limit DCQ |Image$$ZI$$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.

7.10.3 See also

Concepts

- [Image\\$\\$ execution region symbols on page 7-6.](#)

Reference

armasm Reference Guide:

- [Memory access instructions on page 3-9.](#)
- [IMPORT and EXTERN on page 10-58.](#)

7.11 Section-related symbols

Section-related symbols are symbols generated by the linker when it creates an image without scatter-loading.

The linker generates the following types of section-related symbols:

- Image symbols, if you use command-line options to create a simple image. A simple image has three output sections (RO, RW, and ZI) that produce the three 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*.

7.11.1 See also

Concepts

- [Image symbols on page 7-15.](#)
- [Input section symbols on page 7-16.](#)

7.12 Image symbols

Image symbols are generated by the linker when you use a command-line option to create a simple image.

The following table shows the image symbols:

Table 7-4 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.

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.

7.12.1 See also

Tasks

- [Linker-defined symbols that are not defined when scatter-loading on page 8-10.](#)
- [Specifying stack and heap using the scatter file on page 8-11.](#)

Concepts

- [Types of simple image on page 4-9.](#)
- [About weak references and definitions on page 4-30.](#)

Reference

Using ARM C and C++ Libraries and Floating-Point Support:

- [__user_setup_stackheap\(\) on page 2-58.](#)

7.13 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 7-5 Section-related symbols

Symbol	Section type	Description
<i>SectionName\$\$Base</i>	Input	Address of the start of the consolidated section called <i>SectionName</i> .
<i>SectionName\$\$Length</i>	Input	Length of the consolidated section called <i>SectionName</i> (in bytes).
<i>SectionName\$\$Limit</i>	Input	Address of the byte beyond the end of the consolidated section called <i>SectionName</i> .

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.

7.13.1 See also

Tasks

- [Chapter 8 Using scatter files.](#)

Concepts

- [Input sections, output sections, regions, and Program Segments](#) on page 4-5.

7.14 Accessing symbols in another image

Use a symbol definitions (symdefs) file if you want one image to access the global symbol values of another image.

You can use this, 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.

7.14.1 See also

Tasks

- [Creating a symdefs file on page 7-18.](#)
- [Reading a symdefs file on page 7-20.](#)

Reference

- [Symdefs file format on page 7-21.](#)

7.15 Creating a symdefs file

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.

7.15.1 See also

Concepts

- [Accessing symbols in another image on page 7-17.](#)

Reference

- [Symdefs file format on page 7-21.](#)

armlink Reference Guide:

- [--symdefs on page 2-129.](#)

7.16 Outputting a subset of the global symbols

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.

For an application `image1` containing symbols that you want to expose to another application using a symdefs file:

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 after one or more objects used to create `image1` have changed.

You can now use the symdefs file to link additional applications.

7.16.1 See also

Concepts

- [Accessing symbols in another image on page 7-17.](#)

Reference

- [Symdefs file format on page 7-21.](#)

armlink Reference Guide:

- [--symdefs on page 2-129.](#)

7.17 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.

7.17.1 See also

Reference

- [Symdefs file format on page 7-21](#).

7.18 Symdefs file format

The 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, *N.n* [Build *num*]:
- Date and time of the most recent update of the symdefs file, in the format:
Last Updated: *Date*

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.

7.18.1 Example symdefs file

This example shows a typical symdefs file format:

Example 7-3 Symdefs file format for AArch32

```
#<SYMDEFS># ARM Linker, 6.0 [Build num]: 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
0xFFFFFFFF N __SIG_IGN

```

7.18.2 See also

Concepts

- [Accessing symbols in another image on page 7-17.](#)
- [Creating a symdefs file on page 7-18.](#)
- [Outputting a subset of the global symbols on page 7-19.](#)
- [Reading a symdefs file on page 7-20.](#)

7.19 What is 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. 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.
- 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.

7.19.1 See also

Tasks

- [Specifying steering files on the linker command-line on page 7-24.](#)

Reference

- [Steering file command summary on page 7-25.](#)
- [Steering file format on page 7-26.](#)

armlink Reference Guide:

- [--edit on page 2-41.](#)

7.20 Specifying 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.

7.20.1 See also

Concepts

- [What is a steering file? on page 7-23.](#)

Reference

- [Steering file command summary on page 7-25.](#)
- [Steering file format on page 7-26.](#)

armlink Reference Guide:

- [EXPORT on page 3-2.](#)
- [HIDE on page 3-3.](#)
- [IMPORT on page 3-4.](#)
- [RENAME on page 3-5.](#)
- [REQUIRE on page 3-7.](#)
- [RESOLVE on page 3-8.](#)
- [SHOW on page 3-10.](#)

7.21 Steering file command summary

The steering file commands are:

Table 7-6 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.

7.21.1 See also

Tasks

- [Specifying steering files on the linker command-line on page 7-24.](#)

Concepts

- [What is a steering file? on page 7-23.](#)

Reference

- [Steering file format on page 7-26.](#)

armlink Reference Guide:

- [EXPORT on page 3-2.](#)
- [HIDE on page 3-3.](#)
- [IMPORT on page 3-4.](#)
- [RENAME on page 3-5.](#)
- [REQUIRE on page 3-7.](#)
- [RESOLVE on page 3-8.](#)
- [SHOW on page 3-10.](#)

7.22 Steering file format

A steering file is a plain text file of 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 is 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:

Example 7-4 Example 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
```

7.22.1 See also

Tasks

- [Specifying steering files on the linker command-line](#) on page 7-24.

Concepts

- [What is a steering file?](#) on page 7-23.

Reference

- [Steering file command summary](#) on page 7-25.

armlink Reference Guide:

- [EXPORT](#) on page 3-2.
- [HIDE](#) on page 3-3.
- [IMPORT](#) on page 3-4.
- [RENAME](#) on page 3-5.
- [REQUIRE](#) on page 3-7.
- [RESOLVE](#) on page 3-8.
- [SHOW](#) on page 3-10.

7.23 About hiding and renaming global symbols with a steering file

You can use a steering file to hide and rename global symbol names in output files. You use the `HIDE` and `RENAME` commands accordingly.

For example, you can use steering files to protect intellectual property, or avoid namespace clashes.

Example of renaming a symbol:

Example 7-5 RENAME steering command example

```
RENAME func1 AS my_func1
```

Example of hiding symbols:

Example 7-6 HIDE steering command example

```
; Hides all global symbols with the 'internal' prefix
HIDE internal*
```

7.23.1 See also

Tasks

- [Specifying steering files on the linker command-line](#) on page 7-24.

Concepts

- [What is a steering file?](#) on page 7-23.
- [Steering file command summary](#) on page 7-25.

Reference

- [Steering file format](#) on page 7-26.

armlink Reference Guide:

- [--edit](#) on page 2-41.
- [HIDE](#) on page 3-3.
- [RENAME](#) on page 3-5.

7.24 About using `$Super$$` and `$Sub$$` to patch symbol definitions

There are situations where an existing symbol cannot be modified because, 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()`:

`$Super$$foo` Identifies the original unpatched function `foo()`. Use this to call the original function directly.

`$Sub$$foo` Identifies the new function that is called instead of the original function `foo()`. Use this to add processing before or after the original function.

Note

The `$Sub$$` and `$Super$$` mechanism only works at static link time, `$Super$$` references cannot be imported or exported into the dynamic symbol table.

The following example shows how to insert a call to the function `ExtraFunc()` before the call to the legacy function `foo()`.

Example 7-7 Using `$Super$$` and `$Sub$$`

```
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.
       */
}
```

7.24.1 See also

Other information

- *ELF for the ARM Architecture*
<http://infocenter.arm.com/help/topic/com.arm.doc.ih0044-/index.html>.

Chapter 8

Using scatter files

The following topics describe how you use scatter files with `armlink` to create complex images:

Tasks

- [Specifying stack and heap using the scatter file on page 8-11.](#)
- [Creating root execution regions on page 8-13.](#)
- [Using the `FIXED` attribute to create root regions on page 8-16.](#)
- [Placing a variable at a specific address without scatter-loading on page 8-18.](#)
- [Placing a variable in a named section with scatter-loading on page 8-20.](#)
- [Placing a variable at a specific address with scatter-loading on page 8-22.](#)
- [Selecting veneer input sections in scatter-loading descriptions on page 8-37.](#)
- [Reserving an empty region on page 8-55.](#)
- [Using expression evaluation in a scatter file to avoid padding on page 8-61.](#)

Concepts

- [About scatter-loading on page 8-3.](#)
- [When to use scatter-loading on page 8-4.](#)
- [Scatter-loading command-line option on page 8-5.](#)
- [Images with a simple memory map on page 8-6.](#)
- [Images with a complex memory map on page 8-8.](#)
- [Linker-defined symbols that are not defined when scatter-loading on page 8-10.](#)
- [What is a root region? on page 8-12.](#)
- [Methods of placing functions and data at specific addresses on page 8-17.](#)

- *Explicit placement of a named section with scatter-loading on page 8-24.*
- *Placement of unassigned sections with the .ANY module selector on page 8-25.*
- *Examples of using placement algorithms for .ANY sections on page 8-31.*
- *Example of next_fit algorithm showing behavior of full regions, selectors, and priority on page 8-33.*
- *Examples of using sorting algorithms for .ANY sections on page 8-35.*
- *Placement of code and data with __attribute__((section("name"))) on page 8-38.*
- *Placement of sections at a specific address with __attribute__((section(".ARM.__at_address"))) on page 8-40.*
- *Restrictions on placing __at sections on page 8-41.*
- *Automatic placement of __at sections on page 8-42.*
- *Manual placement of __at sections on page 8-44.*
- *Placement of a key in flash memory with __at on page 8-46.*
- *Placement of sections with overlays on page 8-48.*
- *About placing ARM C and C++ library code on page 8-51.*
- *Example of placing code in a root region on page 8-52.*
- *Example of placing ARM C library code on page 8-53.*
- *Example of placing ARM C++ library code on page 8-54.*
- *About creating regions on page boundaries on page 8-57.*
- *Overalignment of execution regions and input sections on page 8-59.*
- *Expression evaluation in scatter files on page 8-60.*
- *Equivalent scatter-loading descriptions for simple images on page 8-62.*
- *Type 1 image, one load region and contiguous execution regions on page 8-63.*
- *Type 2 image, one load region and non-contiguous execution regions on page 8-65.*
- *Type 3 image, two load regions and non-contiguous execution regions on page 8-67.*
- *Scatter file to ELF mapping on page 8-69.*

8.1 About scatter-loading

The scatter-loading mechanism enables you to specify the memory map of an image to the linker using a description in a text file. 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.

8.1.1 See also

Concepts

- [The image structure on page 4-3.](#)
- [Region-related symbols on page 7-5.](#)
- [When to use scatter-loading on page 8-4.](#)
- [Scatter file to ELF mapping on page 8-69.](#)

8.2 When to use scatter-loading

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.

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.

Scatter-loading is usually required for implementing embedded systems because these use ROM, RAM, and memory-mapped peripherals.

8.2.1 See also

Concepts

- [About scatter-loading on page 8-3.](#)

8.3 Scatter-loading command-line option

The `armlink` command-line option for using scatter-loading is:

```
--scatter=scatter_file
```

This instructs the linker to construct the image memory map as described in *scatter_file*.

The Base Platform linking model supports scatter-loading. To enable this model, use the `--base_platform` command-line option.

Be aware that you cannot use `--scatter` with the following memory map related command-line options:

- `--bpabi`.
- `--dll`.
- `--partial`.
- `--ro_base`.
- `--rw_base`.
- `--rosplit`.
- `--split`.
- `--reloc`.
- `--startup`.
- `--zi_base`.

8.3.1 See also

Concepts

- [Base Platform linking model on page 3-6.](#)
- [About scatter-loading on page 8-3.](#)
- [When to use scatter-loading on page 8-4.](#)
- [Equivalent scatter-loading descriptions for simple images on page 8-62.](#)

Reference

armlink Reference Guide:

- [--base_platform on page 2-12.](#)
- [--bpabi on page 2-16.](#)
- [--dll on page 2-38.](#)
- [--partial on page 2-97.](#)
- [--reloc on page 2-103.](#)
- [--ro_base on page 2-106.](#)
- [--rosplit on page 2-107.](#)
- [--rw_base on page 2-108.](#)
- [--scatter on page 2-110.](#)
- [--split on page 2-118.](#)
- [--startup, --no_startup on page 2-119.](#)
- [--zi_base on page 2-152.](#)
- [Chapter 4 Formal syntax of the scatter file.](#)

8.4 Images with a simple memory map

If an image has a simple memory map, you can either:

- Use a scatter file.
- Specify the memory map using basic linker command-line options.

The following figure shows a simple memory map:

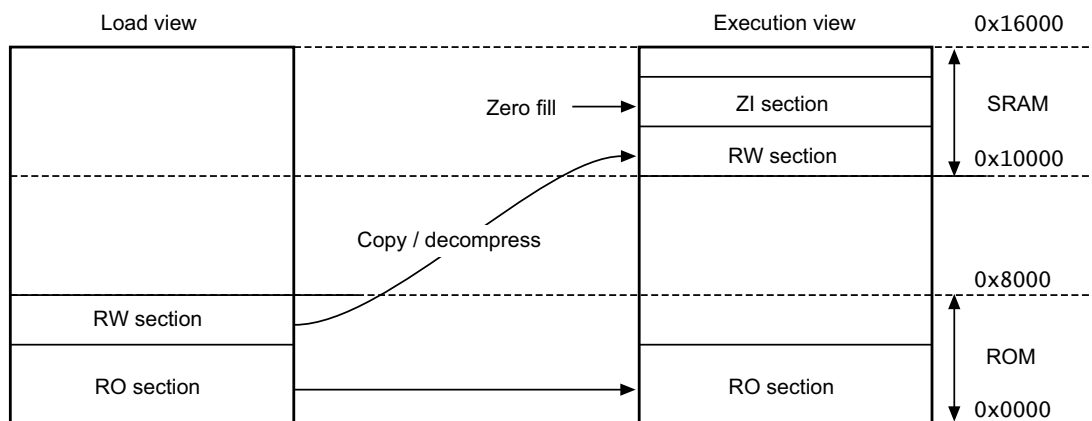


Figure 8-1 Simple scatter-loaded memory map

The following example shows the corresponding scatter-loading description that loads the segments from the object file into memory:

Example 8-1 Simple memory map in a scatter file

```
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 (RAM),
                           ; 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.

In this example, you can achieve the same result, apart from the limit checking with the following linker command-line:

```
armlink --ro_base 0x0 --rw_base 0x10000
```

8.4.1 See also

Concepts

- [About scatter-loading on page 8-3.](#)
- [When to use scatter-loading on page 8-4.](#)
- [Scatter file to ELF mapping on page 8-69.](#)

Reference

armlink Reference Guide:

- [--ro_base on page 2-106.](#)
- [--rw_base on page 2-108.](#)

8.5 Images with a complex memory map

For images with a complex memory map, you cannot specify the memory map using basic linker command-line options. Such images require the use of a scatter file.

The following figure shows a complex memory map:

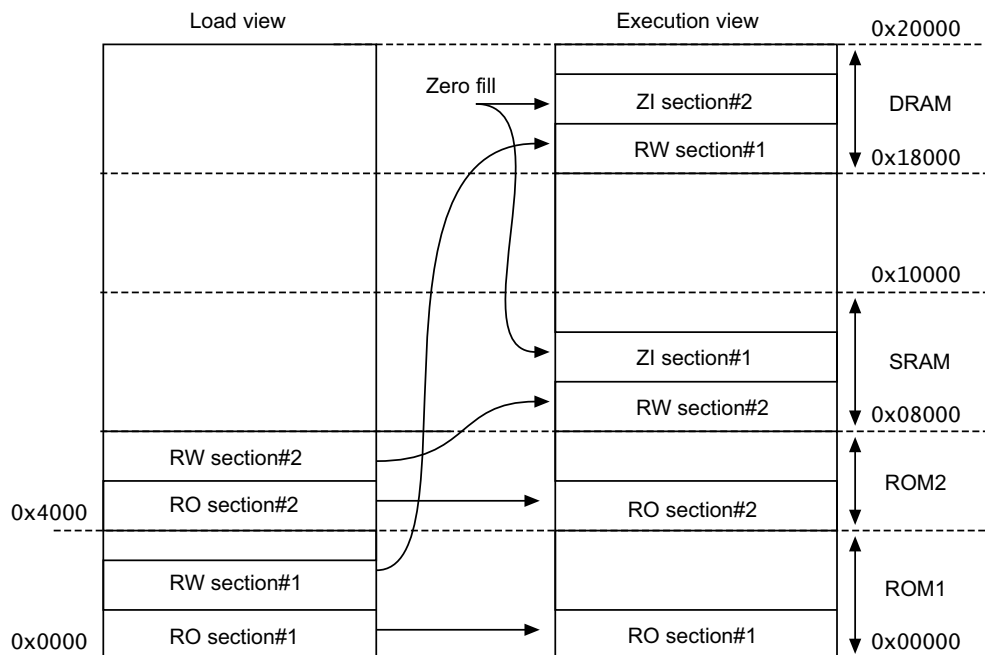


Figure 8-2 Complex memory map

The following example shows the corresponding scatter-loading description that loads the segments from the program1.o and program2.o files into memory:

Example 8-2 Complex memory map in a scatter file

```

LOAD_ROM_1 0x0000          ; Start address for first load region (0x0000)
{
    EXEC_ROM_1 0x0000      ; Start address for first exec region (0x0000)
    {
        program1.o (+R0)   ; Place all code and R0 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      ; Start address for second exec region (0x4000)
    {
        program2.o (+R0)   ; Place all code and R0 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, it is advisable to use the * or .ANY specifier to place leftover code and data.

8.5.1 See also

Tasks

- [Creating root execution regions on page 8-13.](#)
- [Using the FIXED attribute to create root regions on page 8-16.](#)

Concepts

- [About scatter-loading on page 8-3.](#)
- [When to use scatter-loading on page 8-4.](#)
- [Scatter file to ELF mapping on page 8-69.](#)

armlink Reference Guide:

- [Scatter files containing relative base address load regions and a ZI execution region on page 4-38.](#)

8.6 Linker-defined symbols that are not defined when scatter-loading

Be aware that the following symbols are undefined when a scatter file is used:

- Image\$\$RW\$\$Base.
- Image\$\$RW\$\$Limit.
- Image\$\$RO\$\$Base.
- Image\$\$RO\$\$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.

8.6.1 See also

Tasks

- [Accessing linker-defined symbols on page 7-4.](#)
- [Specifying stack and heap using the scatter file on page 8-11.](#)

8.7 Specifying stack and heap using the 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 given in a scatter file.

To select the two region memory model, define two special execution regions in your scatter file named `ARM_LIB_HEAP` and `ARM_LIB_STACK`. Both regions have the `EMPTY` attribute. This causes the library to select the non-default implementation of `__user_setup_stackheap()` that uses the value of the symbols:

- `Image$$ARM_LIB_STACK$$Base`.
- `Image$$ARM_LIB_STACK$$ZI$$Limit`.
- `Image$$ARM_LIB_HEAP$$Base`.
- `Image$$ARM_LIB_HEAP$$ZI$$Limit`.

Only one `ARM_LIB_STACK` or `ARM_LIB_HEAP` region can be specified, and you must allocate a size, for example:

```
LOAD_FLASH ...
{
    ...
    ARM_LIB_STACK 0x40000 EMPTY -0x20000 ; Stack region growing down
    { }
    ARM_LIB_HEAP 0x28000000 EMPTY 0x80000 ; Heap region growing up
    { }
    ...
}
```

You can use a combined stack and heap region by defining a single execution region named `ARM_LIB_STACKHEAP`, with the `EMPTY` attribute. This causes `__user_setup_stackheap()` to use the value of the symbols `Image$$ARM_LIB_STACKHEAP$$Base` and `Image$$ARM_LIB_STACKHEAP$$ZI$$Limit`.

———— Note ————

If you re-implement `__user_setup_stackheap()`, this overrides all library implementations.

8.7.1 See also

Reference

- [Region-related symbols on page 7-5](#).
- ARM C and C++ Libraries and Floating-Point Support User Guide:*
- [__user_setup_stackheap\(\) on page 2-58](#).

8.8 What is a root region?

A root region is a region with the same load and execution address. 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 8-3 Root region with the same load and execution address

```

LR_1 0x040000      ; load region starts at 0x40000
{
    ER_R0 0x040000  ; load address = execution address
    {
        * (+R0)      ; all R0 sections (must include section with
                    ; initial entry point)
    }
    ...              ; rest of scatter-loading description
}

```

8.8.1 See also

Tasks

- [Creating root execution regions on page 8-13.](#)
- [Using the *FIXED* attribute to create root regions on page 8-16.](#)
- [About placing ARM C and C++ library code on page 8-51.](#)

Concepts

- [The image structure on page 4-3.](#)

8.9 Creating root execution regions

To specify a region as a root region in a scatter file you can:

- Specify **ABSOLUTE** as the attribute for the execution region, either explicitly or by permitting 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 an offset of zero (**+0**) is specified for all subsequent execution regions in the load region, then all execution regions not following an execution region containing **ZI** are also root regions.

The following example shows an implicitly defined root region:

Example 8-4 Implicit 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
    {
        * (+R0)      ; all R0 sections (must include section with
                    ; initial entry point)
    }
    ...              ; rest of scatter-loading description
}

```

- 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:

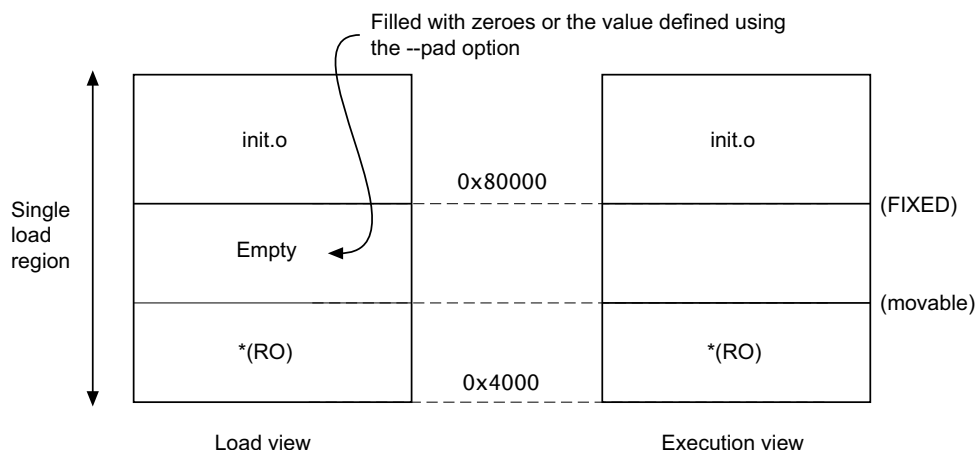


Figure 8-3 Memory map for fixed execution regions

The following example shows the corresponding scatter-loading description:

Example 8-5 Using the FIXED attribute

```

LR_1 0x040000          ; load region starts at 0x40000
{
    ER_RO 0x040000      ; start of execution region descriptions
    {
        ER_RO 0x040000  ; load address = execution address
        {
            * (+R0)      ; R0 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(+R0)   ; all R0 sections from init.o
        }
        ...              ; rest of scatter-loading description
    }
}

```

8.9.1 Examples of misusing the FIXED attribute

The following example shows common cases where the FIXED execution region attribute is misused:

Example 8-6 Misuse of the FIXED attribute

```

LR1 0x8000
{
    ER_LOW +0 0x1000
    {
        * (+R0)
    }
    ; 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
    {
        * (+R0)
    }
    ; The next available load address in LR_HIGH is 0x100000000. 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 0x100000000
}

```

8.9.2 See also

Tasks

- [Using the FIXED attribute to create root regions on page 8-16.](#)

Concepts

- [What is a root region? on page 8-12.](#)

armlink Reference Guide:

- [About load region descriptions on page 4-5.](#)
- [About execution region descriptions on page 4-10.](#)
- [Considerations when using a relative address +offset for load regions on page 4-18.](#)
- [Considerations when using a relative address +offset for execution regions on page 4-19.](#)

Reference

armlink Reference Guide:

- [Load region attributes on page 4-8.](#)
- [Execution region attributes on page 4-13.](#)
- [Address attributes for load and execution regions on page 4-16.](#)

armasm Reference Guide:

- [ENTRY on page 10-35.](#)

8.10 Using the FIXED attribute to create root regions

You can use the FIXED attribute in an execution region scatter file to create root regions that load and execute at fixed addresses.

Use FIXED to create multiple root regions within a single load region and therefore typically a single ROM device. For example, you can use this to place a function or a block of data, such as a constant table or a checksum, at a fixed address in ROM so that it can be accessed easily through pointers.

If you specify, for example, that some initialization code is to be placed at start of ROM and a checksum at the end of ROM, some of the memory contents might be unused. 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, it is suggested that you use the minimum amount of placement specifications in scatter files and leave the detailed placement of functions and data to the linker.

You cannot specify component objects that have been partially linked. For example, if you partially link the 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.

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.
-

8.10.1 See also

Concepts

armlink Reference Guide:

- [About execution region descriptions on page 4-10.](#)

Reference

armlink Reference Guide:

- [Load region attributes on page 4-8.](#)
- [Execution region attributes on page 4-13.](#)
- [Address attributes for load and execution regions on page 4-16.](#)

8.11 Methods of placing functions and data at specific addresses

Normally, the compiler produces RO, RW and ZI sections from a single source file. These regions contain all the code and data from the source file. 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.

The linker has two methods that enable you to place a section at a specific address:

- You can create a scatter file that defines an execution region at the required address with a section description that selects only one section.
- For a specially-named section the linker can get the placement address from the section name. These specially-named sections are called `__at` sections.

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:

- Place the function or data item in its own source file.
- Use `__attribute__((section("name")))` to place functions and variables in a named section.
- Use the AREA directive from assembly language. In assembly code, the smallest locatable unit is an AREA.
- Use the `-ffunction_sections` compiler option to generate one ELF section for each function in the source file.

This option results in a small increase in code size for some functions because it reduces the potential for sharing addresses, data, and string literals between functions. However, this can help to reduce the final image size overall by enabling the linker to remove unused functions when you specify `armlink --remove`.

8.11.1 See also

Tasks

- [Explicit placement of a named section with scatter-loading on page 8-24.](#)
- [Placement of sections at a specific address with `__attribute__\(\(section\(".ARM.__at_address"\)\)\)` on page 8-40.](#)

Concept

- [Restrictions on placing `__at` sections on page 8-41.](#)

Reference

armlink Reference Guide:

- [--autoat, --no_autoat on page 2-11.](#)
- [--map, --no_map on page 2-86.](#)
- [--scatter on page 2-110.](#)

armasm Reference Guide:

- [AREA on page 10-14.](#)

8.12 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:

1. Create the source file `main.c` containing the following code:

```
#include <stdio.h>

extern int sqr(int n1);
int gSquared __attribute__((section(".ARM.__at_0x5000"))); // Place at 0x5000

int main()
{
    gSquared=sqr(3);
    printf("Value squared is: %d\n", gSquared);
}
```

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 armv8a-arm-none-eabi -c -g function.c
armclang -target armv8a-arm-none-eabi -c -g main.c
armlink --force-scanlib --cpu=8-A.32 --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 `gSquared` is to be placed at the absolute address `0x5000`. `gSquared` 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: 0x00000000, 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 RW          7  .ARM.__AT_0x5000  main.o
```

8.12.1 See also

Tasks

- [Placing a variable in a named section with scatter-loading on page 8-20.](#)
- [Placing a variable at a specific address with scatter-loading on page 8-22.](#)
- [Explicit placement of a named section with scatter-loading on page 8-24.](#)
- [Placement of sections at a specific address with `__attribute__\(\(section\(".ARM.__at_address"\)\)\)` on page 8-40.](#)

Concept

- [Methods of placing functions and data at specific addresses on page 8-17.](#)
- [Restrictions on placing `__at` sections on page 8-41.](#)

Reference

armlink Reference Guide:

- [--map, --no_map](#) on page 2-86.

8.13 Placing 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:

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()
{
    gSquared=sqr(3);
    printf("Value squared is: %d\n", gSquared);
}
```

2. Create the source file `function.c` containing the following code:

```
int sqr(int n1)
{
    return n1*n1;
}
```

3. Create the scatter file `scatter.sc` containing the following load region:

```
LR1 0x0000 0x20000
{
    ER1 0x0 0x2000
    {
        *(+R0) ; rest of code and read-only data
    }
    ER2 0x8000 0x2000
    {
        main.o
    }
    ER3 0x10000 0x2000
    {
        function.o
        *(foo) ; Place gSquared in ER3
    }
    RAM 0x200000 (0x1FF00-0x2000) ; RW & ZI data to be placed at 0x200000
    {
        *(+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 armv8a-arm-none-eabi -c -g function.c
armclang -target armv8a-arm-none-eabi -c -g main.c
armlink --force-scanlib --cpu=8-A.32 --map --scatter=scatter.sc 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: 0x00001778, Max: 0x00020000, ABSOLUTE)
...
Execution Region ER3 (Base: 0x00010000, Size: 0x00000004, Max: 0x00020000, ABSOLUTE)

Base Addr      Size      Type  Attr    Idx    E Section Name      Object
0x00010000     0x0000000c  Code  RO      3      .text               function.o  0x00010000  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.

8.13.1 See also

Tasks

- [Placing a variable at a specific address without scatter-loading](#) on page 8-18.
- [Placing a variable at a specific address with scatter-loading](#) on page 8-22.
- [Explicit placement of a named section with scatter-loading](#) on page 8-24.
- [Placement of sections at a specific address with `__attribute__\(\(section\(".ARM.__at_address"\)\)\)`](#) on page 8-40.

Concept

- [Methods of placing functions and data at specific addresses](#) on page 8-17.
- [Restrictions on placing `__at` sections](#) on page 8-41.

Reference

armlink Reference Guide:

- [--autoat, --no_autoat](#) on page 2-11.
- [--map, --no_map](#) on page 2-86.
- [--scatter](#) on page 2-110.

8.14 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:

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()
{
    int squared;
    squared=sqr(gValue);
    printf("Value squared is: %d\n", squared);
}
```

2. Create the source file `function.c` containing the following code:

```
int sqr(int n1)
{
    return n1*n1;
}
```

3. Create the scatter file `scatter.sc` containing the following load region:

```
LR1 0x0
{
    ER1 0x0
    {
        *(+R0) ; rest of code and read-only data
    }
    ER2 +0
    {
        function.o
        *(.ARM.__at_0x10000) ; Place gValue at 0x10000
    }
    RAM 0x200000 (0x1FF00-0x2000) ; RW & ZI data to be placed at 0x200000
    {
        *(+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 armv8a-arm-none-eabi -c -g function.c
armclang -target armv8a-arm-none-eabi -c -g main.c
armlink --force-scanlib --cpu=8-A.32 --no_autoat --scatter=scatter.sc --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: 0x00002a1c, Size: 0x0000d5e8, Max: 0xffffffff, ABSOLUTE)

Base Addr    Size      Type  Attr    Idx    E Section Name      Object
0x00002a00 0x0000001c  Code  RO      3     .text               function.o
0x00002a1c 0x0000d5e4  PAD
0x00010000 0x00000004  Data  RO     29  .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`.

8.14.1 See also

Tasks

- [Placing a variable at a specific address without scatter-loading](#) on page 8-18.
- [Placing a variable in a named section with scatter-loading](#) on page 8-20.
- [Explicit placement of a named section with scatter-loading](#) on page 8-24.
- [Placement of sections at a specific address with `__attribute__\(\(section\(".ARM.__at_address"\)\)\)`](#) on page 8-40.

Concept

- [Methods of placing functions and data at specific addresses](#) on page 8-17.
- [Restrictions on placing `__at` sections](#) on page 8-41.

Reference

armlink Reference Guide:

- [--autoat, --no_autoat](#) on page 2-11.
- [--map, --no_map](#) on page 2-86.
- [--scatter](#) on page 2-110.

8.15 Explicit placement of a named section with scatter-loading

The following example shows how to place a named section explicitly using scatter-loading:

Example 8-7 Explicit section placement

```

LR1 0x0 0x10000
{
    ER1 0x0 0x2000                                ; Root Region, containing init code
    {
        init.o (INIT, +FIRST)                    ; place init code at exactly 0x0
        *(+RO)                                    ; rest of code and read-only data
    }
    RAM_RW 0x400000 (0x1FF00-0x2000) ; RW & ZI data to be placed at 0x400000
    {
        *(+RW)
    }
    RAM_ZI +0
    {
        *(+ZI)
    }
    DATABLOCK 0x1FF00 0xFF                    ; execution region at 0x1FF00
    {
        data.o(+RO-DATA)                        ; maximum space available for table is 0xFF
                                                ; 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`.

8.15.1 See also

Tasks

- [Using the *FIXED* attribute to create root regions on page 8-16.](#)

Concepts

armlink Reference Guide:

- [About load region descriptions on page 4-5.](#)
- [About execution region descriptions on page 4-10.](#)

Reference

armlink Reference Guide:

- [Load region attributes on page 4-8.](#)
- [Execution region attributes on page 4-13.](#)
- [Address attributes for load and execution regions on page 4-16.](#)

armasm Reference Guide:

- [ENTRY on page 10-35.](#)

8.16 Placement of unassigned sections with the .ANY module selector

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 use the .ANY module selector in the scatter file.

In most cases, using a single .ANY selector is equivalent to using the * module selector. However, unlike *, you can specify .ANY in multiple execution regions.

8.16.1 Default rules for placing unassigned sections

By default, the linker places unassigned sections using the following criteria:

- Place an unassigned section in the execution region that currently has the most free space. You can specify a maximum amount of space to use for unassigned sections with the execution region attribute ANY_SIZE.
- Sort sections in descending size order.

8.16.2 Placement rules when using multiple .ANY selectors

If more than one .ANY selector is present in a scatter file, the linker 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(+R0).

If several execution regions are equally specific, then the section is assigned to the execution region with the most available remaining space.

For example:

- If you have two equally specific execution regions where one has a size limit of 0x2000 and the other has no limit, then all the sections are assigned to the second unbounded .ANY region.
- If you have two equally specific execution regions where one has a size limit of 0x2000 and the other has a size limit of 0x3000, then the first sections to be placed are assigned to the second .ANY region of size limit 0x3000 until the remaining size of the second .ANY is reduced to 0x2000. From this point, sections are assigned alternately between both .ANY execution regions.

8.16.3 See also

Concepts

- [Prioritization of .ANY sections on page 8-27.](#)
- [Command-line options for controlling the placement of input sections for multiple .ANY selectors on page 8-28.](#)
- [Specifying the maximum size permitted for placing unassigned sections on page 8-29.](#)
- [Examples of using placement algorithms for .ANY sections on page 8-31.](#)
- [Example of next_fit algorithm showing behavior of full regions, selectors, and priority on page 8-33.](#)
- [Examples of using sorting algorithms for .ANY sections on page 8-35.](#)

armlink Reference Guide:

- [How the linker resolves multiple matches when processing scatter files on page 4-28.](#)
- [Behavior when .ANY sections overflow because of linker-generated content on page 4-30.](#)

Reference

armlink Reference Guide:

- [--any_contingency](#) on page 2-5.
- [--any_placement](#) on page 2-6.
- [--any_sort_order](#) on page 2-8.
- [--info](#) on page 2-61.
- [--map, --no_map](#) on page 2-86.
- [--section_index_display](#) on page 2-111.
- [--tiebreaker](#) on page 2-134.
- [Syntax of an input section description](#) on page 4-24.

8.17 Prioritization of .ANY sections

You can give a priority ordering if you have multiple .ANY sections with the .ANY $_{num}$ selector, where num is a positive integer from zero upwards. 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(+R0) ; evenly distributed with er3
    }
    er2 +0 256
    {
        .ANY2(+R0) ; Highest priority, so filled first
    }
    er3 +0 256
    {
        .ANY1(+R0) ; evenly distributed with er1
    }
}

```

8.17.1 See also

Concepts

- [Command-line options for controlling the placement of input sections for multiple .ANY selectors on page 8-28.](#)
- [Specifying the maximum size permitted for placing unassigned sections on page 8-29.](#)
- [Examples of using placement algorithms for .ANY sections on page 8-31.](#)
- [Example of next_fit algorithm showing behavior of full regions, selectors, and priority on page 8-33.](#)
- [Examples of using sorting algorithms for .ANY sections on page 8-35.](#)

armlink Reference Guide:

- [How the linker resolves multiple matches when processing scatter files on page 4-28.](#)
- [Behavior when .ANY sections overflow because of linker-generated content on page 4-30.](#)

8.18 Command-line options for controlling the placement of input sections for multiple .ANY selectors

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 overflow 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.

8.18.1 See also

Concepts

- [Prioritization of .ANY sections on page 8-27.](#)
- [Specifying the maximum size permitted for placing unassigned sections on page 8-29.](#)
- [Examples of using placement algorithms for .ANY sections on page 8-31.](#)
- [Example of next_fit algorithm showing behavior of full regions, selectors, and priority on page 8-33.](#)
- [Examples of using sorting algorithms for .ANY sections on page 8-35.](#)

armlink Reference Guide:

- [How the linker resolves multiple matches when processing scatter files on page 4-28.](#)
- [Behavior when .ANY sections overflow because of linker-generated content on page 4-30.](#)

8.19 Specifying the maximum size permitted for placing unassigned sections

The execution region attribute `ANY_SIZE max_size` enables you 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.
- You can use `ANY_SIZE` on a region without a `.ANY` selector but it is ignored by `armlink`.

When `ANY_SIZE` is present, `armlink`:

- Does not override a given `.ANY` size. That is, it does not reduce the priority then try to fit more sections in later.
- Never recalculates contingency.
- Never assigns sections in the contingency space.

`ANY_SIZE` does not require `--any_contingency` to be specified. However, when `--any_contingency` is specified and `ANY_SIZE` is not, `armlink` attempts to adjust contingencies. The aims are to:

- Never overflow a `.ANY` region.
- Never refuse to place a section in a contingency reserved space.

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.

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.

8.19.1 See also

Concepts

- [Prioritization of .ANY sections on page 8-27.](#)

- *Command-line options for controlling the placement of input sections for multiple .ANY selectors on page 8-28.*
- *Examples of using placement algorithms for .ANY sections on page 8-31.*
- *Example of next_fit algorithm showing behavior of full regions, selectors, and priority on page 8-33.*
- *Examples of using sorting algorithms for .ANY sections on page 8-35.*

armlink Reference Guide:

- *How the linker resolves multiple matches when processing scatter files on page 4-28.*
- *Behavior when .ANY sections overflow because of linker-generated content on page 4-30.*

8.20 Examples of using placement algorithms for .ANY sections

These examples show the operation of the placement algorithms for R0-CODE sections in sections.o.

The input section properties and ordering are shown in the following table:

Table 8-1 Input section properties

Name	Size
sec1	0x4
sec2	0x4
sec3	0x4
sec4	0x4
sec5	0x4
sec6	0x4

The scatter file used for the examples is:

```
LR 0x100
{
  ER_1 0x100 0x10
  {
    .ANY
  }

  ER_2 0x200 0x10
  {
    .ANY
  }
}
```

Note

These examples have --any_contingency disabled.

8.20.1 Example for first_fit, next_fit, and best_fit

This example shows the situation 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	R0	1	sec1	sections.o
0x00000104	0x00000004	Code	R0	2	sec2	sections.o
0x00000108	0x00000004	Code	R0	3	sec3	sections.o
0x0000010c	0x00000004	Code	R0	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 completely 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.

8.20.2 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 and earlier linker versions.

8.20.3 See also

Concepts

- [Placement of unassigned sections with the .ANY module selector on page 8-25.](#)
- [Example of next_fit algorithm showing behavior of full regions, selectors, and priority on page 8-33.](#)

Reference

armlink Reference Guide:

- [--any_placement on page 2-6.](#)
- [--scatter on page 2-110.](#)

8.21 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 R0-CODE sections in sections.o.

The input section properties and ordering are shown in the following table:

Table 8-2 Input section properties

Name	Size
sec1	0x4
sec2	0x4
sec3	0x4
sec4	0x4
sec5	0x4
sec6	0x4

The scatter file used for the examples is:

```
LR 0x100
{
  ER_1 0x100 0x20
  {
    .ANY1(+R0-CODE)
  }

  ER_2 0x200 0x20
  {
    .ANY2(+R0)
  }

  ER_3 0x300 0x20
  {
    .ANY3(+R0)
  }
}
```

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	R0	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.

8.21.1 See also

Concepts

- [Placement of unassigned sections with the .ANY module selector](#) on page 8-25.
- [Examples of using placement algorithms for .ANY sections](#) on page 8-31.

armlink Reference Guide:

- [How the linker resolves multiple matches when processing scatter files](#) on page 4-28.
- [Behavior when .ANY sections overflow because of linker-generated content](#) on page 4-30.

Reference

armlink Reference Guide:

- [--any_placement](#) on page 2-6.
- [--scatter](#) on page 2-110.

8.22 Examples of using sorting algorithms for .ANY sections

These examples show the operation of the sorting algorithms for R0-CODE sections in sections_a.o and sections_b.o.

The input section properties and ordering are shown in the following tables:

Table 8-3 Input section properties for sections_a.o

Name	Size
seca_1	0x4
seca_2	0x4
seca_3	0x10
seca_4	0x14

Table 8-4 Input section properties for sections_b.o

Name	Size
secb_1	0x4
secb_2	0x4
secb_3	0x10
secb_4	0x14

8.22.1 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 order that the sections are processed by the .ANY assignment algorithm is:

Table 8-5 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

Sections of the same size use the tiebreak specified by --tiebreaker.

8.22.2 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 order that the sections are processed by the .ANY assignment algorithm is:

Table 8-6 Sort order for cmdline algorithm

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

Sections with the same command-line index use the tiebreak specified by `--tiebreaker`.

8.22.3 See also

Concepts

- [Placement of unassigned sections with the .ANY module selector on page 8-25.](#)

Reference

armlink Reference Guide:

- [--any_sort_order on page 2-8.](#)
- [--scatter on page 2-110.](#)
- [--tiebreaker on page 2-134.](#)

8.23 Selecting veneer input sections in scatter-loading descriptions

Veneers allow switching between A32 and T32 code or allow a longer program jump than can be specified in a single instruction. You can place veneers at a specific location by including 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.

8.23.1 See also

Concepts

- [Overview of veneers on page 4-25.](#)

8.24 Placement of code and data with `__attribute__((section("name")))`

You can place code and data by separating them into their own objects without having to use toolchain-specific pragmas or attributes. However, you can also use `__attribute__((section("name")))` to place an item in a separate ELF section. You can then use a scatter file to place the named sections at specific locations.

To use `__attribute__((section("name")))` to place a variable in a separate section:

1. Use `__attribute__((section("name")))` to specify the named section where the variable is to be placed, for example:

Example 8-8 Naming a section

```
int variable __attribute__((section("foo"))) = 10;
```

2. Use a scatter file to place the named section, for example:

Example 8-9 Placing a section

```
FLASH 0x24000000 0x4000000
{
    ...                               ; rest of code

    ADDER 0x08000000
    {
        file.o (foo)                 ; select section foo from file.o
    }
}
```

The following example shows the memory map for the FLASH load region:

```
...
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      16     foo                  file.o
...
```

Be aware of the following:

- Linking with `--autoat` or `--no_autoat` does not affect the placement.
- If scatter-loading is not used, the section is placed in the default `ER_RW` execution region of the `LR_1` load region.
- If you have a scatter file that does not include the `foo` selector, then the section is placed 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;  
}
```

8.24.1 See also

Reference

- [Placement of sections at a specific address with __attribute__\(\(section\(".ARM.__at_address"\)\)\)](#) on page 8-40.
- [Restrictions on placing __at sections](#) on page 8-41.

armlink Reference Guide:

- [--autoat, --no_autoat](#) on page 2-11.
- [--scatter](#) on page 2-110.

8.25 Placement of sections at a specific address with `__attribute__((section(".ARM.__at_address")))`

You can give a section a special name that encodes the address where it must be placed. You specify the name as follows:

`.ARM.__at_address`

Where:

`address` is the required address of the section. You can specify this in hexadecimal or decimal. Sections in the form of `.ARM.__at_address` are referred to by the abbreviation `__at`.

In the compiler, you can assign variables to `__at` sections by explicitly naming the section with `__attribute__((section("name")))`.

Example 8-10 Assigning variables to `__at` sections in C or C++ code

```
// place variable2 in a section called .ARM.__at_0x8000
int variable2 __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. 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.

8.25.1 See also

Concepts

- [Methods of placing functions and data at specific addresses](#) on page 8-17.
- [Placement of code and data with `__attribute__\(\(section\("name"\)\)\)`](#) on page 8-38.
- [Restrictions on placing `__at` sections](#) on page 8-41.
- [Automatic placement of `__at` sections](#) on page 8-42.
- [Manual placement of `__at` sections](#) on page 8-44.
- [Placement of a key in flash memory with `__at`](#) on page 8-46.

Reference

armlink Reference Guide:

- `--autoat`, `--no_autoat` on page 2-11.

8.26 Restrictions on placing __at sections

The following restrictions apply when placing __at sections at specific addresses:

- __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 have an address that is a multiple of their alignment.
- __at sections ignore any +FIRST or +LAST ordering constraints.

8.26.1 See also

Concepts

- *Placement of sections at a specific address with __attribute__((section(".ARM.__at_address")))* on page 8-40.

Other information

- *Base Platform ABI for the ARM Architecture*
<http://infocenter.arm.com/help/topic/com.arm.doc.ih0037-/index.html>.

8.27 Automatic placement of __at sections

The automatic placement of __at sections is enabled by default. This feature is controlled by the linker command-line option, --autoat.

Note

You cannot use __at section placement with position independent execution regions.

When linking with the --autoat option, the __at sections are not placed by the 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 --autoat created execution regions 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 --autoat created execution region must have a base address that is at least 4 byte-aligned. The linker produces an error message if any region is incorrectly aligned.

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, 10240 bytes.
- 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:

Example 8-11 Placement of the variables in C or C++ code

```
// place the RW 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 foo __attribute__((section(".ARM.__at_0x4000"))) = 100;
```

Example 8-12 Automatic placement of __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.

8.27.1 See also

Concepts

- [Placement of code and data with __attribute__\(\(section\("name"\)\)\)](#) on page 8-38.
- [Placement of sections at a specific address with __attribute__\(\(section\(".ARM.__at_address"\)\)\)](#) on page 8-40.
- [Restrictions on placing __at sections](#) on page 8-41.
- [Manual placement of __at sections](#) on page 8-44.
- [Placement of a key in flash memory with __at](#) on page 8-46.

armlink Reference Guide:

- [About execution region descriptions](#) on page 4-10.

Reference

armlink Reference Guide:

- [--autoat, --no_autoat](#) on page 2-11.
- [--ro_base](#) on page 2-106.
- [--rw_base](#) on page 2-108.
- [--zi_base](#) on page 2-152.
- [Execution region attributes](#) on page 4-13.

8.28 Manual placement of __at sections

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.

Example 8-13 Placement of the variables in C or C++ code

```
// place the RO variable in a section called .ARM.__at_0x2000
const int F00 __attribute__((section(".ARM.__at_0x2000"))) = 100;

// place the RW variable in a section called .ARM.__at_0x4000
int bar __attribute__((section(".ARM.__at_0x4000")));
```

Example 8-14 Manual placement of __at sections

```
LR1 0x0
{
    ER_R0 0x0 0x2000
    {
        *(+R0)                ; .ARM.__at_0x0 is selected by +R0
    }
    ER_R02 0x2000
    {
        *(.ARM.__at_0x2000)    ; .ARM.__at_0x2000 is selected by the section named
                                ; .ARM.__at_0x2000
    }
    ER2 0x4000
    {
        *(+RW +ZI)            ; .ARM.__at_0x4000 is selected by +RW
    }
}
```

8.28.1 See also

Concepts

- [Placement of code and data with __attribute__\(\(section\("name"\)\)\)](#) on page 8-38.
- [Placement of sections at a specific address with __attribute__\(\(section\(".ARM.__at_address"\)\)\)](#) on page 8-40.
- [Restrictions on placing __at sections](#) on page 8-41.
- [Automatic placement of __at sections](#) on page 8-42.
- [Placement of a key in flash memory with __at](#) on page 8-46.

armlink Reference Guide:

- [About execution region descriptions](#) on page 4-10.

Reference

armlink Reference Guide:

- [--autoat, --no_autoat](#) on page 2-11.
- [Execution region attributes](#) on page 4-13.

8.29 Placement of a key in flash memory with `__at`

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.

Assuming 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`.

Example 8-15 Placement of the flash key variable in C or C++ code

```
// place flash_key in a section called .ARM.__at_0x8000
long flash_key __attribute__((section(".ARM.__at_0x8000")));
```

The following example shows a scatter file with manual placement of the flash execution region:

Example 8-16 Manual placement of flash execution regions

```
ER_FLASH 0x8000 0x2000
{
    *(+R0)
    *(.ARM.__at_0x8000) ; key
}
```

Use the linker command-line option `--no_autoat` to enable manual placement.

The following example shows a scatter file with automatic placement of the flash execution region. Use the linker command-line option `--autoat` to enable automatic placement.

Example 8-17 Automatic placement of flash execution regions

```
ER_FLASH 0x8000 0x2000
{
    *(+R0) ; other code and read-only data, the
           ; __at section is automatically selected
}
```

8.29.1 See also

Tasks

- [Placing sections with *FIRST* and *LAST* attributes on page 4-20.](#)
- [Placement of sections at a specific address with `__attribute__\(\(section\(".ARM.__at_address"\)\)\)` on page 8-40.](#)

Concepts

- [Automatic placement of `__at` sections on page 8-42.](#)
- [Manual placement of `__at` sections on page 8-44.](#)

armlink Reference Guide:

- [About execution region descriptions on page 4-10.](#)

Reference

armlink Reference Guide:

- [--autoat, --no_autoat](#) on page 2-11.

8.30 Placement of sections with overlays

You can use the OVERLAY attribute in a scatter file to place multiple execution regions at the same address. An overlay manager is required to make sure that only one execution region is instantiated at a time. The ARM Compiler toolchain does not provide an overlay manager.

The following example shows the definition of a static section in RAM followed by a series of overlays. Here, only one of these sections is instantiated at a time.

Example 8-18 Specifying a root region

```

EMB_APP 0x8000
{
    .
    .
    STATIC_RAM 0x0                ; contains most of the RW and ZI code/data
    {
        * (+RW,+ZI)
    }
    OVERLAY_A_RAM 0x1000 OVERLAY    ; start address of overlay...
    {
        module1.o (+RW,+ZI)
    }
    OVERLAY_B_RAM 0x1000 OVERLAY
    {
        module2.o (+RW,+ZI)
    }
    ...                            ; rest of scatter-loading description...
}

```

A region marked as OVERLAY is not initialized by the C library at startup. The contents of the memory used by the overlay region are the responsibility of an overlay manager. If the region contains initialized data, use the NOCOMPRESS attribute to prevent RW data compression.

You can use the linker defined symbols to obtain the addresses required to copy the code and data.

The OVERLAY attribute can be used on a single region that is not the same address as a different region. Therefore, an overlay region can be used as a method to prevent the initialization of particular regions by the C library startup code. As with any overlay region these must be manually initialized in your code.

An overlay region can have a relative base. The behavior of an overlay region with a *+offset* base address depends on the regions that precede it and the value of *+offset*. The linker places consecutive *+offset* regions at the same base address if they have the same *+offset* value.

When a *+offset* execution region ER follows a contiguous overlapping block of overlay execution regions the base address of ER is:

limit address of the overlapping block of overlay execution regions + *offset*

The following table shows the effect of *+offset* when used with the OVERLAY attribute. REGION1 appears immediately before REGION2 in the scatter file:

Table 8-7 Using relative offset in overlays

REGION1 is set with OVERLAY	+offset	REGION2 Base Address
NO	<offset>	REGION1 Limit + <offset>
YES	+0	REGION1 Base Address
YES	<none-zero offset>	REGION1 Limit + <none-zero offset>

The following example shows the use of relative offsets with overlays and the effect on execution region addresses:

Example 8-19 Example of relative offset in overlays

```

EMB_APP 0x8000{
    CODE 0x8000
    {
        *(+R0)
    }

    # REGION1 Base = CODE limit
    REGION1 +0 OVERLAY
    {
        module1.o(*)
    }

    # REGION2 Base = REGION1 Base
    REGION2 +0 OVERLAY
    {
        module2.o(*)
    }

    # REGION3 Base = REGION2 Base = REGION1 Base
    REGION3 +0 OVERLAY
    {
        module3.o(*)
    }

    # REGION4 Base = REGION3 Limit + 4
    Region4 +4 OVERLAY
    {
        module4.o(*)
    }
}

```

If the length of the non-overlay area is unknown, you can use a zero relative offset to specify the start address of an overlay so that it is placed immediately after the end of the static section.

You can use the following command-line options to add extra debug information to the image:

- `--emit_debug_overlay_relocs.`
- `--emit_debug_overlay_section.`

These permit an overlay-aware debugger to track which overlay is currently active.

8.30.1 See also

Concepts

- *Placement of sections at a specific address with `__attribute__((section(".ARM.__at_address")))` on page 8-40.*

armlink Reference Guide:

- *About load region descriptions on page 4-5.*
- *About execution region descriptions on page 4-10.*
- *Considerations when using a relative address +offset for load regions on page 4-18.*
- *Considerations when using a relative address +offset for execution regions on page 4-19.*

Reference

- *Accessing linker-defined symbols on page 7-4.*

armlink Reference Guide:

- *--emit_debug_overlay_relocs on page 2-42.*
- *--emit_debug_overlay_section on page 2-43.*
- *Load region attributes on page 4-8.*
- *Execution region attributes on page 4-13.*
- *Address attributes for load and execution regions on page 4-16.*

Other information

- *ABI for the ARM Architecture: Support for Debugging Overlaid Programs*
<http://infocenter.arm.com/help/topic/com.arm.doc.ih0049-/index.html>.

8.31 About placing ARM C and C++ library code

You can place code from the ARM standard C and C++ libraries in a scatter file. Use `*armlib` or `*cpplib` 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`.

8.31.1 See also

Tasks

- [Creating root execution regions on page 8-13.](#)
- [Using the *FIXED* attribute to create root regions on page 8-16.](#)

Concepts

- [What is a root region? on page 8-12.](#)
- [Example of placing code in a root region on page 8-52.](#)
- [Example of placing ARM C library code on page 8-53.](#)
- [Example of placing ARM C++ library code on page 8-54.](#)

8.32 Example of placing code in a root region

Use a scatter file to specify a root section in the same way as a named section. The following example uses the section selector `InRoot$$Sections` to place all sections that must be in a root region:

Example 8-20 Specifying a root region

```

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
  }
}

```

8.32.1 See also

Tasks

- [Creating root execution regions on page 8-13.](#)
- [Using the *FIXED* attribute to create root regions on page 8-16.](#)

Concepts

- [What is a root region? on page 8-12.](#)
- [About placing ARM C and C++ library code on page 8-51.](#)
- [Example of placing ARM C library code on page 8-53.](#)
- [Example of placing ARM C++ library code on page 8-54.](#)

8.33 Example of placing ARM C library code

The following example shows how to place C library code:

Example 8-21 Placing ARM C library code

```

ROM1 0
{
    * (InRoot$$Sections)
    * (+R0)
}
ROM2 0x1000
{
    *armlib/c_* (+R0)                ; all ARM-supplied C library functions
}
ROM3 0x2000
{
    *armlib/h_* (+R0)                ; just the ARM-supplied __ARM_*
                                    ; redistributable library functions
}
RAM1 0x3000
{
    *armlib* (+R0)                   ; 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 `install_directory\ARMCompiler6\lib\armlib` directory.

8.33.1 See also

Concepts

- [About placing ARM C and C++ library code on page 8-51.](#)
- [Example of placing code in a root region on page 8-52.](#)
- [Example of placing ARM C++ library code on page 8-54.](#)

Reference

ARM C and C++ Libraries and Floating Point Support User Guide:

- [C and C++ library naming conventions on page 2-115.](#)

8.34 Example of placing ARM C++ library code

The following is a C++ program that is to be scatter-loaded:

```
#include <iostream>

using namespace std;

extern "C" int foo ()
{
    cout << "Hello" << endl;
    return 1;
}
```

To place the C++ library code, define the scatter file as follows:

```
LR 0x0
{
    ER1 0x0
    {
        *armlib*(+R0)
    }

    ER2 +0
    {
        *cpplib*(+R0)
        *(.init_array) ; Section .init_array must be placed explicitly,
                        ; otherwise it is shared between two regions, and
                        ; the linker is unable to decide where to place it.
    }

    ER3 +0
    {
        *(+R0)
    }

    ER4 +0
    {
        *(+RW,+ZI)
    }
}
```

The name *install_directory*\ARMCompiler6\lib\armlib indicates the ARM C library files that are located in the armlib directory.

The name *install_directory*\ARMCompiler6\lib\cpplib indicates the ARM C++ library files that are located in the cpplib directory.

8.34.1 See also

Concepts

- [About placing ARM C and C++ library code on page 8-51.](#)
- [Example of placing code in a root region on page 8-52.](#)
- [Example of placing ARM C library code on page 8-53.](#)

Reference

ARM C and C++ Libraries and Floating Point Support User Guide:

- [C and C++ library naming conventions on page 2-115.](#)

8.35 Reserving an empty region

You can use the `EMPTY` attribute in an execution region scatter-loading description to reserve an empty block of memory for the stack.

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 must be an absolute address and not a relative one.

In the following example, the execution region definition `STACK 0x800000 EMPTY -0x10000` defines a region called `STACK` that starts at address `0x7F0000` and ends at address `0x800000`:

Example 8-22 Reserving a region for the stack

```

LR_1 0x800000                                ; load region starts at 0x800000
{
    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.

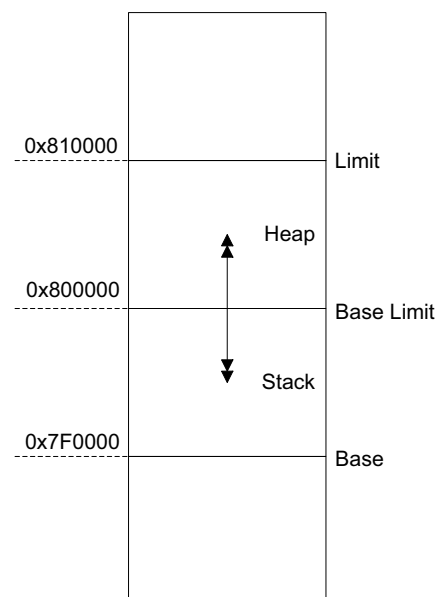


Figure 8-4 Reserving a region for the stack

In this example, the linker generates the 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 used in a load region definition.

The linker checks that the address space used for the EMPTY region does not coincide with any other execution region.

8.35.1 See also

Concepts

armlink Reference Guide:

- [About execution region descriptions on page 4-10.](#)

Reference

- [Image\\$\\$ execution region symbols on page 7-6.](#)

armlink Reference Guide:

- [Execution region attributes on page 4-13.](#)

8.36 About creating regions on page boundaries

You can produce an ELF file that can be loaded directly to a target 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`.
- `GetPageSize`.

Note

Alignment on an execution region causes both the load address and execution address to be aligned.

The following example produces an ELF file with each execution region starting on a new page:

Example 8-23 Creating regions on page boundaries

```
LR1 GetPageSize() + SizeOfHeaders()
{
    ER_RO +0
    {
        *(+R0)
    }
    ER_RW +GetPageSize()
    {
        *(+RW)
    }
    ER_ZI +0
    {
        *(+ZI)
    }
}
```

The default page size 0x8000, is used. You can change the page size with the `--pagesize` command-line option.

8.36.1 See also

Concepts

- [Demand paging on page 4-22](#).
- [Overalignment of execution regions and input sections on page 8-59](#).
- [Expression evaluation in scatter files on page 8-60](#).
- [Using expression evaluation in a scatter file to avoid padding on page 8-61](#).

Reference

armlink Reference Guide:

- [--pagesize on page 2-96](#).
- [Load region attributes on page 4-8](#).
- [Execution region attributes on page 4-13](#).
- [Example of aligning a base address in execution space but still tightly packed in load space on page 4-43](#).
- [AlignExpr\(expr, align\) function on page 4-44](#).

- [*GetPageSize\(\)* function](#) on page 4-45.

8.37 Overallignment of execution regions and input sections

There are situations when you want to overalign code and data sections. How you deal with them depends on whether or not you have access to the source code:

- If you have access to the original source code, you can do this at compile time with the `__align(n)` keyword for example.
- If you do not have access to 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
}
```

8.37.1 See also

Concepts

- [About creating regions on page boundaries on page 8-57.](#)

Reference

armlink Reference Guide:

- [Execution region attributes on page 4-13.](#)
- [About input section descriptions on page 4-23.](#)

8.38 Expression evaluation in scatter files

The linker can carry out simple expression evaluation with a restricted set of operators. The operators are +, -, *, /, AND, OR, and parentheses. The implementation of OR and AND follows C operator precedence rules.

8.38.1 Example of using expression evaluation

Use the directives:

```
#define BASE_ADDRESS 0x8000
#define ALIAS_NUMBER 0x2
#define ALIAS_SIZE 0x400

#define AN_ADDRESS (BASE_ADDRESS+(ALIAS_NUMBER*ALIAS_SIZE))
```

The scatter file might contain:

```
LOAD_FLASH AN_ADDRESS    ; start address
```

After preprocessing, this evaluates to:

```
LOAD_FLASH ( 0x8000 + ( 0x2 * 0x400 )) ; start address
```

After evaluation, the linker parses the scatter file to produce the load region:

```
LOAD_FLASH 0x8800 ; start address
```

8.38.2 See also

Concepts

- [Using expression evaluation in a scatter file to avoid padding on page 8-61.](#)

8.39 Using expression evaluation in a scatter file to avoid padding

Using the `ALIGN`, `ALIGNALL`, and `FIXED` attributes in a scatter file can result in a large amount of padding in the image.

To remove this padding, 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.

8.39.1 Example to avoid padding in scatter file

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`.

8.39.2 See also

Concepts

- [Expression evaluation in scatter files on page 8-60.](#)

armlink Reference Guide:

- [Example of aligning a base address in execution space but still tightly packed in load space on page 4-43.](#)

Reference

armlink Reference Guide:

- [Execution region attributes on page 4-13.](#)
- [AlignExpr\(expr, align\) function on page 4-44.](#)

8.40 Equivalent scatter-loading descriptions for simple images

The command-line options `--reloc`, `--ro_base`, `--rw_base`, and `--split` create the simple image types:

- 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.

8.40.1 See also

Concepts

- [Types of simple image on page 4-9.](#)
- [Type 1 image, one load region and contiguous execution regions on page 8-63.](#)
- [Type 2 image, one load region and non-contiguous execution regions on page 8-65.](#)
- [Type 3 image, two load regions and non-contiguous execution regions on page 8-67.](#)

armlink Reference Guide:

- [About load region descriptions on page 4-5.](#)

Reference

armlink Reference Guide:

- [--reloc on page 2-103.](#)
- [--ro_base on page 2-106.](#)
- [--rw_base on page 2-108.](#)
- [--scatter on page 2-110.](#)
- [--split on page 2-118.](#)
- [Load region attributes on page 4-8.](#)

8.41 Type 1 image, one load region and contiguous execution regions

An image of this type consists of a single load region in the load view and three execution regions in the execution view. The execution regions are placed contiguously in the memory map.

--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:

Example 8-24 Single load region and contiguous execution regions

```

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.
                  ; However, since 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, 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.

8.41.1 See also

Concepts

armlink Reference Guide:

- [About load region descriptions on page 4-5.](#)
- [Considerations when using a relative address +offset for load regions on page 4-18.](#)

- [Considerations when using a relative address +offset for execution regions on page 4-19.](#)

Reference

armlink Reference Guide:

- [--reloc on page 2-103.](#)
- [--ro_base on page 2-106.](#)
- [Load region attributes on page 4-8.](#)

8.42 Type 2 image, one load region and non-contiguous execution regions

An image of this type 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=address1 specifies the load and execution address of the region containing the RO output section. --rw_base=address2 specifies the execution address for the RW execution region.

The following example shows the scatter-loading description equivalent to using --ro_base=0x010000 --rw_base=0x040000:

Example 8-25 Single load region and multiple execution regions

```

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.

8.42.1 See also

Concepts

armlink Reference Guide:

- [About load region descriptions on page 4-5.](#)
- [Considerations when using a relative address +offset for load regions on page 4-18.](#)
- [Considerations when using a relative address +offset for execution regions on page 4-19.](#)

Reference

armlink Reference Guide:

- [--ro_base on page 2-106.](#)

- *--rw_base* on page 2-108.
- *Load region attributes* on page 4-8.

8.43 Type 3 image, two load regions and non-contiguous execution regions

Type 3 images consist of two load regions in load view and three 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 two load regions.

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.

Note

The `--reloc` option and RELOC attribute are not supported for AArch64 state.

`--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.

`--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.

The following example shows the scatter-loading description equivalent to using `--ro_base=0x010000 --rw_base=0x040000 --split`:

Example 8-26 Multiple load regions

```
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_R0, because its execution address is 0x040000.
- The ER_ZI execution region is placed immediately following the end of the preceding execution region, ER_RW.

8.43.1 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`:

Example 8-27 Relocatable load regions

```

LR_1 0x010000 RELOC
{
    ER_R0 + 0
    {
        * (+R0)
    }
}

LR2 0x040000 RELOC
{
    ER_RW + 0
    {
        * (+RW)
    }

    ER_ZI +0
    {
        * (+ZI)
    }
}

```

8.43.2 See also

Concepts

armlink Reference Guide:

- [About load region descriptions on page 4-5.](#)
- [Considerations when using a relative address +offset for load regions on page 4-18.](#)
- [Considerations when using a relative address +offset for execution regions on page 4-19.](#)

Reference

armlink Reference Guide:

- [--reloc on page 2-103.](#)
- [--ro_base on page 2-106.](#)
- [--rw_base on page 2-108.](#)
- [--split on page 2-118.](#)
- [Load region attributes on page 4-8.](#)
- [Address attributes for load and execution regions on page 4-16.](#)

8.44 Scatter file to ELF mapping

For simple images, 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 up to three ELF Sections:
 - One for RO.
 - One for RW.
 - One for ZI.

For example, you might have a scatter file similar to the following:

Example 8-28 Scatter file

```

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 --cpu=8-A.32 --text -v my_image.axf
```

To display the symbol table, enter the command:

```
fromelf --cpu=8-A.32 --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:

Example 8-29 Scatter-loaded sections in the ELF image

```

...
=====

** 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)
...

8.44.1 See also

Concepts

- [About scatter-loading on page 8-3.](#)
- [Images with a simple memory map on page 8-6.](#)

Chapter 9

Base Platform Application Binary Interface Support

The following topics describe how the linker, `arm1ink`, supports the *Base Platform Application Binary Interface* (BPABI):

Concepts

- *About the Base Platform Application Binary Interface (BPABI) on page 9-3.*
- *Platforms supported by the BPABI on page 9-4.*
- *Concepts common to all BPABI models on page 9-5.*
- *About importing and exporting symbols for BPABI models on page 9-6.*
- *Symbol visibility for BPABI models on page 9-7.*
- *Automatic import and export for BPABI models on page 9-8.*
- *Manual import and export for BPABI models on page 9-9.*
- *Symbol versioning for BPABI models on page 9-10.*
- *RW compression for BPABI models on page 9-11.*
- *Linker options for bare metal and DLL-like models on page 9-12.*
- *Bare metal and DLL-like memory model on page 9-13.*
- *Mandatory symbol versioning in the BPABI DLL-like model on page 9-14.*
- *Automatic dynamic symbol table rules in the BPABI DLL-like model on page 9-15.*
- *Addressing modes in the BPABI DLL-like model on page 9-16.*
- *C++ initialization in the BPABI DLL-like model on page 9-17.*
- *About symbol versioning on page 9-18.*
- *Symbol versioning script file on page 9-19.*
- *Example of creating versioned symbols on page 9-20.*
- *About embedded symbols on page 9-21.*
- *Linker options for enabling implicit symbol versioning on page 9-22.*

Reference

- [*Related linker command-line options for the BPABI DLL-like model*](#) on page 9-23.

9.1 About the Base Platform Application Binary Interface (BPABI)

Many embedded systems use an operating system 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. Other more complex *Operating Systems* (OS) are referred to as a platform OS, for example, ARM Linux. These have the ability to load applications and shared libraries on demand.

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 *Base Platform Application Binary Interface* (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.

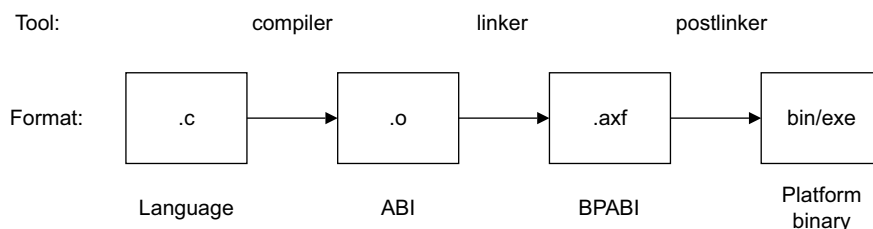


Figure 9-1 BPABI tool flow

9.1.1 See also

Concepts

- [Platforms supported by the BPABI on page 9-4.](#)
- [Concepts common to all BPABI models on page 9-5.](#)

Other information

- [Base Platform ABI for the ARM Architecture](#)
<http://infocenter.arm.com/help/topic/com.arm.doc.ih0037-/index.html>.

9.2 Platforms supported by the BPABI

The *Base Platform Application Binary Interface* (BPABI) defines three platform models based on the type of shared library:

- 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.

9.2.1 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.

9.2.2 See also

Concepts

- [About the Base Platform Application Binary Interface \(BPABI\) on page 9-3.](#)
- [Concepts common to all BPABI models on page 9-5.](#)

Reference

armlink Reference Guide:

- [--dll on page 2-38.](#)

9.3 Concepts 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 concepts are common to all BPABI models:

- Symbol importing.
- Symbol exporting.
- Versioning.
- Visibility of symbols.

9.3.1 See also

Concepts

- [About importing and exporting symbols for BPABI models on page 9-6.](#)
- [Symbol visibility for BPABI models on page 9-7.](#)
- [Automatic import and export for BPABI models on page 9-8.](#)
- [Manual import and export for BPABI models on page 9-9.](#)
- [Symbol versioning for BPABI models on page 9-10.](#)
- [RW compression for BPABI models on page 9-11.](#)

9.4 About importing and exporting symbols for BPABI models

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.

9.4.1 See also

Concepts

- [Linker options for bare metal and DLL-like models on page 9-12.](#)
- [About symbol versioning on page 9-18.](#)

9.5 Symbol visibility for 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 the symbol is a reference, the visibility controls the definitions that the linker can use to define the symbol. If the 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 9-1 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_HIDDENSTV_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 *Base Platform Application Binary Interface* (BPABI). The platform owner defines how this works. See the documentation for your specific platform for more information.

9.5.1 See also

Concepts

- [Optimization with RW data compression on page 5-6.](#)
- [Linker options for bare metal and DLL-like models on page 9-12.](#)
- [About symbol versioning on page 9-18.](#)

Reference

armlink Reference Guide:

- [--keep_protected_symbols on page 2-72.](#)
- [--max_visibility on page 2-89.](#)
- [--override_visibility on page 2-93.](#)
- [--use_definition_visibility on page 2-138.](#)
- [EXPORT on page 3-2.](#)
- [IMPORT on page 3-4.](#)
- [REQUIRE on page 3-7.](#)

armasm Reference Guide:

- [EXPORT or GLOBAL on page 10-37.](#)

9.6 Automatic import and export for BPABI models

The linker can automatically import and export symbols. 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.

9.6.1 See also

Concepts

- [Concepts common to all BPABI models on page 9-5.](#)
- [Linker options for bare metal and DLL-like models on page 9-12.](#)
- [About symbol versioning on page 9-18.](#)

9.7 Manual import and export for BPABI models

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.`

9.7.1 See also

Concepts

- [What is a steering file? on page 7-23.](#)

Reference

armlink Reference Guide:

- [EXPORT on page 3-2.](#)
- [IMPORT on page 3-4.](#)
- [REQUIRE on page 3-7.](#)

9.8 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.

9.8.1 See also

Concepts

- [About symbol versioning on page 9-18.](#)

9.9 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.

9.9.1 See also

Concepts

- [Optimization with RW data compression on page 5-6.](#)

9.10 Linker options for bare metal and DLL-like models

Use the following command-line options to build bare metal executables and *dynamically linked library* (DLL) like models for a platform OS:

Table 9-2 Turning on BPABI support

Command-line options	Description
--base_platform	to use scatter-loading with <i>Base Platform ABI</i> (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.

If you are developing applications or DLL for a specific platform OS, based around the *Base Platform Application Binary Interface* (BPABI), you must use the following information in conjunction with the platform documentation:

- Bare metal and DLL-like 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.

9.10.1 See also

Concepts

- [Bare metal and DLL-like memory model](#) on page 9-13.
- [Mandatory symbol versioning in the BPABI DLL-like model](#) on page 9-14.
- [Automatic dynamic symbol table rules in the BPABI DLL-like model](#) on page 9-15.
- [Addressing modes in the BPABI DLL-like model](#) on page 9-16.
- [C++ initialization in the BPABI DLL-like model](#) on page 9-17.
- [Related linker command-line options for the BPABI DLL-like model](#) on page 9-23.

Reference

- [--base_platform](#) on page 2-12.
- [--bpabi](#) on page 2-16.
- [--dll](#) on page 2-38.

Other information

- [Base Platform ABI for the ARM Architecture](#)
<http://infocenter.arm.com/help/topic/com.arm.doc.ih0037-/index.html>.

9.11 Bare metal and DLL-like memory model

Base Platform Application Binary Interface (BPABI) files have a standard memory model that is described in the BPABI specification. By using the `--bpabi` command-line option, the linker automatically applies this 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)
    }
}
```

Note

The DLL-like model is not supported for AArch64 state.

9.11.1 Customizing the memory model

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`.

9.11.2 See also

Concepts

- [Base Platform linking model on page 3-6.](#)

Reference

armlink Reference Guide:

- [--base_platform on page 2-12.](#)
- [--ro_base on page 2-106.](#)
- [--rosplit on page 2-107.](#)
- [--rw_base on page 2-108.](#)

9.12 Mandatory symbol versioning in the BPABI DLL-like model

The *Base Platform Application Binary Interface* (BPABI) DLL-like model requires static binding. 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.

9.12.1 See also

Concepts

- [About symbol versioning on page 9-18.](#)

Reference

armlink Reference Guide:

- [--symver_script on page 2-130.](#)
- [--symver_soname on page 2-131.](#)

9.13 Automatic dynamic symbol table rules in the BPABI DLL-like model

The following rules apply to the *Base Platform Application Binary Interface* (BPABI) DLL-like model:

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.

9.13.1 See also

Concepts

- [What is a steering file? on page 7-23](#)

Reference

- [Steering file command summary on page 7-25.](#)
- [Steering file format on page 7-26.](#)

armlink Reference Guide:

- [--edit on page 2-41.](#)
- [--export_all, --no_export_all on page 2-49.](#)
- [--export_dynamic, --no_export_dynamic on page 2-50.](#)
- [--keep_protected_symbols on page 2-72.](#)
- [EXPORT on page 3-2.](#)
- [IMPORT on page 3-4.](#)

9.14 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.

Selection of the required addressing mode is controlled by the following command-line options:

- `--pltgot`.
- `--pltgot_opts`.

Note

The DLL-like model is not supported for AArch64 state.

9.14.1 See also

Reference

armlink Reference Guide:

- [--pltgot on page 2-99](#).
- [--pltgot_opts on page 2-100](#).

9.15 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.

9.15.1 See also

Concepts

- [Linker options for bare metal and DLL-like models](#) on page 9-12.
- [Bare metal and DLL-like memory model](#) on page 9-13.
- [Mandatory symbol versioning in the BPABI DLL-like model](#) on page 9-14.
- [Automatic dynamic symbol table rules in the BPABI DLL-like model](#) on page 9-15.
- [Addressing modes in the BPABI DLL-like model](#) on page 9-16.
- [Related linker command-line options for the BPABI DLL-like model](#) on page 9-23.

ARM C and C++ Libraries and Floating-Point Support User Guide:

- [Initialization of the execution environment and execution of the application](#) on page 2-53.
- [C++ initialization, construction and destruction](#) on page 2-54.

9.16 About symbol versioning

Symbol versioning records extra information about symbols imported from, and exported by, a dynamic shared object. The dynamic loader uses this extra information to ensure that all the symbols required by an image are available at load time.

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.

9.16.1 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.

9.16.2 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.

9.16.3 Creating versioned symbols

By default, the linker does not create versioned symbols for a non *Base Platform Application Binary Interface* (BPABI) shared object.

9.16.4 See also

Reference

- [Symbol versioning script file](#) on page 9-19.
- fromelf User Guide:*
- [--symbolversions, --no_symbolversions](#) on page 4-70.

9.17 Symbol versioning script file

You can embed the commands to produce symbol versions in a script file that is specified by 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.

A steering file can be provided 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] ";"
```

The *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. Version names are not significant, but it helps to choose readable names, for example:

```
symbol_association ::=
```

```
    "local:" | "global:" | symbol_name ";"
```

where:

- "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.

———— 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.

9.17.1 See also

Concepts

- [About symbol versioning on page 9-18.](#)
- [Example of creating versioned symbols on page 9-20.](#)
- [Linker options for enabling implicit symbol versioning on page 9-22.](#)

Reference

armlink Reference Guide:

- [--symver_script on page 2-130.](#)

9.18 Example of creating versioned symbols

The following example places the symbols `foo@ver1`, `foo@@ver2`, and `bar@@ver1` into the object symbol table:

Example 9-1 Creating versioned symbols, embedded symbols

```
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, which includes the addition of dependency information so that `ver2` depends on `ver1` is:

Example 9-2 Creating versioned symbols script file

```
ver1
{
    global:
        foo; bar;
    local:
        *;
};

ver2
{
    global:
        foo;
} ver1;
```

9.18.1 See also

Concepts

- [About symbol versioning on page 9-18.](#)
- [Linker options for enabling implicit symbol versioning on page 9-22.](#)

Reference

armlink Reference Guide:

- [--symver_script on page 2-130.](#)

9.19 About 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:

Example 9-3 Creating versioned symbols, embedded symbols

```
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.

9.19.1 See also

Reference

armasm User Guide:

- [Chapter 7 Writing A32/T32 Assembly Language](#).

9.20 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 cannot be combined with embedded symbols or a script file.

9.20.1 See also

Reference

- [About symbol versioning on page 9-18.](#)
- [Symbol versioning script file on page 9-19.](#)
- [About embedded symbols on page 9-21.](#)

armlink Reference Guide:

- [--symver_soname on page 2-131.](#)

9.21 Related linker command-line options for the BPABI DLL-like model

The following linker command-line options relate to the *Base Platform Application Binary Interface* (BPABI) DLL-like model:

Note

The DLL-like model is not supported for AArch64 state.

Reference

armlink Reference Guide:

- [--base_platform](#) on page 2-12.
- [--bpabi](#) on page 2-16.
- [--dll](#) on page 2-38.
- [--export_all](#), [--no_export_all](#) on page 2-49.
- [--pltgot](#) on page 2-99.
- [--pltgot_opts](#) on page 2-100.
- [--ro_base](#) on page 2-106.
- [--rosplit](#) on page 2-107.
- [--rw_base](#) on page 2-108.
- [--symver_script](#) on page 2-130.
- [--symver_soname](#) on page 2-131.

Chapter 10

Features of the Base Platform linking model

The following topics describe features of the Base Platform linking model supported by the ARM linker, `arm-link`:

Concepts

- [Restrictions on the use of scatter files with the Base Platform model on page 10-2.](#)
- [Example scatter file for the Base Platform linking model on page 10-5.](#)
- [Placement of PLT sequences with the Base Platform model on page 10-7.](#)

Note

The Base Platform linking model is not supported for AArch64 state.

10.1 Restrictions on the use of scatter files with the Base Platform model

The Base Platform model supports scatter files. 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:

Example 10-1 Valid scatter file example using RELOC and *+offset*

```
# This is valid. All execution regions have +offset addresses.
LR1 0x8000 RELOC
{
    ER_RELATIVE +0
    {
        *(+R0)
    }
}
```

Example 10-2 Invalid scatter file example using RELOC and *+offset*

```
# This is not valid. One execution region has an absolute base address.
LR1 0x8000 RELOC
{
    ER_RELATIVE +0
    {
        *(+R0)
    }
    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:

Example 10-3 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
    {
        *(+R0)
    }
}
```

Example 10-4 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
    {
        *(+R0)
    }
    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:

Example 10-5 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
    {
        *(+R0)
    }
    ER2 +0
    {
        *(.plt_LR1)
    }
}
LR2 0x10000
{
    ER1 +0
    {
        *(other_code)
    }
}
```

Example 10-6 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
    {
        *(+R0)
    }
}
```

```

LR2 0x10000
{
    ER1 +0
    {
        *(.plt_LR1)
    }
}

```

10.1.1 See also

Concepts

- [Base Platform linking model](#) on page 3-6.
- [Placement of PLT sequences with the Base Platform model](#) on page 10-7.

Reference

armlink Reference Guide:

- [Load region attributes](#) on page 4-8.
- [Execution region attributes](#) on page 4-13.
- [Address attributes for load and execution regions](#) on page 4-16.
- [Inheritance rules for load region address attributes](#) on page 4-20.
- [Inheritance rules for the RELOC address attribute](#) on page 4-22.

10.2 Example scatter file for the Base Platform linking model

This example shows the use of a scatter file with the Base Platform linking model.

The standard *Base Platform Application Binary Interface* (BPABI) memory model in scatter file format, with relocatable load regions is:

Example 10-7 Standard BPABI scatter file with relocatable load regions

```

LR1 0x8000 RELOC
{
    ER_RO +0
    {
        *(+R0)
    }
}

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.

The next example shows two load regions LR1 and LR2 that are not relocatable.

Example 10-8 Scatter file with some load regions that are not relocatable

```

LR1 0x8000
{
    ER_RO +0
    {
        *(+R0)
    }
    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.

10.2.1 See also

Concepts

- [Base Platform Application Binary Interface \(BPABI\) linking model](#) on page 3-5.
- [Base Platform linking model](#) on page 3-6.
- [Restrictions on the use of scatter files with the Base Platform model](#) on page 10-2

Reference

armlink Reference Guide:

- [Load region attributes](#) on page 4-8.

10.3 Placement of PLT sequences with the Base Platform model

The linker supports *Procedure Linkage Table* (PLT) generation for multiple load regions containing code when in Base Platform mode (`--base_platform`).

To turn on PLT generation when in Base Platform mode use `--pltgot=option` that generates PLT sequences. You can use the option `--pltgot_opts=crosslr` to add entries in the PLT for calls between 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.

10.3.1 See also

Concepts

- [Base Platform linking model on page 3-6.](#)

Reference

armlink Reference Guide:

- [--base_platform on page 2-12.](#)
- [--pltgot on page 2-99.](#)
- [--pltgot_opts on page 2-100.](#)