Arm® Compiler

Version 6.12

armlink User Guide

arm

100070_0612_00_en

Arm® Compiler

armlink User Guide

Release Information

Document History

Issue	Date	Confidentiality	Change
A	14 March 2014	Non-Confidential	Arm Compiler v6.00 Release
B	15 December 2014	Non-Confidential	Arm Compiler v6.01 Release
C	30 June 2015	Non-Confidential	Arm Compiler v6.02 Release
D	18 November 2015	Non-Confidential	Arm Compiler v6.3 Release
E	24 February 2016	Non-Confidential	Arm Compiler v6.4 Release
F	29 June 2016	Non-Confidential	Arm Compiler v6.5 Release
G	04 November 2016	Non-Confidential	Arm Compiler v6.6 Release
0607-00	05 April 2017	Non-Confidential	Arm Compiler v6.7 Release. Document numbering scheme has changed.
0608-00	30 July 2017	Non-Confidential	Arm Compiler v6.8 Release.
0609-00	25 October 2017	Non-Confidential	Arm Compiler v6.9 Release.
0610-00	14 March 2018	Non-Confidential	Arm Compiler v6.10 Release.
0611-00	25 October 2018	Non-Confidential	Arm Compiler v6.11 Release.
0612-00	27 February 2019	Non-Confidential	Arm Compiler v6.12 Release.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

Confidentiality Status

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

Unrestricted Access is an Arm internal classification.

Product Status

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

Web Address

http://www.arm.com

Contents
Arm® Compiler armlink User Guide

List of Figures
Arm® Compiler armlink User Guide

List of Tables
Arm® Compiler armlink User Guide

Preface

This preface introduces the *Arm® Compiler armlink User Guide*.

It contains the following:

- *About this book* on page 13.

About this book

Arm® Compiler *armlink User Guide* provides user information for the Arm linker, `armlink`. It describes the basic linker functionality, image structure, BPABI support, how to access image symbols, and how to use scatter files.

Using this book

This book is organized into the following chapters:

Chapter 1 Overview of the Linker
Gives an overview of the Arm linker, `armlink`.

Chapter 2 Linking Models Supported by armlink
Describes the linking models supported by the Arm linker, `armlink`.

Chapter 3 Image Structure and Generation
Describes the image structure and the functionality available in the Arm linker, `armlink`, to generate images.

Chapter 4 Linker Optimization Features
Describes the optimization features available in the Arm linker, `armlink`.

Chapter 5 Getting Image Details
Describes how to get image details from the Arm linker, `armlink`.

Chapter 6 Accessing and Managing Symbols with armlink
Describes how to access and manage symbols with the Arm linker, `armlink`.

Chapter 7 Scatter-loading Features
Describes the scatter-loading features and how you use scatter files with the Arm linker, `armlink`, to create complex images.

Chapter 8 Scatter File Syntax
Describes the format of scatter files.

Chapter 9 BPABI Shared Libraries and Executables
Describes how the Arm linker, `armlink`, supports the *Base Platform Application Binary Interface* (BPABI) shared libraries and executables.

Chapter 10 Features of the Base Platform Linking Model
Describes features of the Base Platform linking model supported by the Arm linker, `armlink`.

Chapter 11 Linker Command-line Options
Describes the command-line options supported by the Arm linker, `armlink`.

Chapter 12 Linker Steering File Command Reference
Describes the steering file commands supported by the Arm linker, `armlink`.

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

Glossary

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

See the *Arm® Glossary* for more information.

Typographic conventions

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

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

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

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

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

`monospace bold`
Denotes language keywords when used outside example code.

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

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

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

Feedback

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:
- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

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

- The title *Arm Compiler armlink User Guide*.
- The number 100070_0612_00_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

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

Other information

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

Chapter 1
Overview of the Linker

Gives an overview of the Arm linker, `armlink`.

It contains the following sections:

1.1 About the linker

The linker combines the contents of one or more object files with selected parts of one or more object libraries to produce executable images, partially linked object files, or shared object files.

This section contains the following subsections:

1.1.1 Summary of the linker features

The linker has many features for linking input files to generate various types of output files.

The linker can:

- Link A32 and T32 code, or A64 code.
- Generate interworking veneers to switch between A32 and T32 states when required.
- Generate range extension 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.
- Position code and data at specific locations within the system memory map, using either a command-line option or a scatter file.
- Perform RW data compression to minimize ROM size.
- Eliminate unused sections 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.
- Build images suitable for all states of the Armv8-M Security Extensions.

Note

Be aware of the following:

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

Note

The command-line option descriptions and related information in the individual Arm Compiler tools documents describe all the features that Arm Compiler supports. Any features not documented are not supported and are used at your own risk. You are responsible for making sure that any generated code using *community features* on page 1-21 is operating correctly.

Related concepts

3.4 Linker support for creating demand-paged files on page 3-51

7.6 Placement of CMSE veneer sections for a Secure image on page 7-140

Related reference

Chapter 2 Linking Models Supported by armlink on page 2-25

Chapter 3 Image Structure and Generation on page 3-32

Chapter 4 Linker Optimization Features on page 4-65

Chapter 5 Getting Image Details on page 5-80

Chapter 6 Accessing and Managing Symbols with armlink on page 6-87

Chapter 7 Scatter-loading Features on page 7-104

Chapter 9 BPABI Shared Libraries and Executables on page 9-194

Chapter 10 Features of the Base Platform Linking Model on page 10-208

Related information

Base Platform ABI for the Arm Architecture

1.1.2 What the linker can accept as input

`armlink` can accept one or more object files from toolchains that support Arm ELF.

Object files must be formatted as Arm ELF. This format is described in:

- *ELF for the Arm® Architecture (IHI 0044)*.
- *ELF for the Arm® 64-bit Architecture (AArch64) (IHI 0056)*.

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.
- A Secure code import library when building a Non-secure image that needs to call a Secure image.
- A Secure code import library when building a Secure image that has to use the entry addresses in a previously generated import library.

Related concepts

6.5 Access symbols in another image on page 6-97

Related reference

Chapter 7 Scatter-loading Features on page 7-104

Chapter 12 Linker Steering File Command Reference on page 12-394

Chapter 8 Scatter File Syntax on page 8-164

11.57 --import_cmse_lib_in=filename on page 11-279

Related information

About the Arm librarian

Building Secure and Non-secure Images Using Armv8-M Security Extensions

ELF for the Arm Architecture (IHI 0044)

ELF for the Arm 64-bit Architecture (AArch64) (IHI 0056)

1.1.3 What the linker outputs

`armlink` can create executable images and object files.

Output from `armlink` can be:

- An ELF executable image.
- A partially linked ELF object that can be used as input in a subsequent link step.
- A Secure code import library that is required by developers building a Non-secure image that needs to call a Secure image.

Note

You can also 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.

Related concepts

2.3 Partial linking model on page 2-28

3.3 Section placement with the linker on page 3-48

3.1 The structure of an Arm® ELF image on page 3-33

Related reference

11.58 --import_cmse_lib_out=filename on page 11-280

Related information

Building Secure and Non-secure Images Using Armv8-M Security Extensions

Overview of the fromelf image converter

1.2 Linker command-line syntax

The `armlink` command can accept many input files together with options that determine how to process the files.

The command for invoking the linker is:

`armlink` *`options input-file-list`*

where:

`options`

Linker command-line options.

`input-file-list`

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

Related reference

11.65 input-file-list on page 11-289
Chapter 11 Linker Command-line Options on page 11-214

1.3 What the linker does when constructing an executable image

`armlink` performs many operations, depending on the content of the input files and the command-line options you specify.

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.
- Removes unused sections.
- Eliminates duplicate common section groups.
- Sorts input sections according to their attributes and names, and merges sections with similar attributes and names into contiguous chunks.
- Organizes object fragments into memory regions according to the grouping and placement information provided.
- Assigns addresses to relocatable values.
- Generates an executable image.

Related concepts

4.2 Elimination of unused sections on page 4-67
3.1 The structure of an Arm® ELF image on page 3-33

1.4 Support level definitions

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

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

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

Identification in the documentation

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

Product features

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

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

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

Beta product features

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

Beta product features are indicated with [BETA].

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

Alpha product features

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

Alpha product features are indicated with [ALPHA].

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

Community features

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

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

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

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

Guidance on use of community features

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

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

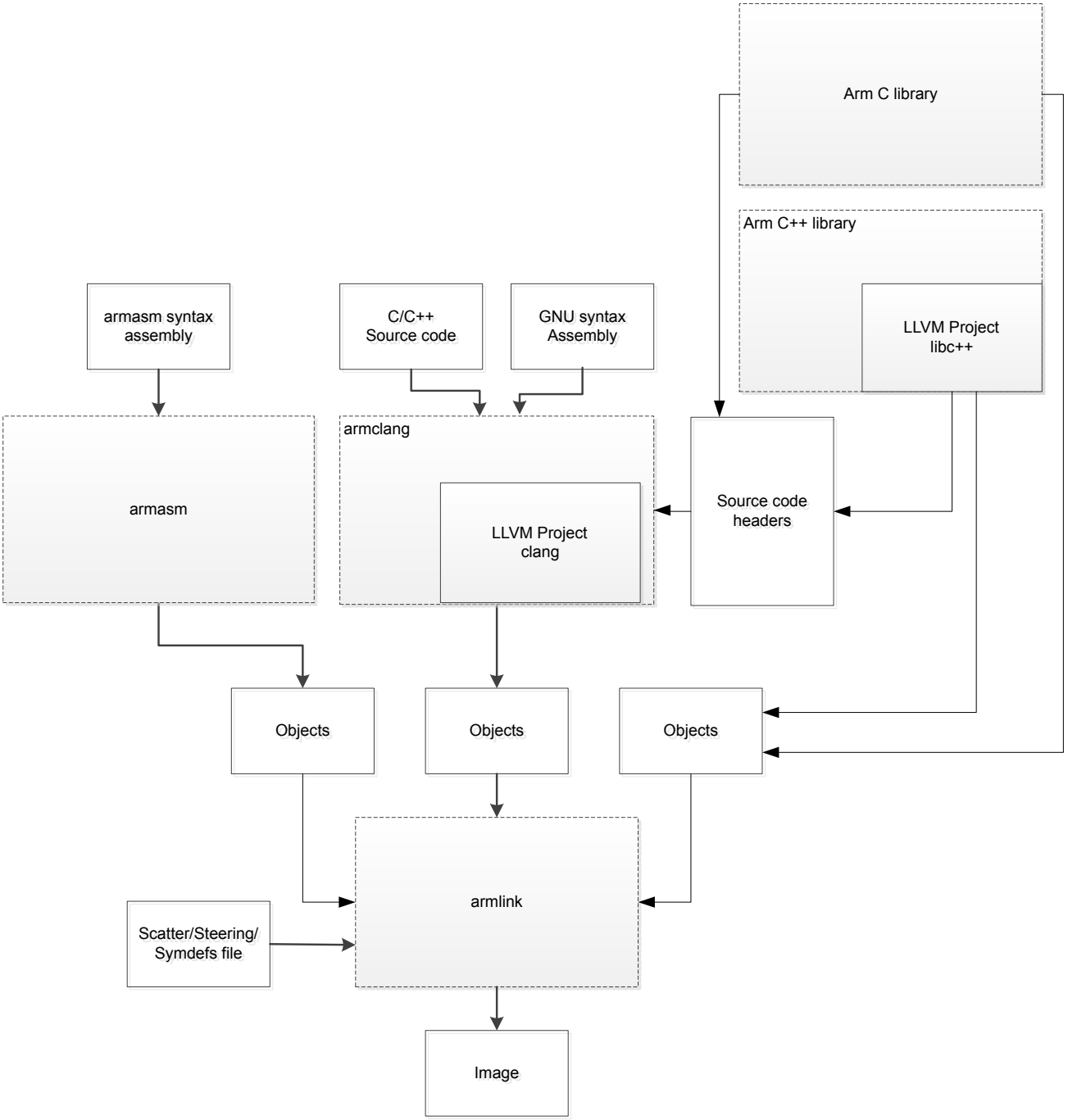

Figure 1-1 Integration boundaries in Arm Compiler 6.

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

- Features primarily used when targeting hosted environments such as Linux or BSD might have significant limitations, or might not be applicable, when targeting bare-metal environments.
- The Clang implementations of compiler features, particularly those that have been present for a long time in other toolchains, are likely to be mature. The functionality of new features, such as support

for new language features, is likely to be less mature and therefore more likely to have limited functionality.

Deprecated features

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

For information on replacing deprecated features with supported features, refer to the Arm Compiler documentation and Release Notes.

Unsupported features

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

Limitations of product features are stated in the documentation. Arm cannot provide an exhaustive list of unsupported features or use-cases for community features. The known limitations on community features are listed in *Community features* on page 1-21.

List of known unsupported features

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

- The Clang option `-stdlib=libstdc++` is not supported.
- C++ static initialization of local variables is not thread-safe when linked against the standard C++ libraries. For thread-safety, you must provide your own implementation of thread-safe functions as described in *Standard C++ library implementation definition*.

  ——— **Note** ———

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

  ———————————

- Use of C11 library features is unsupported.
- Any community feature that is exclusively related to non-Arm architectures is not supported.
- Compilation for targets that implement architectures older than Armv7 or Armv6-M is not supported.
- The **`long double`** data type is not supported for AArch64 state because of limitations in the current Arm C library.
- Complex numbers are not supported because of limitations in the current Arm C library.

Chapter 2
Linking Models Supported by armlink

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

It contains the following sections:

2.1 Overview of linking models

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

The linking models supported by `armlink` are:

Bare-metal
This model does not target any specific platform. It enables you to create an image with your own custom operating system, memory map, and, application code if required. Some limited dynamic linking support is available. You can specify additional options depending on whether or not a scatter file is in use.

Bare-metal Position Independent Executables (PIE)
This model produces a bare-metal Position Independent Executable (PIE). This is an executable that does not need to be executed at a specific address but can be executed at any suitably aligned address. All objects and libraries linked into the image must be compiled to be position independent.

——— **Note** ———

The Bare-metal PIE feature is deprecated.

———————————

Partial linking
This model produces a relocatable ELF object suitable for input to the linker in a subsequent link step. The partial object can be used as input to another link step. The linker performs limited processing of input objects to produce a single output object.

BPABI
This model supports the DLL-like *Base Platform Application Binary Interface* (BPABI). It is intended to produce applications and DLLs that can run on a platform OS that varies in complexity. The memory model is restricted according to the *Base Platform ABI for the Arm® Architecture* (IHI 0037 C).

——— **Note** ———

Not supported for AArch64 state.

———————————

Base Platform
This is an extension to the BPABI model to support scatter-loading.

——— **Note** ———

Not supported for AArch64 state.

———————————

You can combine related options in each model to tighten control over the output.

Related concepts

2.2 Bare-metal linking model on page 2-27
2.3 Partial linking model on page 2-28
2.4 Base Platform Application Binary Interface (BPABI) linking model on page 2-29
2.5 Base Platform linking model on page 2-30

Related reference

Chapter 9 BPABI Shared Libraries and Executables on page 9-194

Related information

Base Platform ABI for the Arm Architecture

2.2 Bare-metal linking model

Focuses on the conventional embedded market where the whole program, possibly including a *Real-Time Operating System* (RTOS), is linked in one pass.

The linker can make very few assumptions about the memory map of a bare-metal system. Therefore, you must use the scatter-loading mechanism if you want more precise control. Scatter-loading allows different regions in an image memory map to be placed at addresses other than at their natural address. Such an image is a relocatable image, and the linker must adjust program addresses and resolve references to external symbols.

By default, the linker attempts to resolve all the relocations statically. However, it is also possible to create a position-independent or relocatable image. Such an image can be executed from different addresses and have its relocations resolved at load or run-time. You can use a dynamic model to create relocatable images. A position-independent image does not require a dynamic model.

With the bare-metal model, you can:

- Identify the regions that can be relocated or are position-independent using a scatter file or command-line options.
- Identify the symbols that can be imported and exported using a steering file.

You can use `--scatter=`*`file`* with this model.

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

- `--reloc` (not supported for AArch64 state).
- `--ro_base=`*`address`*.
- `--ropi`.
- `--rosplit`.
- `--rw_base=`*`address`*.
- `--rwpi`.
- `--split`.
- `--xo_base=`*`address`*.
- `--zi_base`.

Note

`--xo_base` cannot be used with `--ropi` or `--rwpi`.

Related concepts

3.1.4 Methods of specifying an image memory map with the linker on page 3-37

2.4 Base Platform Application Binary Interface (BPABI) linking model on page 2-29

10.2 Scatter files for the Base Platform linking model on page 10-211

Related reference

11.157 --xo_base=address on page 11-389

11.36 --edit=file_list on page 11-258

11.111 --reloc on page 11-341

11.114 --ro_base=address on page 11-344

11.115 --ropi on page 11-345

11.116 --rosplit on page 11-346

11.117 --rw_base=address on page 11-347

11.118 --rwpi on page 11-348

11.120 --scatter=filename on page 11-350

11.127 --split on page 11-359

11.161 --zi_base=address on page 11-393

Chapter 12 Linker Steering File Command Reference on page 12-394

2.3 Partial linking model

Produces a single output file that can be used as input to a subsequent link step.

Partial linking:

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

If the linker finds multiple entry points in the input files it generates an error because the single output file can have only one entry point.

To link with this model, use the `--partial` command-line option.

Note

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

Related concepts

6.6 Edit the symbol tables with a steering file on page 6-100

Related reference

6.6.3 Steering file format on page 6-101

Chapter 12 Linker Steering File Command Reference on page 12-394

11.36 --edit=file_list on page 11-258

11.100 --partial on page 11-329

2.4 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.

The BPABI model produces as much dynamic information as possible without focusing on any specific platform.

Note

BPABI is not supported for AArch64 state.

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

- `--dll`.
- `--force_so_throw`, `--no_force_so_throw`.
- `--pltgot=`*`type`*.
- `--ro_base=`*`address`*.
- `--rosplit`.
- `--rw_base=`*`address`*.
- `--rwpi`.

Be aware of the following:

- You cannot use scatter-loading. However, the Base Platform linking model supports scatter-loading.
- The model by default assumes that shared objects cannot throw a C++ exception (`--no_force_so_throw`).
- The default value of the `--pltgot` option is `direct`.
- You must use symbol versioning to ensure that all the required symbols are available at load time.

Related concepts

2.2 Bare-metal linking model on page 2-27
9.5 Symbol versioning on page 9-205

Related reference

11.11 --bpabi on page 11-230
11.32 --dll on page 11-254
11.50 --force_so_throw, --no_force_so_throw on page 11-272
11.104 --pltgot=type on page 11-334
11.114 --ro_base=address on page 11-344
11.116 --rosplit on page 11-346
11.117 --rw_base=address on page 11-347
11.118 --rwpi on page 11-348

Related information

Base Platform ABI for the Arm Architecture

2.5 Base Platform linking model

Enables you to create dynamically linkable images that do not have the memory map enforced by the *Base Platform Application Binary Interface* (BPABI) linking model.

The Base Platform linking model enables you to:

- Create images with a memory map described in a scatter file.
- Have dynamic relocations so the images can be dynamically linked. The dynamic relocations can also target within the same image.

Note

Base Platform is not supported for AArch64 state.

Note

The BPABI specification places constraints on the memory model that can be violated using scatter-loading. However, because Base Platform is a superset of BPABI, it is possible to create a BPABI conformant image with Base Platform.

To link with the Base Platform model, use the `--base_platform` command-line option.

If you specify this option, the linker acts as if you specified `--bpabi`, with the following exceptions:

- Scatter-loading is available with `--scatter`. If you do not specify `--scatter`, then the standard BPABI memory model scatter file is used.
- The following options are available:
  - `--dll`.
  - `--force_so_throw`, `--no_force_so_throw`.
  - `--pltgot=`*type*.
  - `--rosplit`.
- The default value of the `--pltgot` option is different to that for `--bpabi`:
  - For `--base_platform`, the default is `--pltgot=none`.
  - For `--bpabi` the default is `--pltgot=direct`.
- Each load region containing code might require a *Procedure Linkage Table* (PLT) section to indirect calls from the load region to functions where the address is not known at static link time. The PLT section for a load region `LR` must be placed in `LR` and be accessible at all times to code within `LR`.

  If you do not use a scatter file, the linker can ensure that the PLT section is placed correctly, and contains entries for calls only to imported symbols. If you specify a scatter file, the linker might not be able to find a suitable location to place the PLT.

  To ensure calls between relocated load regions use a PLT entry:
  - Use the `--pltgot=direct` option to turn on PLT generation.
  - Use the `--pltgot_opts=crosslr` option to add entries in the PLT for calls from and to `RELOC` load regions. The linker generates a PLT for each load region so that calls do not have to be extended to reach a distant PLT.

Be aware of the following:

- The model by default assumes that shared objects cannot throw a C++ exception (`--no_force_so_throw`).
- You must use symbol versioning to ensure that all the required symbols are available at load time.
- There are restrictions on the type of scatter files you can use.

Related concepts

10.1 Restrictions on the use of scatter files with the Base Platform model on page 10-209

10.2 Scatter files for the Base Platform linking model on page 10-211

2.4 Base Platform Application Binary Interface (BPABI) linking model on page 2-29
3.1.4 Methods of specifying an image memory map with the linker on page 3-37
9.5 Symbol versioning on page 9-205

Related reference

11.7 --base_platform on page 11-225
11.32 --dll on page 11-254
11.105 --pltgot_opts=mode on page 11-335
11.116 --rosplit on page 11-346
11.120 --scatter=filename on page 11-350
11.104 --pltgot=type on page 11-334

Chapter 3
Image Structure and Generation

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

It contains the following sections:

3.1 The structure of an Arm® ELF image

An Arm ELF image contains sections, regions, and segments, and each link stage has a different view of the image.

The structure of an image is defined by the:

- Number of its constituent regions and output sections.
- Positions in memory of these regions and sections when the image is loaded.
- Positions in memory of these regions and sections when the image executes.

This section contains the following subsections:

3.1.1 Views of the image at each link stage

Each link stage has a different view of the image.

The image views are:

ELF object file view (linker input)

The ELF object file view comprises input sections. The ELF object file can be:

- A relocatable file that holds code and data suitable for linking with other object files to create an executable or a shared object file.
- A shared object file that holds code and data.

Linker view

The linker has two views for the address space of a program that become distinct in the presence of overlaid, position-independent, and relocatable program fragments (code or data):

- The load address of a program fragment is the target address that the linker expects an external agent such as a program loader, dynamic linker, or debugger to copy the fragment from the ELF file. This might not be the address at which the fragment executes.
- The execution address of a program fragment is the target address where the linker expects the fragment to reside whenever it participates in the execution of the program.

If a fragment is position-independent or relocatable, its execution address can vary during execution.

ELF image file view (linker output)

The ELF image file view comprises program segments and output sections:

- A load region corresponds to a program segment.
- An execution region contains one or more of the following output sections:
  - RO section.
  - RW section.
  - XO section.
  - ZI section.

One or more execution regions make up a load region.

Note

With `armlink`, the maximum size of a program segment is 2GB.

When describing a memory view:

- The term *root region* means a region that has the same load and execution addresses.
- Load regions are equivalent to ELF segments.

The following figure shows the relationship between the views at each link stage:

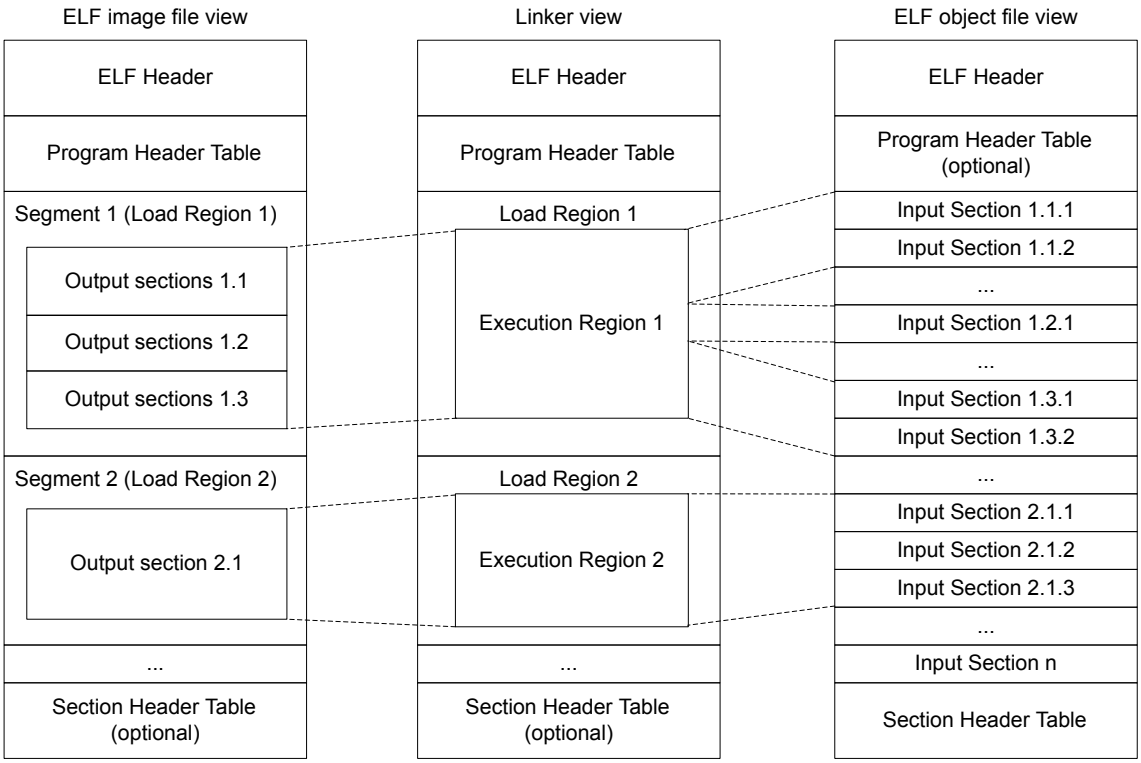

Figure 3-1 Relationship between sections, regions, and segments

3.1.2 Input sections, output sections, regions, and program segments

An object or image file is constructed from a hierarchy of input sections, output sections, regions, and program segments.

Input section

An input section is an individual section from an input object file. It contains code, initialized data, or describes a fragment of memory that is not initialized or that must be set to zero before the image can execute. These properties are represented by attributes such as RO, RW, XO, and ZI. These attributes are used by `armlink` to group input sections into bigger building blocks called output sections and regions.

Output section

An output section is a group of input sections that have the same RO, RW, XO, or ZI attribute, and that are placed contiguously in memory by the linker. An output section has the same attributes as its constituent input sections. Within an output section, the input sections are sorted according to the section placement rules.

Region

A region contains up to three output sections depending on the contents and the number of sections with different attributes. By default, the output sections in a region are sorted according to their attributes:

- If no XO output sections are present, then the RO output section is placed first, followed by the RW output section, and finally the ZI output section.
- If all code in the execution region is execute-only, then an XO output section is placed first, followed by the RW output section, and finally the ZI output section.

A region typically maps onto a physical memory device, such as ROM, RAM, or peripheral. You can change the order of output sections using scatter-loading.

Program segment

A program segment corresponds to a load region and contains execution regions. Program segments hold information such as text and data.

Note

With `armlink`, the maximum size of a program segment is 2GB.

Note

XO memory is supported only for Armv7-M and Armv8-M architectures.

Considerations when execute-only sections are present

Be aware of the following when *execute-only* (XO) sections are present:

- You can mix XO and non-XO sections in the same execution region. In this case, the XO section loses its XO property and results in the output of a RO section.
- If an input file has one or more XO sections then the linker generates a separate XO execution region if the XO and RO sections are in distinct regions. In the final image, the XO execution region immediately precedes the RO execution region, unless otherwise specified by a scatter file or the `--xo_base` option.

  The linker automatically fabricates a separate ER_XO execution region for XO sections when all the following are true:
  - You do not specify the `--xo_base` option or a scatter file.
  - The input files contain at least one XO section.

Related concepts

3.1.1 Views of the image at each link stage on page 3-33
3.1.4 Methods of specifying an image memory map with the linker on page 3-37
3.3 Section placement with the linker on page 3-48

3.1.3 Load view and execution view of an image

Image regions are placed in the system memory map at load time. The location of the regions in memory might change during execution.

Before you can execute the image, you might have to move some of its regions to their execution addresses and create the ZI output sections. For example, initialized RW data might have to be copied from its load address in ROM to its execution address in RAM.

The memory map of an image has the following distinct views:

Load view

Describes each image region and section in terms of the address where it is located when the image is loaded into memory, that is, the location before image execution starts.

Execution view

Describes each image region and section in terms of the address where it is located during image execution.

The following figure shows these views for an image without an *execute-only* (XO) section:

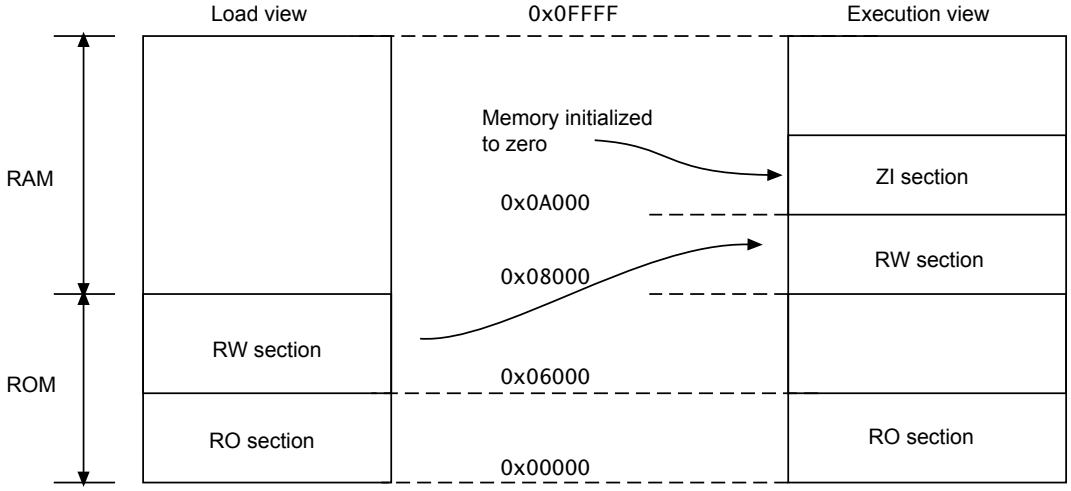

Figure 3-2 Load and execution memory maps for an image without an XO section

The following figure shows load and execution views for an image with an XO section:

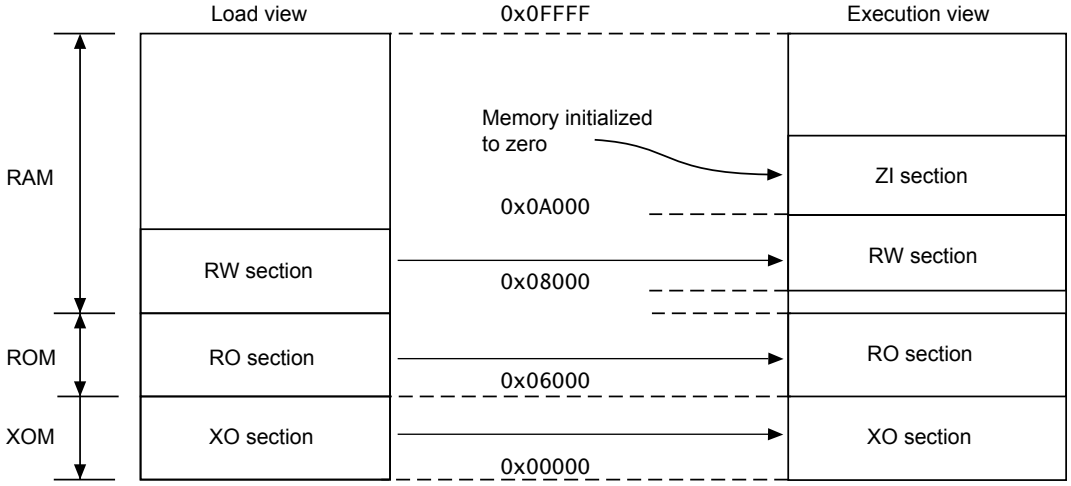

Figure 3-3 Load and execution memory maps for an image with an XO section

Note

XO memory is supported only for Armv7-M and Armv8-M architectures.

The following table compares the load and execution views:

Table 3-1 Comparing load and execution views

Load	Description	Execution	Description
Load address	The address where a section or region is loaded into memory before the image containing it starts executing. The load address of a section or a non-root region can differ from its execution address.	Execution address	The address where a section or region is located while the image containing it is being executed.
Load region	A load region describes the layout of a contiguous chunk of memory in load address space.	Execution region	An execution region describes the layout of a contiguous chunk of memory in execution address space.

Related concepts

3.1.1 Views of the image at each link stage on page 3-33

3.1.4 Methods of specifying an image memory map with the linker on page 3-37

3.3 Section placement with the linker on page 3-48

3.1.2 Input sections, output sections, regions, and program segments on page 3-34

3.1.4 Methods of specifying an image memory map with the linker

An image can consist of any number of regions and output sections. Regions can have different load and execution addresses.

When constructing the memory map of an image, `armlink` must have information about:

- How input sections are grouped into output sections and regions.
- Where regions are to be located in the memory map.

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

Command-line options for simple memory map descriptions

You can use the following options for simple cases where an image has only one or two load regions and up to three execution regions:

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

These options provide a simplified notation that gives the same settings as a scatter-loading description for a simple image. However, no limit checking for regions is available when using these options.

Scatter file for complex memory map descriptions

A scatter file is a textual description of the memory layout and code and data placement. It is used for more complex cases where you require complete control over the grouping and placement of image components. To use a scatter file, specify `--scatter=`*`filename`* at the command-line.

Note

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

Table 3-2 Comparison of scatter file and equivalent command-line options

Scatter file	Equivalent command-line options
`LR1 0x0000 0x20000` `{`	
`  ER_RO 0x0 0x2000` `  {`	`--ro_base=0x0`

`    init.o (INIT, +FIRST)`
`    *(+RO)`
`  }`
	`--first=init.o(init)`

Table 3-2 Comparison of scatter file and equivalent command-line options (continued)

Scatter file	Equivalent command-line options
`ER_RW 0x400000 { *(+RW) }`	`--rw_base=0x400000`
`ER_ZI 0x405000 { *(+ZI) } }`	`--zi_base=0x405000`
`LR_XO 0x8000 0x4000 {`	
`ER_XO 0x8000 { *(XO) } }`	`--xo_base=0x8000`

Note

If XO sections are present, a separate load and execution region is created only when you specify `--xo_base`. If you do not specify `--xo_base`, then the ER_XO region is placed in the LR1 region at the address specified by `--ro_base`. The ER_RO region is then placed immediately after the ER_XO region.

Related concepts

3.1.3 Load view and execution view of an image on page 3-35
3.2 Simple images on page 3-41
3.1 The structure of an Arm® ELF image on page 3-33
3.1.2 Input sections, output sections, regions, and program segments on page 3-34

Related reference

11.48 --first=section_id on page 11-270
11.69 --last=section_id on page 11-295
11.114 --ro_base=address on page 11-344
11.115 --ropi on page 11-345
11.116 --rosplit on page 11-346
11.117 --rw_base=address on page 11-347
11.118 --rwpi on page 11-348
11.120 --scatter=filename on page 11-350
11.127 --split on page 11-359
11.157 --xo_base=address on page 11-389
11.161 --zi_base=address on page 11-393

3.1.5 Image entry points

An entry point in an image is the location that is loaded into the PC. It is the location where program execution starts. Although there can be more than one entry point in an image, you can specify only one when linking.

Not every ELF file has to have an entry point. Multiple entry points in a single ELF file are not permitted.

Note

For embedded programs targeted at a Cortex®-M-based processor, the program starts at whatever location is loaded into the PC from the Reset vector. Typically, the Reset vector points to the CMSIS `Reset_Handler` function.

Types of entry point

There are two distinct types of entry point:

Initial entry point

The *initial* entry point for an image is a single value that is stored in the ELF header file. For programs loaded into RAM by an operating system or boot loader, the loader starts the image execution by transferring control to the initial entry point in the image.

An image can have only one initial entry point. The initial entry point can be, but is not required to be, one of the entry points set by the `ENTRY` directive.

Entry points set by the `ENTRY` directive

You can select one of many possible entry points for an image. An image can have only one entry point.

You create entry points in objects with the `ENTRY` directive in an assembler file. In embedded systems, typical use of this directive is to mark code that is entered through the processor exception vectors, such as RESET, IRQ, and FIQ.

The directive marks the output code section with an `ENTRY` keyword that instructs the linker not to remove the section when it performs unused section elimination.

For C and C++ programs, the `__main()` function in the C library is also an entry point.

If an embedded image is to be used by a loader, it must have a single initial entry point specified in the header. Use the `--entry` command-line option to select the entry point.

The initial entry point for an image

There can be only one initial entry point for an image, otherwise linker warning `L6305W` is output.

The initial entry point must meet the following conditions:

- The image entry point must always lie within an execution region.
- The execution region must not overlay another execution region, and must be a root execution region. That is, where the load address is the same as the execution address.

If you do not use the `--entry` option to specify the initial entry point, then:

- If the input objects contain only one entry point set by the `ENTRY` directive, the linker uses that entry point as the initial entry point for the image.
- The linker generates an image that does not contain an initial entry point when either:
  - More than one entry point is specified using the `ENTRY` directive.
  - No entry point is specified using the `ENTRY` directive.

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

Note

High vectors are not supported in AArch64 state.

Note

Some processors, such as Cortex-M7, can boot from a different address in some configurations.

Related concepts

7.2 Root region and the initial entry point on page 7-111

Related reference

11.41 --entry=location on page 11-263

Related information

ENTRY

List of the armlink error and warning messages

3.1.6 Restrictions on image structure

When an instruction accesses a memory address on an AArch64 target, the data must be within 4GB of the program counter.

For example, consider the following scatter file:

```
LOAD_REGION 0x0000000000 0x200000
{
    ROOT_REGION +0
    {
        *(Init, +FIRST)
        * (+RO)
        * (+RW, +ZI)
    }
    STACKHEAP 0x1FFFF0 EMPTY -0x18000
    {
    }
}

LOAD_REGION2 0x4000000000 0x200000
{
    ROOT_REGION2 +0
    {
        *(high_mem)
    }
}
```

`LOAD_REGION2` is 16GB away from `LOAD_REGION`, so data in `high_mem` is not accessible from code in `LOAD_REGION`. This results in a relocation out of range error at link time.

3.2 Simple images

A simple image consists of a number of input sections of type RO, RW, XO, and ZI. The linker collates the input sections to form the RO, RW, XO, and ZI output sections.

This section contains the following subsections:

3.2.1 Types of simple image

The types of simple image the linker can create depends on how the output sections are arranged within load and execution regions.

The types are:

Type 1

One region in load view, four contiguous regions in execution view. Use the `--ro_base` option to create this type of image.

Any XO sections are placed in an ER_XO region at the address specified by `--ro_base`, with the ER_RO region immediately following the ER_XO region.

Type 2

One region in load view, four non-contiguous regions in execution view. Use the `--ro_base` and `--rw_base` options to create this type of image.

Type 3

Two regions in load view, four non-contiguous regions in execution view. Use the `--ro_base`, `--rw_base`, and `--split` options to create this type of image.

For all the simple image types when `--xo_base` is not specified:

- If any XO sections are present, the first execution region contains the XO output section. The address specified by `--ro_base` is used as the base address of this output section.
- The second execution region contains the RO output section. This output section immediately follows an XO output.
- The third execution region contains the RW output section, if present.
- The fourth execution region contains the ZI output section, if present.

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

When you specify `--xo_base`, then XO sections are placed in a separate load and execution region.

However, you can also use the `--rosplit` option for a Type 3 image. This option splits the default load region into two RO output sections, one for code and one for data.

You can also use the `--zi_base` command-line option to specify the base address of a ZI execution region for Type 1 and Type 2 images. This option is ignored if you also use the `--split` command-line option that is required for Type 3 images.

You can also create simple images with scatter files.

Related concepts

7.13 Equivalent scatter-loading descriptions for simple images on page 7-152

3.2.2 Type 1 image structure, one load region and contiguous execution regions on page 3-42

3.2.3 Type 2 image structure, one load region and non-contiguous execution regions on page 3-43

3.2.4 Type 3 image structure, multiple load regions and non-contiguous execution regions on page 3-45

Related reference

11.114 --ro_base=address on page 11-344
11.116 --rosplit on page 11-346
11.117 --rw_base=address on page 11-347
11.120 --scatter=filename on page 11-350
11.127 --split on page 11-359
11.157 --xo_base=address on page 11-389
11.161 --zi_base=address on page 11-393

3.2.2 Type 1 image structure, one load region and contiguous execution regions

A Type 1 image consists of a single load region in the load view and three default execution regions, ER_RO, ER_RW, ER_ZI. These are placed contiguously in the memory map. An additional ER_XO execution region is created only if any input section is execute-only.

This approach is suitable for systems that load programs into RAM, for example, an OS bootloader or a desktop system. The following figure shows the load and execution view for a Type 1 image without *execute-only* (XO) code:

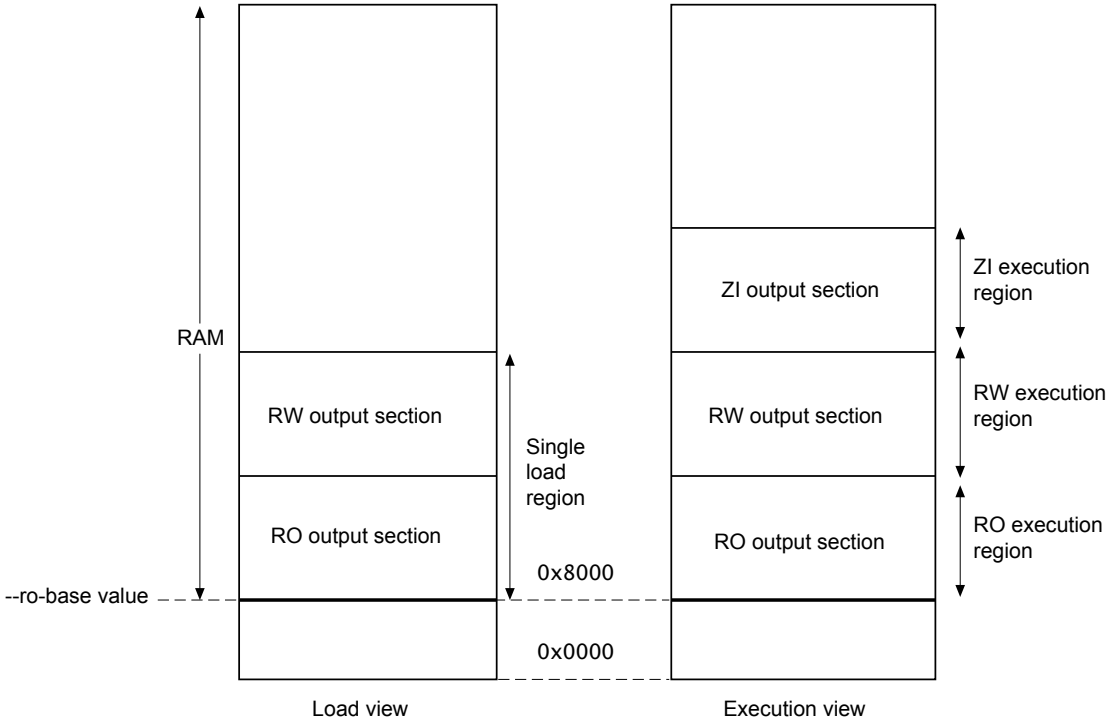

Figure 3-4 Simple Type 1 image

Use the following command for images of this type:

```
armlink --cpu=8-A.32 --ro_base 0x8000
```

Note

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

Load view

The single load region consists of the RO and RW output sections, placed consecutively. The RO and RW execution regions are both root regions. The ZI output section does not exist at load time. It is created before execution, using the output section description in the image file.

Execution view

The three execution regions containing the RO, RW, and ZI output sections are arranged contiguously. The execution addresses of the RO and RW regions are the same as their load addresses, so nothing has to be moved from its load address to its execution address. However, the ZI execution region that contains the ZI output section is created at run-time.

Use `armlink` option `--ro_base=address` to specify the load and execution address of the region containing the RO output. The default address is `0x8000`.

Use the `--zi_base` command-line option to specify the base address of a ZI execution region.

Load view for images containing execute-only regions

For images that contain XO sections, the XO output section is placed at the address that is specified by `--ro_base`. The RO and RW output sections are placed consecutively and immediately after the XO section.

Execution view for images containing execute-only regions

For images that contain XO sections, the XO execution region is placed at the address that is specified by `--ro_base`. The RO, RW, and ZI execution regions are placed contiguously and immediately after the XO execution region.

Note

XO memory is supported only for Armv7-M and Armv8-M architectures.

Related concepts

3.1 The structure of an Arm® ELF image on page 3-33

3.1.2 Input sections, output sections, regions, and program segments on page 3-34

3.1.3 Load view and execution view of an image on page 3-35

Related reference

11.114 --ro_base=address on page 11-344

11.157 --xo_base=address on page 11-389

11.161 --zi_base=address on page 11-393

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

A Type 2 image consists of a single load region, and three execution regions in execution view. The RW execution region is not contiguous with the RO execution region.

This approach is used, for example, for ROM-based embedded systems, where RW data is copied from ROM to RAM at startup. The following figure shows the load and execution view for a Type 2 image without *execute-only* (XO) code:

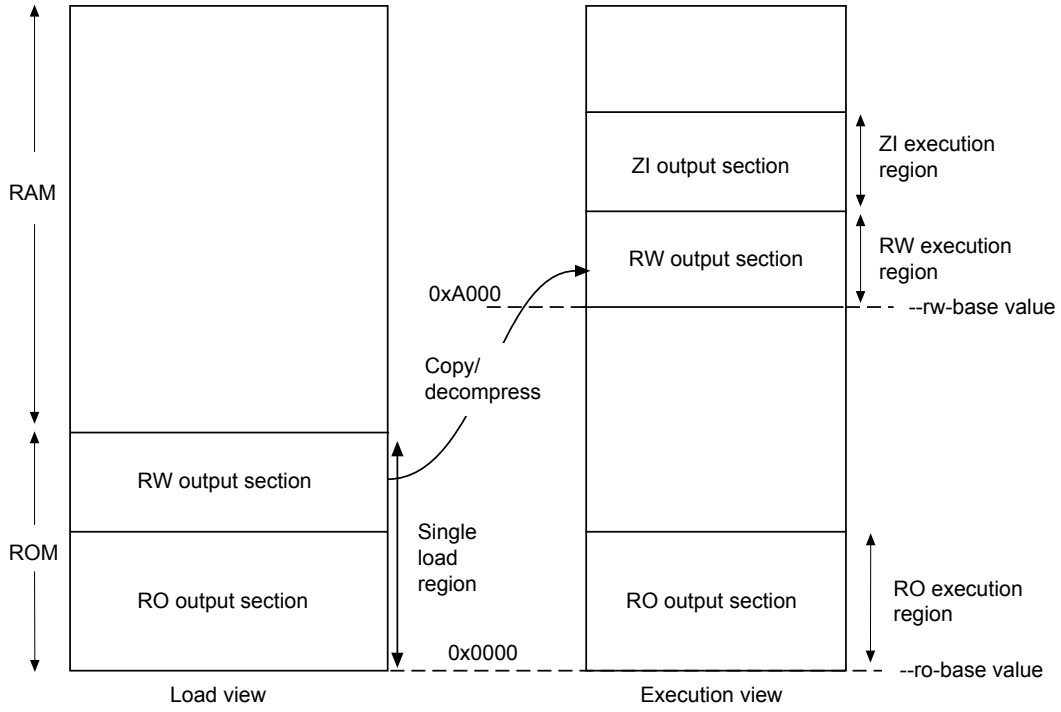

Figure 3-5 Simple Type 2 image

Use the following command for images of this type:

```
armlink --cpu=8-A.32 --ro_base=0x0 --rw_base=0xA000
```

Load view

In the load view, the single load region consists of the RO and RW output sections placed consecutively, for example, in ROM. Here, the RO region is a root region, and the RW region is non-root. The ZI output section does not exist at load time. It is created at runtime.

Execution view

In the execution view, the first execution region contains the RO output section and the second execution region contains the RW and ZI output sections.

The execution address of the region containing the RO output section is the same as its load address, so the RO output section does not have to be moved. That is, it is a root region.

The execution address of the region containing the RW output section is different from its load address, so the RW output section is moved from its load address (from the single load region) to its execution address (into the second execution region). The ZI execution region, and its output section, is placed contiguously with the RW execution region.

Use `armlink` options `--ro_base=`*`address`* to specify the load and execution address for the RO output section, and `--rw_base=`*`address`* to specify the execution address of the RW output section. If you do not use the `--ro_base` option to specify the address, the default value of `0x8000` is used by `armlink`. For an embedded system, `0x0` is typical for the `--ro_base` value. If you do not use the `--rw_base` option to specify the address, the default is to place RW directly above RO (as in a Type 1 image).

Use the `--zi_base` command-line option to specify the base address of a ZI execution region.

Note

The execution region for the RW and ZI output sections cannot overlap any of the load regions.

Load view for images containing execute-only regions

For images that contain XO sections, the XO output section is placed at the address specified by `--ro_base`. The RO and RW output sections are placed consecutively and immediately after the XO section.

Execution view for images containing execute-only regions

For images that contain XO sections, the XO execution region is placed at the address specified by `--ro_base`. The RO execution region is placed contiguously and immediately after the XO execution region.

If you use `--xo_base` *`address`*, then the XO execution region is placed in a separate load region at the specified address.

Note

XO memory is supported only for Armv7-M and Armv8-M architectures.

Related concepts

3.1 The structure of an Arm® ELF image on page 3-33

3.1.2 Input sections, output sections, regions, and program segments on page 3-34

3.1.3 Load view and execution view of an image on page 3-35

3.2.2 Type 1 image structure, one load region and contiguous execution regions on page 3-42

Related reference

11.114 --ro_base=address on page 11-344

11.117 --rw_base=address on page 11-347

11.157 --xo_base=address on page 11-389

11.161 --zi_base=address on page 11-393

3.2.4 Type 3 image structure, multiple load regions and non-contiguous execution regions

A Type 3 image is similar to a Type 2 image except that the single load region is split into multiple root load regions.

The following figure shows the load and execution view for a Type 3 image without *execute-only* (XO) code:

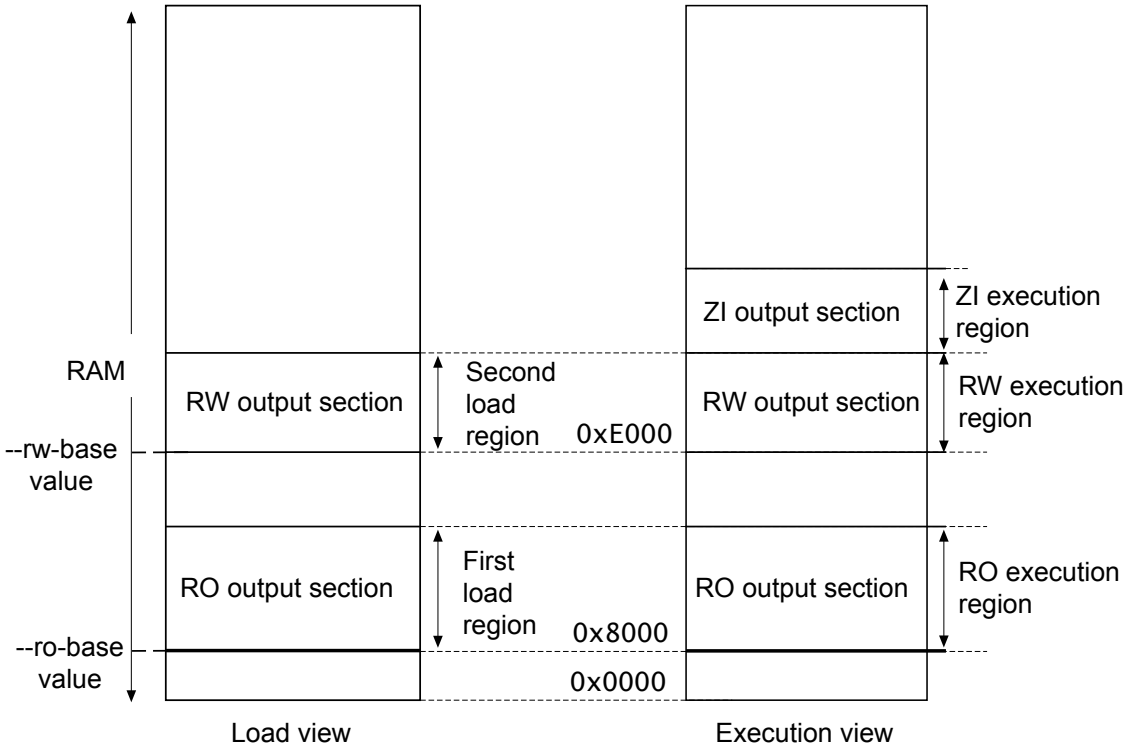

Figure 3-6 Simple Type 3 image

Use the following command for images of this type:

```
armlink --cpu=8-A.32 --split --ro_base 0x8000 --rw_base 0xE000
```

Load view

In the load view, the first load region consists of the RO output section, and the second load region consists of the RW output section. The ZI output section does not exist at load time. It is created before execution, using the description of the output section contained in the image file.

Execution view

In the execution view, the first execution region contains the RO output section, the second execution region contains the RW output section, and the third execution region contains the ZI output section.

The execution address of the RO region is the same as its load address, so the contents of the RO output section do not have to be moved or copied from their load address to their execution address.

The execution address of the RW region is also the same as its load address, so the contents of the RW output section are not moved from their load address to their execution address. However, the ZI output section is created at run-time and is placed contiguously with the RW region.

Specify the load and execution address using the following linker options:

`--ro_base=`*`address`*

Instructs `armlink` to set the load and execution address of the region containing the RO section at a four-byte aligned *`address`*, for example, the address of the first location in ROM. If you do not use the `--ro_base` option to specify the address, the default value of `0x8000` is used by `armlink`.

`--rw_base=address`

Instructs `armlink` to set the execution address of the region containing the RW output section at a four-byte aligned *`address`*. If this option is used with `--split`, this specifies both the load and execution addresses of the RW region, for example, a root region.

`--split`

Splits the default single load region, that contains both the RO and RW output sections, into two root load regions:

- One containing the RO output section.
- One containing the RW output section.

You can then place them separately using `--ro_base` and `--rw_base`.

Load view for images containing XO sections

For images that contain XO sections, the XO output section is placed at the address specified by `--ro_base`. The RO and RW output sections are placed consecutively and immediately after the XO section.

If you use `--split`, then the one load region contains the XO and RO output sections, and the other contains the RW output section.

Execution view for images containing XO sections

For images that contain XO sections, the XO execution region is placed at the address specified by `--ro_base`. The RO execution region is placed contiguously and immediately after the XO execution region.

If you specify `--split`, then the XO and RO execution regions are placed in the first load region, and the RW and ZI execution regions are placed in the second load region.

If you specify `--xo_base` *`address`*, then the XO execution region is placed at the specified address in a separate load region from the RO execution region.

Note

XO memory is supported only for Armv7-M and Armv8-M architectures.

Related concepts

3.1 The structure of an Arm® ELF image on page 3-33

3.1.2 Input sections, output sections, regions, and program segments on page 3-34

3.1.3 Load view and execution view of an image on page 3-35

3.2.3 Type 2 image structure, one load region and non-contiguous execution regions on page 3-43

Related reference

11.114 --ro_base=address on page 11-344

11.117 --rw_base=address on page 11-347

11.157 --xo_base=address on page 11-389

11.127 --split on page 11-359

3.3 Section placement with the linker

The linker places input sections in a specific order by default, but you can specify an alternative sorting order if required.

This section contains the following subsections:

3.3.1 Default section placement

By default, the linker places input sections in a specific order within an execution region.

The sections are placed in the following order:

1. By attribute as follows:
   a. Read-only code.
   b. Read-only data.
   c. Read-write code.
   d. Read-write data.
   e. Zero-initialized data.
2. By input section name if they have the same attributes. Names are considered to be case-sensitive and are compared in alphabetical order using the ASCII collation sequence for characters.
3. By a tie-breaker if they have the same attributes and section names. By default, it is the order that `armlink` processes the section. You can override the tie-breaker and sorting by input section name with the `FIRST` or `LAST` input section attribute.

Note

The sorting order is unaffected by ordering of section selectors within execution regions.

These rules mean that the positions of input sections with identical attributes and names included from libraries depend on the order the linker processes objects. This can be difficult to predict when many libraries are present on the command line. The `--tiebreaker=`*`cmdline`* option uses a more predictable order based on the order the section appears on the command line.

The base address of each input section is determined by the sorting order defined by the linker, and is correctly aligned within the output section that contains it.

The linker produces one output section for each attribute present in the execution region:

- One *execute-only* (XO) section if the execution region contains only XO sections.
- One RO section if the execution region contains read-only code or data.
- One RW section if the execution region contains read-write code or data.
- One ZI section if the execution region contains zero-initialized data.

Note

If an attempt is made to place data in an XO only execution region, then the linker generates an error.

XO sections lose the XO property if mixed with RO code in the same Execution region.

The XO and RO output sections can be protected at run-time on systems that have memory management hardware. RO and XO sections can be placed in ROM or Flash.

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

Note

XO memory is supported only for Armv7-M and Armv8-M architectures.

Example

The following scatter file shows how the linker places sections:

```
LoadRegion 0x8000
{
    ExecRegion1 0x0000 0x4000
    {
        *(sections)
        *(moresections)
    }
    ExecRegion2 0x4000 0x2000
    {
        *(evenmoresections)
    }
}
```

The order of execution regions within the load region is not altered by the linker.

Handling unassigned sections

The linker might not be able to place some input sections in any execution region.

When the linker is unable to place some input sections it generates an error message. This might occur because your current scatter file does not permit all possible module select patterns and input section selectors.

How you fix this depends on the importance of placing these sections correctly:

- If the sections must be placed at specific locations, then modify your scatter file to include specific module selectors and input section selectors as required.
- If the placement of the unassigned sections is not important, you can use one or more `.ANY` module selectors with optional input section selectors.

Related concepts

7.2.3 Methods of placing functions and data at specific addresses on page 7-114

7.3 Example of how to explicitly place a named section with scatter-loading on page 7-126

3.1 The structure of an Arm® ELF image on page 3-33

3.6 Linker-generated veneers on page 3-53

3.3.3 Section alignment with the linker on page 3-50

3.1.2 Input sections, output sections, regions, and program segments on page 3-34

3.3.2 Section placement with the FIRST and LAST attributes on page 3-49

3.5 Linker reordering of execution regions containing T32 code on page 3-52

Related reference

7.4 Placement of unassigned sections on page 7-128

8.5.2 Syntax of an input section description on page 8-181

11.126 --sort=algorithm on page 11-357

3.3.2 Section placement with the FIRST and LAST attributes

You can make sure that a section is placed either first or last in its execution region. For example, you might want to make sure the section containing the vector table is placed first in the image.

To do this, use one of the following methods:

- If you are not using scatter-loading, use the `--first` and `--last` linker command-line options to place input sections.
- If you are using scatter-loading, use the attributes `FIRST` and `LAST` in the scatter file to mark the first and last input sections in an execution region if the placement order is important.

  Caution

  `FIRST` and `LAST` must not violate the basic attribute sorting order. For example, `FIRST RW` is placed after any read-only code or read-only data.

Related concepts

3.1 The structure of an Arm® ELF image on page 3-33

3.1.2 Input sections, output sections, regions, and program segments on page 3-34

3.1.3 Load view and execution view of an image on page 3-35

7.1 The scatter-loading mechanism on page 7-105

Related reference

8.5.2 Syntax of an input section description on page 8-181

11.48 --first=section_id on page 11-270

11.69 --last=section_id on page 11-295

3.3.3 Section alignment with the linker

The linker ensures each input section starts at an address that is a multiple of the input section alignment.

When input sections have been ordered and before the base addresses are fixed, `armlink` inserts padding, if required, to force each input section to start at an address that is a multiple of the input section alignment.

`armlink` supports strict conformance with the ELF specification with the default option `--no_legacyalign`. The linker faults the base address of a region if it is not aligned so padding might be inserted to ensure compliance. With `--no_legacyalign`, the region alignment is the maximum alignment of any input section contained by the region.

If you use the option `--legacyalign`, the linker permits ELF program headers and output sections to be aligned on a four-byte boundary regardless of the maximum alignment of the input sections. This enables `armlink` to minimize the amount of padding that it inserts into the image.

If you are using scatter-loading, you can increase the alignment of a load region or execution region with the `ALIGN` attribute. For example, you can change an execution region that is normally four-byte aligned to be eight-byte aligned. However, you cannot reduce the natural alignment. For example, you cannot force two-byte alignment on a region that is normally four-byte aligned.

Related concepts

8.6.9 Example of aligning a base address in execution space but still tightly packed in load space on page 8-192

Related tasks

7.9 Aligning regions to page boundaries on page 7-147

Related reference

8.3.3 Load region attributes on page 8-169

11.70 --legacyalign, --no_legacyalign on page 11-296

8.4.3 Execution region attributes on page 8-175

3.4 Linker support for creating demand-paged files

The linker provides features for you to create files that are memory mapped.

In operating systems that support virtual memory, an ELF file can be loaded by mapping the ELF files into the address space of the process loading the file. When a virtual address in a page that is mapped to the file is accessed, the operating system loads that page from disk. ELF files that are to be used this way must conform to a certain format.

Use the `--paged` command-line option to enable demand paging mode. This helps produce ELF files that can be demand paged efficiently.

The basic constraints for a demand-paged ELF file are:

- There is no difference between the load and execution address for any output section.
- All PT_LOAD Program Headers have a minimum alignment, `pt_align`, of the page size for the operating system.
- All PT_LOAD Program Headers have a file offset, `pt_offset`, that is congruent to the virtual address (`pt_addr`) modulo `pt_align`.

When you specify `--paged`:

- The 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 applies.
- The operating system page size is controlled by the `--pagesize` command-line option.
- The linker attempts to place the ELF Header and Program Header in the first PT_LOAD program header, if space is available.

Example

This is an example of a demand paged scatter file:

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

Related concepts

7.1 The scatter-loading mechanism on page 7-105

Related tasks

7.9 Aligning regions to page boundaries on page 7-147

Related reference

11.120 --scatter=filename on page 11-350

8.6.7 GetPageSize() function on page 8-191

11.98 --paged on page 11-327

11.99 --pagesize=pagesize on page 11-328

8.6.8 SizeOfHeaders() function on page 8-191

3.5 Linker reordering of execution regions containing T32 code

The linker reorders execution regions containing T32 code only if the size of the T32 code exceeds the branch range.

If the code size of an execution region exceeds the maximum branch range of a T32 instruction, then `armlink` reorders the input sections using a different sorting algorithm. This sorting algorithm attempts to minimize the amount of veneers generated.

The T32 branch instructions that can be veneered are always encoded as a pair of 16-bit instructions. Processors that support Thumb-2 technology have a range of 16MB. Processors that do not support Thumb-2 technology have a range of 4MB.

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

Related concepts

3.6 Linker-generated veneers on page 3-53

Related reference

11.68 --largeregions, --no_largeregions on page 11-293

3.6 Linker-generated veneers

Veneers are small sections of code generated by the linker and inserted into your program.

This section contains the following subsections:

3.6.1 What is a veneer?

A veneer extends the range of a branch by becoming the intermediate target of the branch instruction.

The range of a `BL` instruction depends on the architecture:

- For AArch32 state, the range is 32MB for A32 instructions, 16MB for 32-bit T32 instructions, and 4MB for 16-bit T32 instructions. A veneer extends the range of the branch by becoming the intermediate target of the branch instruction. The veneer then sets the PC to the destination address.

  This enables the veneer to branch anywhere in the 4GB address space. If the veneer is inserted between A32 and T32 code, the veneer also handles instruction set state change.
- For AArch64 state, the range is 128MB. A veneer extends the range of the branch by becoming the intermediate target of the branch instruction. The veneer then loads the destination address and branches to it.

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

  Note

  There are no state-change veneers in AArch64 state.

The linker can generate the following veneer types depending on what is required:

- Inline veneers.
- Short branch veneers.
- Long branch veneers.

`armlink` creates one input section called `Veneer$$Code` for each veneer. A veneer is generated only if no other existing veneer can satisfy the requirements. If two input sections contain a long branch to the same destination, only one veneer is generated that is shared by both branch instructions. A veneer is only shared in this way if it can be reached by both sections.

Note

If *execute-only* (XO) sections are present, only XO-compliant veneer code is created in XO regions.

Related concepts

3.6.2 Veneer sharing on page 3-53

3.6.3 Veneer types on page 3-54

3.6.4 Generation of position independent to absolute veneers on page 3-55

3.6.5 Reuse of veneers when scatter-loading on page 3-55

3.6.2 Veneer sharing

If multiple objects result in the same veneer being created, the linker creates a single instance of that veneer. The veneer is then shared by those objects.

You can use the command-line option `--no_veneershare` to specify that veneers are not shared. This assigns ownership of the created veneer section to the object that created the veneer and so enables you to select veneers from a particular object in a scatter file, for example:

```
LR 0x8000
{
    ER_ROOT +0
    {
          object1.o(Veneer$$Code)
    }
}
```

Be aware that veneer sharing makes it impossible to assign an owning object. Using `--no_veneershare` provides a more consistent image layout. However, this comes at the cost of a significant increase in code size, because of the extra veneers generated by the linker.

Related concepts

3.6.1 What is a veneer? on page 3-53

7.1 The scatter-loading mechanism on page 7-105

Related reference

Chapter 8 Scatter File Syntax on page 8-164

11.152 --veneershare, --no_veneershare on page 11-384

3.6.3 Veneer types

Veneers have different capabilities and use different code pieces.

The linker selects the most appropriate, smallest, and fastest depending on the branching requirements:

- Inline veneer:
  - Performs only a state change.
  - The veneer must be inserted just before the target section to be in range.
  - An A32 to T32 interworking veneer has a range of 256 bytes so the function entry point must appear within 256 bytes of the veneer.
  - A T32 to A32 interworking veneer has a range of zero bytes so the function entry point must appear immediately after the veneer.
  - An inline veneer is always position-independent.
- Short branch veneer:
  - An interworking T32 to A32 short branch veneer has a range of 32MB, the range for an A32 instruction. An A64 short branch veneer has a range of 128MB.
  - A short branch veneer is always position-independent.
  - A Range Extension T32 to T32 short branch veneer for processors that support Thumb-2 technology.
- Long branch veneer:
  - Can branch anywhere in the address space.
  - All long branch veneers are also interworking veneers.
  - There are different long branch veneers for absolute or position-independent code.

When you are using veneers be aware of the following:

- The inline veneer limitations mean that you cannot move inline veneers out of an execution region using a scatter file. Use the command-line option `--no_inlineveneer` to prevent the generation of inline veneers.
- All veneers cannot be collected into one input section because the resulting veneer input section might not be within range of other input sections. If the sections are not within addressing range, long branching is not possible.
- The linker generates position-independent variants of the veneers automatically. However, because such veneers are larger than non position-independent variants, the linker only does this where necessary, that is, where the source and destination execution regions are both position-independent and are rigidly related.

To optimize the code size of veneers, `armlink` chooses the variant in the order of preference:

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

Related concepts

3.6.1 What is a veneer? on page 3-53

Related reference

11.87 --max_veneer_passes=value on page 11-316

11.64 --inlineveneer, --no_inlineveneer on page 11-288

3.6.4 Generation of position independent to absolute veneers

Calling from *position independent* (PI) code to absolute code requires a veneer.

The normal call instruction encodes the address of the target as an offset from the calling address. When calling from PI code to absolute code the offset cannot be calculated at link time, so the linker must insert a long-branch veneer.

The generation of PI to absolute veneers can be controlled using the `--piveneer` option, that is set by default. When this option is turned off using `--no_piveneer`, the linker generates an error when a call from PI code to absolute code is detected.

Note

Not supported for AArch64 state.

Related concepts

3.6.1 What is a veneer? on page 3-53

Related reference

11.87 --max_veneer_passes=value on page 11-316

11.102 --piveneer, --no_piveneer on page 11-331

3.6.5 Reuse of veneers when scatter-loading

The linker reuses veneers whenever possible, but there are some limitations on the reuse of veneers in protected load regions and overlaid execution regions.

A scatter file enables you to create regions that share the same area of RAM:

- If you use the `PROTECTED` attribute for a load region it prevents:
  - Overlapping of load regions.
  - Veneer sharing.
  - String sharing with the `--merge` option.
- If you use the `AUTO_OVERLAY` attribute for a region, no other execution region can reuse a veneer placed in an overlay execution region.
- If you use the `OVERLAY` attribute for a region, no other execution region can reuse a veneer placed in an overlay execution region.

If it is not possible to reuse a veneer, new veneers are created instead. Unless you have instructed the linker to place veneers somewhere specific using scatter-loading, a veneer is usually placed in the execution region that contains the call requiring the veneer. However, in some situations the linker has to place the veneer in an adjacent execution region, either to maximize sharing opportunities or for a short branch veneer to reach its target.

Related concepts

3.6.1 What is a veneer? on page 3-53

8.3.4 Inheritance rules for load region address attributes on page 8-170

8.3.5 Inheritance rules for the RELOC address attribute on page 8-171

8.4.4 Inheritance rules for execution region address attributes on page 8-178

Related reference

8.3.3 Load region attributes on page 8-169

3.6.6 Generation of secure gateway veneers

`armlink` can generate secure gateway veneers for symbols that are present in a Secure image. It can also output symbols to a specified output import library, when necessary.

`armlink` generates a secure gateway veneer when it finds in the Secure image an entry function that has both symbols `__acle_se_<entry>` and `<entry>` pointing to the same offset in the same section.

The secure gateway veneer is a sequence of two instructions:

```
<entry>:
        SG
        B.W __acle_se_<entry>
```

The original symbol `<entry>` is changed to point to the `SG` instruction of the secure gateway veneer.

You can specify an input import library and output import library with the following command-line options:

- `--import_cmse_lib_in=`*`filename`*.
- `--import_cmse_lib_out=`*`filename`*.

Placement of secure gateway veneers is controlled by an input import library and by a scatter file selection. The linker can also output addresses of secure gateways to an output import library.

Example

The following example shows the generation of a secure gateway veneer:

Input code:

```
        .text
entry:
__acle_se_entry:
        [entry's code]
        BXNS lr
```

Output code produced by `armlink`:

```
        .text
__acle_se_entry:
        [entry's code]
        BXNS lr

        .section Veneer$$CMSE, "ax"
entry:
        SG
        B.W __acle_se_entry
```

Related concepts

7.6 Placement of CMSE veneer sections for a Secure image on page 7-140

Related reference

11.57 --import_cmse_lib_in=filename on page 11-279
11.58 --import_cmse_lib_out=filename on page 11-280

Related information

Building Secure and Non-secure Images Using Armv8-M Security Extensions

3.7 Command-line options used to control the generation of C++ exception tables

You can control the generation of C++ exception tables using command-line options.

By default, or if the option `--exceptions` is specified, the image can contain exception tables. Exception tables are discarded silently if no code throws an exception. However, if the option `--no_exceptions` is specified, the linker generates an error if any exceptions tables are present after unused sections have been eliminated.

You can use the `--no_exceptions` option to ensure that your code is exceptions free. The linker generates an error message to highlight that exceptions have been found and does not produce a final image.

However, you can use the `--no_exceptions` option with the `--diag_warning` option to downgrade the error message to a warning. The linker produces a final image but also generates a message to warn you that exceptions have been found.

Related reference

11.31 --diag_warning=tag[,tag,...] on page 11-253
11.43 --exceptions, --no_exceptions on page 11-265

Related information

-fno-exceptions compiler option

3.8 Weak references and definitions

Weak references and definitions provide additional flexibility in the way the linker includes various functions and variables in a build.

Weak references and definitions are typically used in connection with library functions.

Weak references

If the linker cannot resolve normal, non-weak, references to symbols from the content loaded so far, it attempts to do so by finding the symbol in a library:

- If it is unable to find such a reference, the linker reports an error.
- If such a reference is resolved, a section that is reachable from an entry point by at least one non-weak reference is marked as used. This ensures the section is not removed by the linker as an unused section. Each non-weak reference must be resolved by exactly one definition. If there are multiple definitions, the linker reports an error.

Symbols can be given weak binding by the compiler and assembler.

The linker does not load an object from a library to resolve a weak reference. It is able to resolve the weak reference only if the definition is included in the image for other reasons. The weak reference does not cause the linker to mark the section containing the definition as used, so it might be removed by the linker as unused. The definition might already exist in the image for several reasons:

- The symbol has a non-weak reference from somewhere else in the code.
- The symbol definition exists in the same ELF section as a symbol definition that is included for any of these reasons.
- The symbol definition is in a section that has been specified using `--keep`, or contains an `ENTRY` point.
- The symbol definition is in an object file included in the link and the `--no_remove` option is used. The object file is not referenced from a library unless that object file within the library is explicitly included on the linker command-line.

In summary, a weak reference is resolved if the definition is already included in the image, but it does not determine if that definition is included.

An unresolved weak function call is replaced with either:

- A no-operation instruction, `NOP`.
- A branch with link instruction, `BL`, to the following instruction. That is, the function call just does not happen.

Weak definitions

You can mark a function or variable definition as weak in a source file. A weak symbol definition is then present in the created object file.

You can use a weak definition to resolve any reference to that symbol in the same way as a normal definition. However, if another non-weak definition of that symbol exists in the build, the linker uses that definition instead of the weak definition, and does not produce an error because of multiply-defined symbols.

Example of a weak reference

A library contains a function `foo()`, that is called in some builds of an application but not in others. If it is used, `init_foo()` must be called first. You can use weak references to automate the call to `init_foo()`.

The library can define `init_foo()` and `foo()` in the same ELF section. The application initialization code must call `init_foo()` weakly. If the application includes `foo()` for any reason, it also includes

`init_foo()` and this is called from the initialization code. In any builds that do not include `foo()`, the call to `init_foo()` is removed by the linker.

Typically, the code for multiple functions defined within a single source file is placed into a single ELF section by the compiler. However, certain build options might alter this behavior, so you must use them with caution if your build is relying on the grouping of files into ELF sections. The compiler command-line option `-ffunction-sections` results in each function being placed in its own section. In this example, compiling the library with this option results in `foo()` and `init_foo()` being placed in separate sections. Therefore `init_foo()` is not automatically included in the build due to a call to `foo()`.

In this example, there is no need to rebuild the initialization code between builds that include `foo()` and do not include `foo()`. There is also no possibility of accidentally building an application with a version of the initialization code that does not call `init_foo()`, and other parts of the application that call `foo()`.

An example of `foo.c` source code that is typically built into a library is:

```
void init_foo()
{
    // Some initialization code
}
void foo()
{
     // A function that is included in some builds
     // and requires init_foo() to be called first.
}
```

An example of `init.c` is:

```
__attribute__((weak)) void init_foo(void);
int main(void)
{
    init_foo();
    //  Rest of code that may make calls to foo() directly or indirectly.
}
```

An example of a weak reference generated by the assembler is:

```
init.s:
  main:
    ...
    bl      init_foo
    // Rest of code

    .weak   init_foo
```

Example of a weak definition

You can provide a simple or dummy implementation of a function as a weak definition. This enables you to build software with defined behavior without having to provide a full implementation of the function. It also enables you to provide a full implementation for some builds if required.

Related concepts

3.9 How the linker performs library searching, selection, and scanning on page 3-60

3.12 How the linker resolves references on page 3-63

Related reference

11.66 --keep=section_id on page 11-290

11.113 --remove, --no_remove on page 11-343

Related information

EXPORT or GLOBAL

IMPORT and EXTERN

NOP

B

ENTRY

3.9 How the linker performs library searching, selection, and scanning

The linker always searches user libraries before the Arm libraries.

If you specify the `--no_scanlib` command-line option, the linker does not search for the default Arm libraries and uses only those libraries that are specified in the input file list to resolve references.

The linker creates an internal list of libraries as follows:

1. Any libraries explicitly specified in the input file list are added to the list.
2. The user-specified search path is examined to identify Arm standard libraries to satisfy requests embedded in the input objects.

   The best-suited library variants are chosen from the searched directories and their subdirectories. Libraries supplied by Arm have multiple variants that are named according to the attributes of their members.

Be aware of the following differences between the way the linker adds object files to the image and the way it adds libraries to the image:

- Each object file in the input list is added to the output image unconditionally, whether or not anything refers to it. At least one object must be specified.
- A member from a library is included in the output only if:
  - An object file or an already-included library member makes a non-weak reference to it.
  - The linker is explicitly instructed to add it.

Note

If a library member is explicitly requested in the input file list, the member is loaded even if it does not resolve any current references. In this case, an explicitly requested member is treated as if it is an ordinary object.

Unresolved references to weak symbols do not cause library members to be loaded.

Related concepts

3.10 How the linker searches for the Arm® standard libraries on page 3-61

Related reference

11.66 --keep=section_id on page 11-290
11.113 --remove, --no_remove on page 11-343
11.119 --scanlib, --no_scanlib on page 11-349

3.10 How the linker searches for the Arm® standard libraries

The linker searches for the Arm standard libraries using information specified on the command-line, or by examining environment variables.

By default, the linker searches for the Arm standard libraries in `../lib`, relative to the location of the `armlink` executable. Use the `--libpath` command-line option to specify a different location.

The --libpath command-line option

Use the `--libpath` command-line option with a comma-separated list of parent directories. This list must end with the parent directory of the Arm library directories `armlib`, `cpplib`, and `libcxx`.

The sequential nature of the search ensures that `armlink` chooses the library that appears earlier in the list if two or more libraries define the same symbol.

Library search order

The linker searches for libraries in the following order:

1. At the location specified with the command-line option `--libpath`.
2. In `../lib`, relative to the location of the `armlink` executable.

How the linker selects Arm® library variants

The Arm Compiler toolchain includes a number of variants of each of the libraries, that are built using different build options. For example, architecture versions, endianness, and instruction set. The variant of the Arm library is coded into the library name. The linker must select the best-suited variant from each of the directories identified during the library search.

The linker accumulates the attributes of each input object and then selects the library variant best suited to those attributes. If more than one of the selected libraries are equally suited, the linker retains the first library selected and rejects all others.

The `--no_scanlib` option prevents the linker from searching the directories for the Arm standard libraries.

Related concepts

3.9 How the linker performs library searching, selection, and scanning on page 3-60

Related reference

11.71 --libpath=pathlist on page 11-297

Related information

C and C++ library naming conventions

The C and C++ libraries

Toolchain environment variables

3.11 Specifying user libraries when linking

You can specify your own libraries when linking.

To specify user libraries, either:

- Include them with path information explicitly in the input file list.
- Add the `--userlibpath` option to the `armlink` command line with a comma-separated list of directories, and then specify the names of the libraries as input files.

You can use the `--library=`*`name`* option to specify static libraries, `lib`*`name`*`.a`.

If you do not specify a full path name to a library on the command line, the linker tries to locate the library in the directories specified by the `--userlibpath` option. For example, if the directory `/mylib` contains `my_lib.a` and `other_lib.a`, add `/mylib/my_lib.a` to the input file list with the command:

```
armlink --userlibpath /mylib my_lib.a *.o
```

If you add a particular member from a library this does not add the library to the list of searchable libraries used by the linker. To load a specific member and add the library to the list of searchable libraries include the library *`filename`* on its own as well as specifying *`library(member)`*. For example, to load `strcmp.o` and place `mystring.lib` on the searchable library list add the following to the input file list:

```
mystring.lib(strcmp.o) mystring.lib
```

Note

Any search paths used for the Arm standard libraries specified by the linker command-line option `--libpath` are not searched for user libraries.

Related concepts

3.10 How the linker searches for the Arm® standard libraries on page 3-61

Related reference

11.71 --libpath=pathlist on page 11-297

11.148 --userlibpath=pathlist on page 11-380

Related information

The C and C++ libraries

Toolchain environment variables

3.12 How the linker resolves references

When the linker has constructed the list of libraries, it repeatedly scans each library in the list to resolve references.

`armlink` maintains two separate lists of files. The lists are scanned in the following order to resolve all dependencies:

1. The list of user files and libraries that have been loaded.
2. List of Arm standard libraries found in a directory relative to the `armlink` executable, or the directories specified by `--libpath`.

Each list is scanned using the following process:

1. Scan each of the libraries to load the required members:
   a. For each currently unsatisfied non-weak reference, search sequentially through the list of libraries for a matching definition. The first definition found is marked for processing in step *1.b*.

      The sequential nature of the search ensures that the linker chooses the library that appears earlier in the list if two or more libraries define the same symbol. This enables you to override function definitions from other libraries, for example, the Arm C libraries, by adding your libraries to the input file list. However you must be careful to consistently override all the symbols in a library member. If you do not, you risk the objects from both libraries being loaded when there is a reference to an overridden symbol and a reference to a symbol that was not overridden. This results in a multiple symbol definition error `L6200E` for each overridden symbol.
   b. Load the library members marked in step *1.a*. As each member is loaded it might satisfy some unresolved references, possibly including weak ones. Loading a library member might also create new unresolved weak and non-weak references.
   c. Repeat these stages until all non-weak references are either resolved or cannot be resolved by any library.
2. If any non-weak reference remains unsatisfied at the end of the scanning operation, generate an error message.

Related concepts

3.9 How the linker performs library searching, selection, and scanning on page 3-60

3.10 How the linker searches for the Arm® standard libraries on page 3-61

Related tasks

3.11 Specifying user libraries when linking on page 3-62

Related reference

11.71 --libpath=pathlist on page 11-297

Related information

Toolchain environment variables

List of the armlink error and warning messages

3.13 The strict family of linker options

The linker provides options to overcome the limitations of the standard linker checks.

The strict options are not directly related to error severity. Usually, you add a strict option because the standard linker checks are not precise enough or are potentially noisy with legacy objects.

The strict options are:

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

Related reference

11.130 --strict on page 11-362

11.134 --strict_relocations, --no_strict_relocations on page 11-366

11.135 --strict_symbols, --no_strict_symbols on page 11-367

11.136 --strict_visibility, --no_strict_visibility on page 11-368

Chapter 4
Linker Optimization Features

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

It contains the following sections:

4.1 Elimination of common section groups

The linker can detect multiple copies of section groups, and discard the additional copies.

Arm Compiler generates complete objects for linking. Therefore:

- If there are inline functions in C and C++ sources, each object contains the out-of-line copies of the inline functions that the object requires.
- If templates are used in C++ sources, each object contains the template functions that the object requires.

When these functions are declared in a common header file, the functions might be defined many times in separate objects that are subsequently linked together. To eliminate duplicates, the compiler compiles these functions into separate instances of common section groups.

It is possible that the separate instances of common section groups, are not identical. Some of the copies, for example, might be found in a library that has been built with different, but compatible, build options, different optimization, or debug options.

If the copies are not identical, `armlink` retains the best available variant of each common section group, based on the attributes of the input objects. `armlink` discards the rest.

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

You control this optimization with the following linker options:

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

  The image changes if you compile all files containing a COMDAT group A with `-g`, even if you use `--no_bestdebug`.

Related concepts

4.2 Elimination of unused sections on page 4-67

3.1.2 Input sections, output sections, regions, and program segments on page 3-34

Related reference

11.8 --bestdebug, --no_bestdebug on page 11-227

4.2 Elimination of unused sections

Elimination of unused sections is the most significant optimization on image size that the linker performs.

Unused section elimination:

- Removes unreachable code and data from the final image.
- Is suppressed in cases that might result in the removal of all sections.

To control this optimization, use the `--remove`, `--no_remove`, `--first`, `--last`, and `--keep` linker options.

Unused section elimination requires an entry point. Therefore, if no entry point is specified for an image, use the `--entry` linker option to specify an entry point.

Use the `--info unused` linker option to instruct the linker to generate a list of the unused sections that it eliminates.

An input section is retained in the final image when:

- It contains an entry point or an externally accessible symbol, for example, an entry function into the secure code for Armv8-M Security Extensions.
- It is an `SHT_INIT_ARRAY`, `SHT_FINI_ARRAY`, or `SHT_PREINIT_ARRAY` section.
- It is specified as the first or last input section, either by the `--first` or `--last` option or by a scatter-loading equivalent.
- It is marked as unremovable by the `--keep` option.
- It is referred to, directly or indirectly, by a non-weak reference from an input section retained in the image.
- Its name matches the name referred to by an input section symbol, and that symbol is referenced from a section that is retained in the image.

Note

Compilers usually collect functions and data together and emit one section for each category. The linker can only eliminate a section if it is entirely unused.

You can mark a function or variable in source code with the `__attribute__((used))` attribute. This attribute causes `armclang` to generate the symbol `__tagsym$$used.`*num* for each function or variable, where *num* is a counter to differentiate each symbol. Unused section elimination does not remove a section that contains `__tagsym$$used.`*num*.

You can also use the `-ffunction-sections` compiler command-line option to instruct the compiler to generate one ELF section for each function in the source file.

Related concepts

4.1 Elimination of common section groups on page 4-66
3.1.2 Input sections, output sections, regions, and program segments on page 3-34
3.8 Weak references and definitions on page 3-58

Related reference

11.113 --remove, --no_remove on page 11-343
11.41 --entry=location on page 11-263
11.48 --first=section_id on page 11-270
11.66 --keep=section_id on page 11-290
11.69 --last=section_id on page 11-295
11.59 --info=topic[,topic,...] on page 11-281

Related information

Building Secure and Non-secure Images Using Armv8-M Security Extensions

4.3 Optimization with RW data compression

RW data areas typically contain a large number of repeated values, such as zeros, that makes them suitable for compression.

RW data compression is enabled by default to minimize ROM size.

The linker compresses the data. This data is then decompressed on the target at run time.

The Arm libraries contain some decompression algorithms and the linker chooses the optimal one to add to your image to decompress the data areas when the image is executed. You can override the algorithm chosen by the linker.

Note

Not supported for AArch64 state.

This section contains the following subsections:

4.3.1 How the linker chooses a compressor

`armlink` gathers information about the content of data sections before choosing the most appropriate compression algorithm to generate the smallest image.

If compression is appropriate, `armlink` can only use one data compressor for all the compressible data sections in the image. Different compression algorithms might be tried on these sections to produce the best overall size. Compression is applied automatically if:

```
Compressed data size + Size of decompressor < Uncompressed data size
```

When a compressor has been chosen, `armlink` adds the decompressor to the code area of your image. If the final image does not contain any compressed data, no decompressor is added.

Related concepts

4.3.2 Options available to override the compression algorithm used by the linker on page 4-68

4.3 Optimization with RW data compression on page 4-68

4.3.3 How compression is applied on page 4-69

4.3.4 Considerations when working with RW data compression on page 4-69

4.3.2 Options available to override the compression algorithm used by the linker

The linker has options to disable compression or to specify a compression algorithm to be used.

You can override the compression algorithm used by the linker by either:

- Using the `--datacompressor off` option to turn off compression.
- Specifying a compression algorithm.

To specify a compression algorithm, use the number of the required compressor on the linker command line, for example:

```
armlink --datacompressor 2 …
```

Use the command-line option `--datacompressor list` to get a list of compression algorithms available in the linker:

```
armlink --datacompressor list…
Num     Compression algorithm
=========================================================
0       Run-length encoding
```

```
1       Run-length encoding, with LZ77 on small-repeats
2       Complex LZ77 compression
```

When choosing a compression algorithm be aware that:

- Compressor 0 performs well on data with large areas of zero-bytes but few nonzero bytes.
- Compressor 1 performs well on data where the nonzero bytes are repeating.
- Compressor 2 performs well on data that contains repeated values.

The linker prefers compressor 0 or 1 where the data contains mostly zero-bytes (>75%). Compressor 2 is chosen where the data contains few zero-bytes (<10%). If the image is made up only of A32 code, then A32 decompressors are used automatically. If the image contains any T32 code, T32 decompressors are used. If there is no clear preference, all compressors are tested to produce the best overall size.

Note

It is not possible to add your own compressors into the linker. The algorithms that are available, and how the linker chooses to use them, might change in the future.

Related concepts

4.3 Optimization with RW data compression on page 4-68

4.3.3 How compression is applied on page 4-69

4.3.1 How the linker chooses a compressor on page 4-68

4.3.4 Considerations when working with RW data compression on page 4-69

Related reference

11.25 --datacompressor=opt on page 11-247

4.3.3 How compression is applied

The linker applies compression depending on the compression type specified, and might apply additional compression on repeated phrases.

Run-length compression encodes data as non-repeated bytes and repeated zero-bytes. Non-repeated bytes are output unchanged, followed by a count of zero-bytes.

Lempel-Ziv 1977 (LZ77) compression keeps track of the last n bytes of data seen. When a phrase is encountered that has already been seen, it outputs a pair of values corresponding to:

- The position of the phrase in the previously-seen buffer of data.
- The length of the phrase.

Related concepts

4.3 Optimization with RW data compression on page 4-68

4.3.2 Options available to override the compression algorithm used by the linker on page 4-68

4.3.1 How the linker chooses a compressor on page 4-68

4.3.4 Considerations when working with RW data compression on page 4-69

Related reference

11.25 --datacompressor=opt on page 11-247

4.3.4 Considerations when working with RW data compression

There are some considerations to be aware of when working with RW data compression.

When working with RW data compression:

- Use the linker option `--map` to see where compression has been applied to regions in your code.
- The linker in *RealView Compiler Tools* (RVCT) v4.0 and later 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.

Related concepts

4.3 Optimization with RW data compression on page 4-68

4.3.1 How the linker chooses a compressor on page 4-68

4.3.2 Options available to override the compression algorithm used by the linker on page 4-68

4.3.3 How compression is applied on page 4-69

Related reference

6.3.3 Load$$ execution region symbols on page 6-91

Chapter 7 Scatter-loading Features on page 7-104

11.85 --map, --no_map on page 11-314

Chapter 8 Scatter File Syntax on page 8-164

4.4 Function inlining with the linker

The linker inlines functions depending on what options you specify and the content of the input files.

The linker can inline small functions in place of a branch instruction to that function. For the linker to be able to do this, the function (without the return instruction) must fit in the four bytes of the branch instruction.

Use the `--inline` and `--no_inline` command-line options to control branch inlining. However, `--no_inline` only turns off inlining for user-supplied objects. The linker still inlines functions from the Arm standard libraries by default.

If branch inlining optimization is enabled, the linker scans each function call in the image and then inlines as appropriate. When the linker finds a suitable function to inline, it replaces the function call with the instruction from the function that is being called.

The linker applies branch inlining optimization before any unused sections are eliminated so that inlined sections can also be removed if they are no longer called.

Note

- For Armv7-A, the linker can inline two 16-bit encoded Thumb® instructions in place of the 32-bit encoded Thumb `BL` instruction.
- For Armv8-A and Armv8-M, the linker can inline two 16-bit T32 instructions in place of the 32-bit T32 `BL` instruction.

Use the `--info=inline` command-line option to list all the inlined functions.

Note

The linker does not inline small functions in AArch64 state.

Related concepts

4.5 Factors that influence function inlining on page 4-72

4.2 Elimination of unused sections on page 4-67

Related reference

11.59 --info=topic[,topic,...] on page 11-281

11.62 --inline, --no_inline on page 11-286

4.5 Factors that influence function inlining

There are a number of factors that influence how the linker inlines functions.

The following factors influence the way functions are inlined:

- The linker handles only the simplest cases and does not inline any instructions that read or write to the PC because this depends on the location of the function.
- If your image contains both A32 and T32 code, functions that are called from the opposite state must be built for interworking. The linker can inline functions containing up to two 16-bit T32 instructions. However, an A32 calling function can only inline functions containing either a single 16-bit encoded T32 instruction or a 32-bit encoded T32 instruction.
- The action that the linker takes depends on the size of the function being called. The following table shows the state of both the calling function and the function being called:

Table 4-1 Inlining small functions

Calling function state	Called function state	Called function size
A32	A32	4 to 8 bytes
A32	T32	2 to 6 bytes
T32	T32	2 to 6 bytes

  The linker can inline in different states if there is an equivalent instruction available. For example, if a T32 instruction is `adds r0, r0` then the linker can inline the equivalent A32 instruction. It is not possible to inline from A32 to T32 because there is less chance of T32 equivalent to an A32 instruction.
- For a function to be inlined, the last instruction of the function must be either:

```
MOV pc, lr
```

  or

```
BX lr
```

  A function that consists only of a return sequence can be inlined as a `NOP`.
- A conditional A32 instruction can only be inlined if either:
  — The condition on the `BL` matches the condition on the instruction being inlined. For example, `BLEQ` can only inline an instruction with a matching condition like `ADDEQ`.
  — The `BL` instruction or the instruction to be inlined is unconditional. An unconditional A32 `BL` can inline any conditional or unconditional instruction that satisfies all the other criteria. An instruction that cannot be conditionally executed cannot be inlined if the `BL` instruction is conditional.
- A `BL` that is the last instruction of a T32 *If-Then* (IT) block cannot inline a 16-bit encoded T32 instruction or a 32-bit `MRS`, `MSR`, or `CPS` instruction. This is because the IT block changes the behavior of the instructions within its scope so inlining the instruction changes the behavior of the program.

Related concepts

4.6 About branches that optimize to a NOP on page 4-74

Related information

Conditional instructions
ADD
B
CPS
IT
MOV
MRS (PSR to general-purpose register)

MSR (general-purpose register to PSR)

4.6 About branches that optimize to a NOP

Although the linker can replace branches with a `NOP`, there might be some situations where you want to stop this happening.

By default, the linker replaces any branch with a relocation that resolves to the next instruction with a `NOP` instruction. This optimization can also be applied if the linker reorders tail calling sections.

However, there are cases where you might want to disable the option, for example, when performing verification or pipeline flushes.

To control this optimization, use the `--branchnop` and `--no_branchnop` command-line options.

Related concepts

4.7 Linker reordering of tail calling sections on page 4-75

Related reference

11.12 --branchnop, --no_branchnop on page 11-231

4.7 Linker reordering of tail calling sections

There are some situations when you might want the linker to reorder tail calling sections.

A tail calling section is a section that contains a branch instruction at the end of the section. If the branch instruction has a relocation that targets a function at the start of another section, the linker can place the tail calling section immediately before the called section. The linker can then optimize the branch instruction at the end of the tail calling section to a `NOP` instruction.

To take advantage of this behavior, use the command-line option `--tailreorder` to move tail calling sections immediately before their target.

Use the `--info=tailreorder` command-line option to display information about any tail call optimizations performed by the linker.

Note

The linker does not reorder tail calling functions in AArch64 state.

Related concepts

4.6 About branches that optimize to a NOP on page 4-74

4.8 Restrictions on reordering of tail calling sections on page 4-76

3.6.3 Veneer types on page 3-54

Related reference

11.59 --info=topic[,topic,...] on page 11-281

11.141 --tailreorder, --no_tailreorder on page 11-373

4.8 Restrictions on reordering of tail calling sections

There are some restrictions on the reordering of tail calling sections.

The linker:

- Can only move one tail calling section for each tail call target. If there are multiple tail calls to a single section, the tail calling section with an identical section name is moved before the target. If no section name is found in the tail calling section that has a matching name, then the linker moves the first section it encounters.
- Cannot move a tail calling section out of its execution region.
- Does not move tail calling sections before inline veneers.

Related concepts

4.7 Linker reordering of tail calling sections on page 4-75

4.9 Linker merging of comment sections

If input files have any comment sections that are identical, then the linker can merge them.

If input object files have any `.comment` sections that are identical, then the linker merges them to produce the smallest `.comment` section while retaining all useful information.

The linker associates each input `.comment` section with the filename of the corresponding input object. If it merges identical `.comment` sections, then all the filenames that contain the common section are listed before the section contents, for example:

```
file1.o
file2.o
.comment section contents.
```

The linker merges these sections by default. To prevent the merging of identical `.comment` sections, use the `--no_filtercomment` command-line option.

Note

`armlink` does not preprocess comment sections from `armclang`. If you do not want to retain the information in a `.comment` section, then use the `fromelf` command with the `--strip=comment` option to strip this section from the image.

Related reference

11.20 --comment_section, --no_comment_section on page 11-240

11.46 --filtercomment, --no_filtercomment on page 11-268

Related information

--strip (fromelf option)

4.10 Merging identical constants

The linker can attempt to merge identical constants in objects targeted at AArch32 state. The objects must be produced with Arm Compiler 6. If you compile with the `armclang -ffunction-sections` option, the merge is more efficient. This option is the default.

The following procedure is an example that shows the merging feature.

Procedure

1. Create a C source file, `litpool.c`, containing the following code:

```
int f1() {
    return 0xdeadbeef;
}
int f2() {
    return 0xdeadbeef;
}
```

2. Compile the source with `-S` to create an assembly file:

   ```
   armclang -c -S -target arm-arm-none-eabi -mcpu=cortex-m0 -ffunction-sections
   litpool.c -o litpool.s
   ```

   Note

   `-ffunction-sections` is the default.

   Results:
   Because `0xdeadbeef` is a difficult constant to create using instructions, a literal pool is created, for example:

```
...
f1:
    .fnstart
@ BB#0:
    ldr    r0, __arm_cp.0_0
    bx    lr
    .p2align    2
@ BB#1:
__arm_cp.0_0:
    .long    3735928559              @ 0xdeadbeef
...
    .fnend

...
    .code    16                      @ @f2
    .thumb_func
f2:
    .fnstart
@ BB#0:
    ldr    r0, __arm_cp.1_0
    bx    lr
    .p2align    2
@ BB#1:
__arm_cp.1_0:
    .long    3735928559              @ 0xdeadbeef
...
    .fnend
...
```

   Note

   There is one copy of the constant for each function, because `armclang` cannot share these constants between both functions.

3. Compile the source to create an object:

   ```
   armclang -c -target arm-arm-none-eabi -mcpu=cortex-m0 litpool.c -o litpool.o
   ```

4. Link the object file using the `--merge_litpools` option:

   ```
   armlink --cpu=Cortex-M0 --merge_litpools litpool.o -o litpool.axf
   ```

Note

`--merge_litpools` is the default.

5. Run `fromelf` to view the image structure:

   ```
   fromelf -c -d -s -t -v -z litpool.axf
   ```

   Results: The following example shows the result of the merge:

```
...
    f1
        0x00008000:    4801        .H      LDR      r0,[pc,#4] ; [0x8008] = 0xdeadbeef
        0x00008002:    4770        pG      BX       lr
    f2
        0x00008004:    4800        .H      LDR      r0,[pc,#0] ; [0x8008] = 0xdeadbeef
        0x00008006:    4770        pG      BX       lr
    $d.4
    __arm_cp.1_0
        0x00008008:    deadbeef    ....    DCD    3735928559
...
```

Related reference

11.90 --merge_litpools, --no_merge_litpools on page 11-319

Related information

-ffunction-sections, -fno-function-sections (armclang option)

Chapter 5
Getting Image Details

Describes how to get image details from the Arm linker, `armlink`.

It contains the following sections:

5.1 Options for getting information about linker-generated files

The linker provides options for getting information about the files it generates.

You can use following options to get information about how your file is generated by the linker, and about the properties of the files:

`--info`

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.

`--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.

Related concepts

3.3.3 Section alignment with the linker on page 3-50

4.3 Optimization with RW data compression on page 4-68

Related tasks

5.2 Identifying the source of some link errors on page 5-82

Related reference

5.3 Example of using the --info linker option on page 5-83

11.59 --info=topic[,topic,...] on page 11-281

11.75 --list=filename on page 11-302

11.85 --map, --no_map on page 11-314

11.122 --show_cmdline on page 11-353

11.137 --symbols, --no_symbols on page 11-369

11.153 --verbose on page 11-385

11.158 --xref, --no_xref on page 11-390

11.159 --xrefdbg, --no_xrefdbg on page 11-391

5.2 Identifying the source of some link errors

The linker provides options to help you identify the source of some link errors.

To identify the source of some link errors, use `--info inputs`. 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.

Related reference

5.1 Options for getting information about linker-generated files on page 5-81
11.59 --info=topic[,topic,...] on page 11-281
11.153 --verbose on page 11-385

5.3 Example of using the --info linker option

An example of the `--info` output.

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:

```
Image component sizes

      Code (inc. data)   RO Data    RW Data    ZI Data      Debug   Object Name

        30         16          0          0          0          0   foo.o
        56         10        960          0       1024        372   startup_ARMCM7.o

    ----------------------------------------------------------------------
        88         26        992          0       5120        372   Object Totals
         0          0         32          0       4096          0   (incl. Generated)
         2          0          0          0          0          0   (incl. Padding)

    ----------------------------------------------------------------------

      Code (inc. data)   RO Data    RW Data    ZI Data      Debug   Library Member Name

         8          0          0          0          0         68   __main.o
         0          0          0          0          0          0   __rtentry.o
        12          0          0          0          0          0   __rtentry2.o
         8          4          0          0          0          0   __rtentry5.o
        52          8          0          0          0          0   __scatter.o
        26          0          0          0          0          0   __scatter_copy.o
        28          0          0          0          0          0   __scatter_zi.o
        10          0          0          0          0         68   defsig_exit.o
        50          0          0          0          0         88   defsig_general.o
        80         58          0          0          0         76   defsig_rtmem_inner.o
        14          0          0          0          0         80   defsig_rtmem_outer.o
        52         38          0          0          0         76   defsig_rtred_inner.o
        14          0          0          0          0         80   defsig_rtred_outer.o
        18          0          0          0          0         80   exit.o
        76          0          0          0          0         88   fclose.o
       470          0          0          0          0         88   flsbuf.o
       236          4          0          0          0        128   fopen.o
        26          0          0          0          0         68   fputc.o
       248          6          0          0          0         84   fseek.o
        66          0          0          0          0         76   ftell.o
        94          0          0          0          0         80   h1_alloc.o
        52          0          0          0          0         68   h1_extend.o
        78          0          0          0          0         80   h1_free.o
        14          0          0          0          0         84   h1_init.o
        80          6          0          4          0         96   heapauxa.o
         4          0          0          0          0        136   hguard.o
         0          0          0          0          0          0   indicate_semi.o
       138          0          0          0          0        168   init_alloc.o
       312         46          0          0          0        112   initio.o
         2          0          0          0          0          0   libinit.o
         6          0          0          0          0          0   libinit2.o
        16          8          0          0          0          0   libinit4.o
         2          0          0          0          0          0   libshutdown.o
         6          0          0          0          0          0   libshutdown2.o
         0          0          0          0         96          0   libspace.o
         0          0          0          0          0          0   maybetermalloc1.o
        44          4          0          0          0         84   puts.o
         8          4          0          0          0         68
rt_errno_addr_intlibspace.o
         8          4          0          0          0         68
rt_heap_descriptor_intlibspace.o
        78          0          0          0          0         80   rt_memclr_w.o
         2          0          0          0          0          0   rtexit.o
        10          0          0          0          0          0   rtexit2.o
        70          0          0          0          0         80   setvbuf.o
       240          6          0          0          0        156   stdio.o
         0          0          0         12        252          0   stdio_streams.o
        62          0          0          0          0         76   strlen.o
        12          4          0          0          0         68   sys_exit.o
       102          0          0          0          0        240   sys_io.o
```

```
         0          0         12          0          0          0   sys_io_names.o
        14          0          0          0          0         76   sys_wrch.o
         2          0          0          0          0         68   use_no_semi.o

    ----------------------------------------------------------------------
      2962        200         14         16        352       3036   Library Totals
        12          0          2          0          4          0   (incl. Padding)

    ----------------------------------------------------------------------

      Code (inc. data)   RO Data    RW Data    ZI Data      Debug   Library Name

      2950        200         12         16        348       3036   c_wu.l

    ----------------------------------------------------------------------
      2962        200         14         16        352       3036   Library Totals

    ----------------------------------------------------------------------

==============================================================================


      Code (inc. data)   RO Data    RW Data    ZI Data      Debug

      3050        226       1006         16       5472       1948   Grand Totals
      3050        226       1006         16       5472       1948   ELF Image Totals
      3050        226       1006         16          0          0   ROM Totals

==============================================================================

    Total RO  Size (Code + RO Data)                 4056 (   3.96kB)
    Total RW  Size (RW Data + ZI Data)              5488 (   5.36kB)
    Total ROM Size (Code + RO Data + RW Data)       4072 (   3.98kB)

==============================================================================
```

In this example:

Code (inc. data)
The number of bytes occupied by the code. In this image, there are 3050 bytes of code. This value includes 226 bytes of inline data (`inc. data`), for example, literal pools, and short strings.

RO Data
The number of bytes occupied by the RO data. This value is in addition to the inline data included in the `Code (inc. data)` column.

RW Data
The number of bytes occupied by the RW data.

ZI Data
The number of bytes occupied by the ZI data.

Debug
The number of bytes occupied by the debug data, for example, debug Input sections and the symbol and string table.

Object Totals
The number of bytes occupied by the objects when 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.

Combined across all of the object files (`foo.o` and `startup_ARMCM7.o`), the example shows that there are 992 bytes of RO data, of which 32 bytes are linker-generated RO data.

Note

If the scatter file contains `EMPTY` regions, the linker might generate ZI data. In the example, the 4096 bytes of ZI data labeled (`incl. Generated`) correspond to an `ARM_LIB_STACKHEAP` execution region used to set up the stack and heap in a scatter file as follows:

```
ARM_LIB_STACKHEAP +0x0 EMPTY 0x1000 {} ; 4KB stack + heap
```

Library Totals

The number of bytes occupied by the library members that have been extracted and added to the image as individual objects.

(incl. Padding)

If necessary, `armlink` inserts padding 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 992 bytes of RO data in the object total, of which 0 bytes is linker-generated padding, and 14 bytes of RO data in the library total, with 2 bytes of padding.

Grand Totals

Shows the true size of the image. In the example, there are 5120 bytes of ZI data (in `Object Totals`) and 352 of ZI data (in `Library Totals`) giving a total of 5472 bytes.

ELF Image Totals

If you are using RW data compression (the default) to optimize ROM size, the size of the final image changes. This change 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 size does not include ZI data and debug information that is not stored in the ROM.

Related reference

5.1 Options for getting information about linker-generated files on page 5-81
11.59 --info=topic[,topic,...] on page 11-281

5.4 How to find where a symbol is placed when linking

To find where a symbol is placed when linking you must find the section that defines the symbol, and ensure that the linker has not removed the section.

You can do this with the `--keep="section_id"` and `--symbols` options. For example, if `object(section)` is the section containing the symbol, enter:

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

Note

You can also run `fromelf -s` on the resultant image.

As an example, do the following:

Procedure

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

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

int main(void)
{
    return sizeof(array);
}
```

2. Compile the source:

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

3. Link the object `s.o`, keeping the `ARRAY` symbol and displaying the symbols:

```
armlink --cpu=8-A.32 --keep="s.o(ARRAY)" --map --symbols 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
```

```
...
Execution Region ER_RW (Base: 0x00008360, Size: 0x00000050, Max: 0xffffffff, ABSOLUTE)

Base Addr    Size         Type   Attr      Idx    E Section Name        Object

0x00008360   0x00000050   Data   RW            3    ARRAY               s.o
```

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

Related reference

11.66 --keep=section_id on page 11-290

11.85 --map, --no_map on page 11-314

11.92 -o filename, --output=filename on page 11-321

Related information

Using fromelf to find where a symbol is placed in an executable ELF image

-c compiler option

-march compiler option

-o compiler option

--target compiler option

Chapter 6
Accessing and Managing Symbols with armlink

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

It contains the following sections:

6.1 About mapping symbols

Mapping symbols are generated by the compiler and assembler to identify various inline transitions.

For Armv7-A, inline transitions can be between:

- Code and data at literal pool boundaries.
- Arm code and Thumb code, such as Arm and Thumb interworking veneers.

For Armv8-A, inline transitions can be between:

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

For Armv6-M, Armv7-M, and Armv8-M, inline transitions can be between code and data at literal pool boundaries.

The mapping symbols available for each architecture are:

Symbol	Description	Architecture
`$a`	Start of a sequence of Arm/A32 instructions.	All
`$t`	Start of a sequence of Thumb/T32 instructions.	All
`$t.x`	Start of a sequence of ThumbEE instructions.	Armv7-A
`$d`	Start of a sequence of data items, such as a literal pool.	All
`$x`	Start of A64 code.	Armv8-A

`armlink` generates the `$d.realdata` mapping symbol to communicate to `fromelf` that the data is from a non-executable section. Therefore, the code and data sizes output by `fromelf -z` are the same as the output from `armlink --info sizes`, for example:

```
    Code (inc. data)   RO Data
       x          y          z
```

In this example, the `y` is marked with `$d`, and `RO Data` is marked with `$d.realdata`.

Note

Symbols beginning with the characters `$v` are mapping symbols related to VFP and might be output when building for a target with VFP. Avoid using symbols beginning with `$v` in your source code.

Be aware that modifying an executable image with the `fromelf --elf --strip=localsymbols` command removes all mapping symbols from the image.

Related reference

11.76 --list_mapping_symbols, --no_list_mapping_symbols on page 11-303

11.135 --strict_symbols, --no_strict_symbols on page 11-367

Related information

Symbol naming rules

--strip=option[,option,...] fromelf option

--text fromelf option

ELF for the Arm Architecture

6.2 Linker-defined symbols

The linker defines some symbols that are reserved by Arm, and that you can access if required.

Symbols that contain the character sequence `$$`, and all other external names containing the sequence `$$`, are names reserved by Arm.

You can import these symbolic addresses and use them as relocatable addresses by your assembly language programs, or refer to them as **`extern`** symbols from your C or C++ source code.

Be aware that:

- Linker-defined symbols are only generated when your code references them.
- If *execute-only* (XO) sections are present, linker-defined symbols are defined with the following constraints:
  - XO linker defined symbols cannot be defined with respect to an empty region or a region that has no XO sections.
  - XO linker defined symbols cannot be defined with respect to a region that contains only RO sections.
  - RO linker defined symbols cannot be defined with respect to a region that contains only XO sections.

Note

XO memory is supported only for Armv7-M and Armv8-M architectures.

Related concepts

6.3.7 Methods of importing linker-defined symbols in C and C++ on page 6-93
6.3.8 Methods of importing linker-defined symbols in Arm® assembly language on page 6-94

6.3 Region-related symbols

The linker generates various types of region-related symbols that you can access if required.

This section contains the following subsections:

6.3.1 Types of region-related symbols

The linker generates the different types of region-related symbols for each region in the image.

The types are:

- `Image$$` and `Load$$` for each execution region.
- `Load$$LR$$` for each load region.

If you are using a scatter file these symbols are generated for each region in the scatter file.

If you are not using scatter-loading, the symbols are generated for the default region names. That is, the region names are fixed and the same types of symbol are supplied.

Related concepts

6.3.5 Region name values when not scatter-loading on page 6-93

Related reference

6.3.2 Image$$ execution region symbols on page 6-90

6.3.3 Load$$ execution region symbols on page 6-91

6.3.4 Load$$LR$$ load region symbols on page 6-92

6.3.2 Image$$ execution region symbols

The linker generates `Image$$` symbols for every execution region present in the image.

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

Table 6-1 Image$$ execution region symbols

Symbol	Description
`Image$$`*region_name*`$$Base`	Execution address of the region.
`Image$$`*region_name*`$$Length`	Execution region length in bytes excluding ZI length.
`Image$$`*region_name*`$$Limit`	Address of the byte beyond the end of the non-ZI part of the execution region.
`Image$$`*region_name*`$$RO$$Base`	Execution address of the RO output section in this region.
`Image$$`*region_name*`$$RO$$Length`	Length of the RO output section in bytes.
`Image$$`*region_name*`$$RO$$Limit`	Address of the byte beyond the end of the RO output section in the execution region.
`Image$$`*region_name*`$$RW$$Base`	Execution address of the RW output section in this region.
`Image$$`*region_name*`$$RW$$Length`	Length of the RW output section in bytes.

Table 6-1 Image$$ execution region symbols (continued)

Symbol	Description
`Image$$`*region_name*`$$RW$$Limit`	Address of the byte beyond the end of the RW output section in the execution region.
`Image$$`*region_name*`$$XO$$Base`	Execution address of the XO output section in this region.
`Image$$`*region_name*`$$XO$$Length`	Length of the XO output section in bytes.
`Image$$`*region_name*`$$XO$$Limit`	Address of the byte beyond the end of the XO output section in the execution region.
`Image$$`*region_name*`$$ZI$$Base`	Execution address of the ZI output section in this region.
`Image$$`*region_name*`$$ZI$$Length`	Length of the ZI output section in bytes.
`Image$$`*region_name*`$$ZI$$Limit`	Address of the byte beyond the end of the ZI output section in the execution region.

Related concepts

6.3.1 Types of region-related symbols on page 6-90

6.3.3 Load$$ execution region symbols

The linker generates `Load$$` symbols for every execution region present in the image.

Note

`Load$$`*region_name* symbols apply only to execution regions. `Load$$LR$$`*load_region_name* symbols apply only to load regions.

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

Table 6-2 Load$$ execution region symbols

Symbol	Description
`Load$$`*region_name*`$$Base`	Load address of the region.
`Load$$`*region_name*`$$Length`	Region length in bytes.
`Load$$`*region_name*`$$Limit`	Address of the byte beyond the end of the execution region.
`Load$$`*region_name*`$$RO$$Base`	Address of the RO output section in this execution region.
`Load$$`*region_name*`$$RO$$Length`	Length of the RO output section in bytes.
`Load$$`*region_name*`$$RO$$Limit`	Address of the byte beyond the end of the RO output section in the execution region.
`Load$$`*region_name*`$$RW$$Base`	Address of the RW output section in this execution region.
`Load$$`*region_name*`$$RW$$Length`	Length of the RW output section in bytes.
`Load$$`*region_name*`$$RW$$Limit`	Address of the byte beyond the end of the RW output section in the execution region.
`Load$$`*region_name*`$$XO$$Base`	Address of the XO output section in this execution region.
`Load$$`*region_name*`$$XO$$Length`	Length of the XO output section in bytes.
`Load$$`*region_name*`$$XO$$Limit`	Address of the byte beyond the end of the XO output section in the execution region.
`Load$$`*region_name*`$$ZI$$Base`	Load address of the ZI output section in this execution region.

Table 6-2 Load$$ execution region symbols (continued)

Symbol	Description
`Load$$`*`region_name`*`$$ZI$$Length`	Load length of the ZI output section in bytes. The Load Length of ZI is zero unless *`region_name`* has the `ZEROPAD` scatter-loading keyword set.
`Load$$`*`region_name`*`$$ZI$$Limit`	Load address of the byte beyond the end of the ZI output section in the execution region.

All symbols in this table refer to load addresses before the C library is initialized. Be aware of the following:

- The symbols are absolute because section-relative symbols can only have execution addresses.
- The symbols take into account RW compression.
- References to linker-defined symbols from RW compressed execution regions must be to symbols that are resolvable before RW compression is applied.
- If the linker detects a relocation from an RW-compressed region to a linker-defined symbol that depends on RW compression, then the linker disables compression for that region.
- Any zero bytes written to the file are visible. Therefore, the Limit and Length values must take into account the zero bytes written into the file.

Related concepts

6.3.1 Types of region-related symbols on page 6-90

6.3.7 Methods of importing linker-defined symbols in C and C++ on page 6-93

6.3.8 Methods of importing linker-defined symbols in Arm® assembly language on page 6-94

6.3.5 Region name values when not scatter-loading on page 6-93

4.3 Optimization with RW data compression on page 4-68

Related reference

6.3.2 Image$$ execution region symbols on page 6-90

6.3.4 Load$$LR$$ load region symbols on page 6-92

8.4.3 Execution region attributes on page 8-175

6.3.4 Load$$LR$$ load region symbols

The linker generates `Load$$LR$$` symbols for every load region present in the image.

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

Note

`Load$$LR$$`*`load_region_name`* symbols apply only to load regions. `Load$$`*`region_name`* symbols apply only to execution regions.

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

Table 6-3 Load$$LR$$ load region symbols

Symbol	Description
`Load$$LR$$`*`load_region_name`*`$$Base`	Address of the load region.
`Load$$LR$$`*`load_region_name`*`$$Length`	Length of the load region.
`Load$$LR$$`*`load_region_name`*`$$Limit`	Address of the byte beyond the end of the load region.

Related concepts

6.3.1 Types of region-related symbols on page 6-90

3.1 The structure of an Arm® ELF image on page 3-33

3.1.2 Input sections, output sections, regions, and program segments on page 3-34

3.1.3 Load view and execution view of an image on page 3-35

6.3.5 Region name values when not scatter-loading

When scatter-loading is not used when linking, the linker uses default region name values.

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

- `ER_XO`, for an execute-only execution region, if present.
- `ER_RO`, for the read-only execution region.
- `ER_RW`, for the read-write execution region.
- `ER_ZI`, for the zero-initialized execution region.

You can insert these names into the following symbols to obtain the required address:

- `Image$$` execution region symbols.
- `Load$$` execution region symbols.

For example, `Load$$ER_RO$$Base`.

Related concepts

6.3.1 Types of region-related symbols on page 6-90

6.4 Section-related symbols on page 6-95

Related reference

6.3.2 Image$$ execution region symbols on page 6-90

6.3.3 Load$$ execution region symbols on page 6-91

6.3.6 Linker defined symbols and scatter files

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

The scatter file:

- Names all the load and execution regions in the image, and provides their load and execution addresses.
- Defines both stack and heap. The linker also generates special stack and heap symbols.

Related reference

Chapter 7 Scatter-loading Features on page 7-104

11.120 --scatter=filename on page 11-350

6.3.7 Methods of importing linker-defined symbols in C and C++

You can import linker-defined symbols into your C or C++ source code. They are external symbols and you must take the address of them.

The only case where the `&` operator is not required is when the array declaration is used, for example `extern char symbol_name[];`.

The following examples show how to obtain the correct value:

Importing a linker-defined symbol

```
extern int Image$$ER_ZI$$Limit;
heap_base = (uintptr_t)&Image$$ER_ZI$$Limit;
```

Importing symbols that define a ZI output section

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

Related reference

6.3.2 Image$$ execution region symbols on page 6-90

6.3.8 Methods of importing linker-defined symbols in Arm® assembly language

You can import linker-defined symbols into your Arm assembly code.

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

32-bit applications

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

```
        .global Image$$ER_ZI$$Limit
        …
.zi_limit:
        .word Image$$ER_ZI$$Limit
```

To load the value into a register, such as r1, use the `LDR` instruction:

```
        LDR r1, .zi_limit
```

The `LDR` instruction must be able to reach the 32-bit data word. The accessible memory range varies between A64, A32, and T32, and the architecture you are using.

64-bit applications

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

```
        .global Image$$ER_ZI$$Limit
        …
.zi_limit:
        .quad Image$$ER_ZI$$Limit
```

To load the value into a register, such as x1, use the `LDR` instruction:

```
        LDR x1, .zi_limit
```

The `LDR` instruction must be able to reach the 64-bit data word.

Related reference

6.3.2 Image$$ execution region symbols on page 6-90

Related information

A32 and T32 Instructions

IMPORT and EXTERN

6.4 Section-related symbols

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

This section contains the following subsections:

6.4.1 Types of section-related symbols

The linker generates different types of section-related symbols for output and input sections.

The types of symbols are:

- Image symbols, if you do not use scatter-loading to create a simple image. A simple image has up to four output sections (XO, RO, RW, and ZI) that produce the corresponding execution regions.
- Input section symbols, for every input section present in the image.

The linker sorts sections within an execution region first by attribute RO, RW, or ZI, then by name. So, for example, all `.text` sections are placed in one contiguous block. A contiguous block of sections with the same attribute and name is known as a *consolidated section*.

Related reference

6.4.2 Image symbols on page 6-95

6.4.3 Input section symbols on page 6-96

6.4.2 Image symbols

Image symbols are generated by the linker when you do not use scatter-loading to create a simple image.

The following table shows the image symbols:

Table 6-4 Image symbols

Symbol	Section type	Description
`Image$$RO$$Base`	Output	Address of the start of the RO output section.
`Image$$RO$$Limit`	Output	Address of the first byte beyond the end of the RO output section.
`Image$$RW$$Base`	Output	Address of the start of the RW output section.
`Image$$RW$$Limit`	Output	Address of the byte beyond the end of the ZI output section. (The choice of the end of the ZI region rather than the end of the RW region is to maintain compatibility with legacy code.)
`Image$$ZI$$Base`	Output	Address of the start of the ZI output section.
`Image$$ZI$$Limit`	Output	Address of the byte beyond the end of the ZI output section.

Note

- Arm recommends that you use region-related symbols in preference to section-related symbols.
- The ZI output sections of an image are not created statically, but are automatically created dynamically at runtime.
- There are no load address symbols for RO, RW, and ZI output sections.

If you are using a scatter file, the image symbols are undefined. If your code accesses any of these symbols, you must treat them as a weak reference.

The standard implementation of `__user_setup_stackheap()` uses the value in `Image$$ZI$$Limit`. Therefore, if you are using a scatter file you must manually place the stack and heap. You can do this either:

- In a scatter file using one of the following methods:
  - Define separate stack and heap regions called `ARM_LIB_STACK` and `ARM_LIB_HEAP`.
  - Define a combined region containing both stack and heap called `ARM_LIB_STACKHEAP`.
- By re-implementing `__user_setup_stackheap()` to set the heap and stack boundaries.

Related concepts

3.2 Simple images on page 3-41

3.8 Weak references and definitions on page 3-58

Related tasks

7.1.4 Placing the stack and heap with a scatter file on page 7-106

Related reference

7.1.3 Linker-defined symbols that are not defined when scatter-loading on page 7-106

Related information

__user_setup_stackheap()

6.4.3 Input section symbols

Input section symbols are generated by the linker for every input section present in the image.

The following table shows the input section symbols:

Table 6-5 Section-related symbols

Symbol	Section type	Description
SectionName`$$Base`	Input	Address of the start of the consolidated section called *`SectionName`*.
SectionName`$$Length`	Input	Length of the consolidated section called *`SectionName`* (in bytes).
SectionName`$$Limit`	Input	Address of the byte beyond the end of the consolidated section called *`SectionName`*.

If your code refers to the input-section symbols, it is assumed that you expect all the input sections in the image with the same name to be placed contiguously in the image memory map.

If your scatter file places input sections non-contiguously, the linker issues an error. This is because the use of the base and limit symbols over non-contiguous memory is ambiguous.

Related concepts

3.1.2 Input sections, output sections, regions, and program segments on page 3-34

Related reference

Chapter 7 Scatter-loading Features on page 7-104

6.5 Access symbols in another image

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

This section contains the following subsections:

6.5.1 Creating a symdefs file

You can specify a symdefs file on the linker command-line.

You can use a symdefs file, for example, if you have one image that always resides in ROM and multiple images that are loaded into RAM. The images loaded into RAM can access global functions and data from the image located in ROM.

Use the `armlink` option `--symdefs=`*`filename`* to generate a symdefs file.

The linker produces a symdefs file during a successful final link stage. It is not produced for partial linking or for unsuccessful final linking.

Note

If *`filename`* does not exist, the linker creates the file and adds entries for all the global symbols to that file. If *`filename`* exists, the linker uses the existing contents of *`filename`* to select the symbols that are output when it rewrites the file. This means that only the existing symbols in the filename are updated, and no new symbols (if any) are added at all. If you do not want this behavior, ensure that any existing symdefs file is deleted before the link step.

Related tasks

6.5.2 Outputting a subset of the global symbols on page 6-97

6.5.3 Reading a symdefs file on page 6-98

Related reference

6.5.4 Symdefs file format on page 6-98

11.138 --symdefs=filename on page 11-370

6.5.2 Outputting a subset of the global symbols

You can use a symdefs file to output a subset of the global symbols to another application.

By default, all global symbols are written to the symdefs file. When a symdefs file exists, the linker uses its contents to restrict the output to a subset of the global symbols.

This example uses an application `image1` containing symbols that you want to expose to another application using a symdefs file.

Procedure

1. Specify `--symdefs=`*`filename`* when you are doing a final link for `image1`. The linker creates a symdefs file *`filename`*.
2. Open *`filename`* in a text editor, remove any symbol entries you do not want in the final list, and save the file.
3. Specify `--symdefs=`*`filename`* when you are doing a final link for `image1`.

   You can edit *`filename`* at any time to add comments and link `image1` again. For example, to update the symbol definitions to create `image1` after one or more objects have changed.

You can use the symdefs file to link additional applications.

Related concepts

6.5 Access symbols in another image on page 6-97

Related tasks

6.5.1 Creating a symdefs file on page 6-97

Related reference

6.5.4 Symdefs file format on page 6-98

11.138 --symdefs=filename on page 11-370

6.5.3 Reading a symdefs file

A symdefs file can be considered as an object file with symbol information but no code or data.

To read a symdefs file, add it to your file list as you do for any object file. The linker reads the file and adds the symbols and their values to the output symbol table. The added symbols have `ABSOLUTE` and `GLOBAL` attributes.

If a partial link is being performed, the symbols are added to the output object symbol table. If a full link is being performed, the symbols are added to the image symbol table.

The linker generates error messages for invalid rows in the file. A row is invalid if:

- Any of the columns are missing.
- Any of the columns have invalid values.

The symbols extracted from a symdefs file are treated in exactly the same way as symbols extracted from an object symbol table. The same restrictions apply regarding multiple symbol definitions.

Note

The same function name or symbol name cannot be defined in both A32 code and in T32 code.

Related reference

6.5.4 Symdefs file format on page 6-98

6.5.4 Symdefs file format

A symdefs file defines symbols and their values.

The file consists of:

Identification line

The identification line in a symdefs file comprises:

- An identifying string, `#<SYMDEFS>#`, which must be the first 11 characters in the file for the linker to recognize it as a symdefs file.
- Linker version information, in the format:

  `ARM Linker, vvvvbbb:`
- Date and time of the most recent update of the symdefs file, in the format:

  `Last Updated: day month date hh:mm:ss year`

For example, for version 6.3, build 169:

```
#<SYMDEFS># ARM Linker, 6030169: Last Updated: Thu Jun 4 12:49:45 2015
```

The version and update information are not part of the identifying string.

Comments

You can insert comments manually with a text editor. Comments have the following properties:

- The first line must start with the special identifying comment `#<SYMDEFS>#`. This comment is inserted by the linker when the file is produced and must not be manually deleted.
- Any line where the first non-whitespace character is a semicolon (`;`) or hash (`#`) is a comment.
- A semicolon (`;`) or hash (`#`) after the first non-whitespace character does not start a comment.
- Blank lines are ignored and can be inserted to improve readability.

Symbol information

The symbol information is provided on a single line, and comprises:

Symbol value

The linker writes the absolute address of the symbol in fixed hexadecimal format, for example, `0x00008000`. If you edit the file, you can use either hexadecimal or decimal formats for the address value.

Type flag

A single letter to show symbol type:

X
A64 code (AArch64 only)

A
A32 code (AArch32 only)

T
T32 code (AArch32 only)

D
Data

N
Number.

Symbol name
The symbol name.

Example

This example shows a typical symdefs file format:

```
#<SYMDEFS># ARM Linker, 6030169: Last Updated: Date
;value type name, this is an added comment
0x00008000 A __main
0x00008004 A __scatterload
0x000080E0 T main
0x0000814D T _main_arg
0x0000814D T __argv_alloc
0x00008199 T __rt_get_argv
…
   # This is also a comment, blank lines are ignored
…
0x0000A4FC D __stdin
0x0000A540 D __stdout
0x0000A584 D __stderr
0xFFFFFFFD N __SIG_IGN
```

Related tasks

6.5.3 Reading a symdefs file on page 6-98
6.5.1 Creating a symdefs file on page 6-97

6.6 Edit the symbol tables with a steering file

A steering file is a text file that contains a set of commands to edit the symbol tables of output objects and the dynamic sections of images.

This section contains the following subsections:

6.6.1 Specifying steering files on the linker command-line

You can specify one or more steering files on the linker command-line.

Use the option `--edit file-list` to specify one or more steering files on the linker command-line.

When you specify more than one steering file, you can use either of the following command-line formats:

```
armlink --edit file1 --edit file2 --edit file3
```

```
armlink --edit file1,file2,file3
```

Do not include spaces between the comma and the filenames when using a comma-separated list.

Related reference

6.6.2 Steering file command summary on page 6-100

6.6.3 Steering file format on page 6-101

6.6.2 Steering file command summary

Steering file commands enable you to manage symbols in the symbol table, control the copying of symbols from the static symbol table to the dynamic symbol table, and store information about the libraries that a link unit depends on.

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

The steering file commands are:

Table 6-6 Steering file command summary

Command	Description
`EXPORT`	Specifies that a symbol can be accessed by other shared objects or executables.
`HIDE`	Makes defined global symbols in the symbol table anonymous.
`IMPORT`	Specifies that a symbol is defined in a shared object at runtime.
`RENAME`	Renames defined and undefined global symbol names.
`REQUIRE`	Creates a `DT_NEEDED` tag in the dynamic array. `DT_NEEDED` tags specify dependencies to other shared objects used by the application, for example, a shared library.
`RESOLVE`	Matches specific undefined references to a defined global symbol.
`SHOW`	Makes global symbols visible. This command is useful if you want to make a specific symbol visible that is hidden using a `HIDE` command with a wildcard.

Note

The steering file commands control only global symbols. Local symbols are not affected by any of these commands.

Related tasks

6.6.1 Specifying steering files on the linker command-line on page 6-100

Related reference

6.6.3 Steering file format on page 6-101

11.36 --edit=file_list on page 11-258

12.1 EXPORT steering file command on page 12-395

12.2 HIDE steering file command on page 12-396

12.3 IMPORT steering file command on page 12-397

12.4 RENAME steering file command on page 12-398

12.5 REQUIRE steering file command on page 12-399

12.6 RESOLVE steering file command on page 12-400

12.7 SHOW steering file command on page 12-402

6.6.3 Steering file format

Each command in a steering file must be on a separate line.

A steering file has the following format:

- Lines with a semicolon (`;`) or hash (`#`) character as the first non-whitespace character are interpreted as comments. A comment is treated as a blank line.
- Blank lines are ignored.
- Each non-blank, non-comment line is either a command, or part of a command that is split over consecutive non-blank lines.
- Command lines that end with a comma (`,`) as the last non-whitespace character are continued on the next non-blank line.

Each command line consists of a command, followed by one or more comma-separated operand groups. Each operand group comprises either one or two operands, depending on the command. The command is applied to each operand group in the command. The following rules apply:

- Commands are case-insensitive, but are conventionally shown in uppercase.
- Operands are case-sensitive because they must be matched against case-sensitive symbol names. You can use wildcard characters in operands.

Commands are applied to global symbols only. Other symbols, such as local symbols, are not affected.

The following example shows a sample steering file:

```
; Import my_func1 as func1
IMPORT my_func1 AS func1
# Rename a very long function name to a shorter name
RENAME a_very_long_function_name AS,
       short_func_name
```

Related tasks

6.6.1 Specifying steering files on the linker command-line on page 6-100

Related reference

6.6.2 Steering file command summary on page 6-100

12.1 EXPORT steering file command on page 12-395

12.2 HIDE steering file command on page 12-396

12.3 IMPORT steering file command on page 12-397

12.4 RENAME steering file command on page 12-398

12.5 REQUIRE steering file command on page 12-399

12.6 RESOLVE steering file command on page 12-400
12.7 SHOW steering file command on page 12-402

6.6.4 Hide and rename global symbols with a steering file

You can use a steering file to hide and rename global symbol names in output files.

Use the `HIDE` and `RENAME` commands as required.

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

Example of renaming a symbol:

RENAME steering command example

```
RENAME func1 AS my_func1
```

Example of hiding symbols:

HIDE steering command example

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

Related concepts

6.6 Edit the symbol tables with a steering file on page 6-100

Related tasks

6.6.1 Specifying steering files on the linker command-line on page 6-100

Related reference

6.6.2 Steering file command summary on page 6-100
6.5.4 Symdefs file format on page 6-98
12.2 HIDE steering file command on page 12-396
12.4 RENAME steering file command on page 12-398
11.36 --edit=file_list on page 11-258

6.7 Use of $Super$$ and $Sub$$ to patch symbol definitions

There are special patterns that you can use for situations where an existing symbol cannot be modified or recompiled.

An existing symbol cannot be modified if, for example, it is located in an external library or in ROM code. In such cases, you can use the `$Super$$` and `$Sub$$` patterns to patch an existing symbol.

To patch the definition of the function `foo()`, `$Sub$$foo` and the original definition of `foo()` must be a global or weak definition:

`$Super$$foo`

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

`$Sub$$foo`

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

The `$Sub$$` and `$Super$$` linker mechanism can operate only on symbol definitions and references that are visible to the tool. For example, the compiler can replace a call to `printf("Hello\n")` with `puts("Hello")` in a C program. Only the reference to the symbol `puts` is visible to the linker, so defining `$Sub$$printf` will not redirect this call.

Note

- The `$Sub$$` and `$Super$$` mechanism only works at static link time, `$Super$$` references cannot be imported or exported into the dynamic symbol table.
- If the compiler inlines a function, for example `foo()`, then it is not possible to patch the inlined function with the substitute function, `$Sub$$foo`.

Example

The following example shows how to use `$Super$$` and `$Sub$$` to insert a call to the function `ExtraFunc()` before the call to the legacy function `foo()`.

```
extern void ExtraFunc(void);
extern void $Super$$foo(void);

/* this function is called instead of the original foo() */
void $Sub$$foo(void)
{
  ExtraFunc();    /* does some extra setup work */
  $Super$$foo();  /* calls the original foo() function */
                  /* To avoid calling the original foo() function
                   * omit the $Super$$foo(); function call.
                   */
}
```

Related information

ELF for the Arm Architecture

Chapter 7
Scatter-loading Features

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

It contains the following sections:

7.1 The scatter-loading mechanism

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

This section contains the following subsections:

7.1.1 Overview of scatter-loading

Scatter-loading gives you complete control over the grouping and placement of image components.

You can use scatter-loading to create simple images, but it is generally only used for images that have a complex memory map. That is, where multiple memory regions are scattered in the memory map at load and execution time.

An image memory map is made up of regions and output sections. Every region in the memory map can have a different load and execution address.

To construct the memory map of an image, the linker must have:

- Grouping information that describes how input sections are grouped into output sections and regions.
- Placement information that describes the addresses where regions are to be located in the memory maps.

When the linker creates an image using a scatter file, it creates some region-related symbols. The linker creates these special symbols only if your code references them.

Related concepts

7.1.2 When to use scatter-loading on page 7-105

7.16 Scatter file to ELF mapping on page 7-162

3.1 The structure of an Arm® ELF image on page 3-33

Related reference

6.3 Region-related symbols on page 6-90

7.1.2 When to use scatter-loading

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

Situations where scatter-loading is either required or very useful:

Complex memory maps

Code and data that must be placed into many distinct areas of memory require detailed instructions on where to place the sections in the memory space.

Different types of memory

Many systems contain a variety of physical memory devices such as flash, ROM, SDRAM, and fast SRAM. A scatter-loading description can match the code and data with the most appropriate type of memory. For example, interrupt code might be placed into fast SRAM to improve interrupt response time but infrequently-used configuration information might be placed into slower flash memory.

Memory-mapped peripherals

The scatter-loading description can place a data section at a precise address in the memory map so that memory mapped peripherals can be accessed.

Functions at a constant location

A function can be placed at the same location in memory even though the surrounding application has been modified and recompiled. This is useful for jump table implementation.

Using symbols to identify the heap and stack

Symbols can be defined for the heap and stack location when the application is linked.

Related concepts

7.1.1 Overview of scatter-loading on page 7-105

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

When scatter-loading an image, some linker-defined symbols are undefined.

The following symbols are undefined when a scatter file is used:

- `Image$$RO$$Base`.
- `Image$$RO$$Limit`.
- `Image$$RW$$Base`.
- `Image$$RW$$Limit`.
- `Image$$XO$$Base`.
- `Image$$XO$$Limit`.
- `Image$$ZI$$Base`.
- `Image$$ZI$$Limit`.

If you use a scatter file but do not use the special region names for stack and heap, or do not re-implement `__user_setup_stackheap()`, an error message is generated.

Related concepts

6.2 Linker-defined symbols on page 6-89

Related tasks

7.1.4 Placing the stack and heap with a scatter file on page 7-106

7.1.4 Placing the stack and heap with a scatter file

The Arm C library provides multiple implementations of the function `__user_setup_stackheap()`, and can select the correct one for you automatically from information that is given in a scatter file.

Note

- If you re-implement `__user_setup_stackheap()`, your version does not get invoked when stack and heap are defined in a scatter file.
- You might have to update your startup code to use the correct initial stack pointer. Some processors, such as the Cortex-M3 processor, require that you place the initial stack pointer in the vector table. See *Stack and heap configuration* in *AN179 - Cortex®-M3 Embedded Software Development* for more details.

Procedure

1. Define two special execution regions in your scatter file that are named `ARM_LIB_HEAP` and `ARM_LIB_STACK`.
2. Assign the `EMPTY` attribute to both regions.

   Because the stack and heap are in separate regions, the library selects the non-default implementation of `__user_setup_stackheap()` that uses the value of the symbols:

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

You can specify only one `ARM_LIB_STACK` or `ARM_LIB_HEAP` region, and you must allocate a size.

Example:

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

3. Alternatively, define a single execution region that is named `ARM_LIB_STACKHEAP` to use a combined stack and heap region. Assign the `EMPTY` attribute to the region.

   Because the stack and heap are in the same region, `__user_setup_stackheap()` uses the value of the symbols `Image$$ARM_LIB_STACKHEAP$$ZI$$Base` and `Image$$ARM_LIB_STACKHEAP$$ZI$$Limit`.

Related reference

6.3 Region-related symbols on page 6-90

Related information

__user_setup_stackheap()

7.1.5 Scatter-loading command-line options

The command-line options to the linker give some control over the placement of data and code, but complete control of placement requires more detailed instructions than can be entered on the command line.

Complex memory maps

Placement of code and data in complex memory maps must be specified in a scatter file. You specify the scatter file with the option:

`--scatter=`*`scatter_file`*

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

You can use `--scatter` with the `--base_platform` linking model.

Simple memory maps

For simple memory maps, you can place code and data with with the following memory map related command-line options:

- `--bpabi`.
- `--dll`.
- `--partial`.
- `--ro_base`.
- `--rw_base`.
- `--ropi`.
- `--rwpi`.
- `--rosplit`.
- `--split`.
- `--reloc`.
- `--xo_base`
- `--zi_base`.

Note

Apart from `--dll`, you cannot use `--scatter` with these options.

Related concepts

2.5 Base Platform linking model on page 2-30
7.1 The scatter-loading mechanism on page 7-105
7.1.2 When to use scatter-loading on page 7-105
7.13 Equivalent scatter-loading descriptions for simple images on page 7-152

Related reference

11.7 --base_platform on page 11-225
11.11 --bpabi on page 11-230
11.32 --dll on page 11-254
11.100 --partial on page 11-329
11.111 --reloc on page 11-341
11.114 --ro_base=address on page 11-344
11.115 --ropi on page 11-345
11.116 --rosplit on page 11-346
11.117 --rw_base=address on page 11-347
11.118 --rwpi on page 11-348
11.120 --scatter=filename on page 11-350
11.127 --split on page 11-359
11.157 --xo_base=address on page 11-389
11.161 --zi_base=address on page 11-393
Chapter 8 Scatter File Syntax on page 8-164

7.1.6 Scatter-loading images with a simple memory map

For images with a simple memory map, you can specify the memory map using only linker command-line options, or with a scatter file.

The following figure shows a simple memory map:

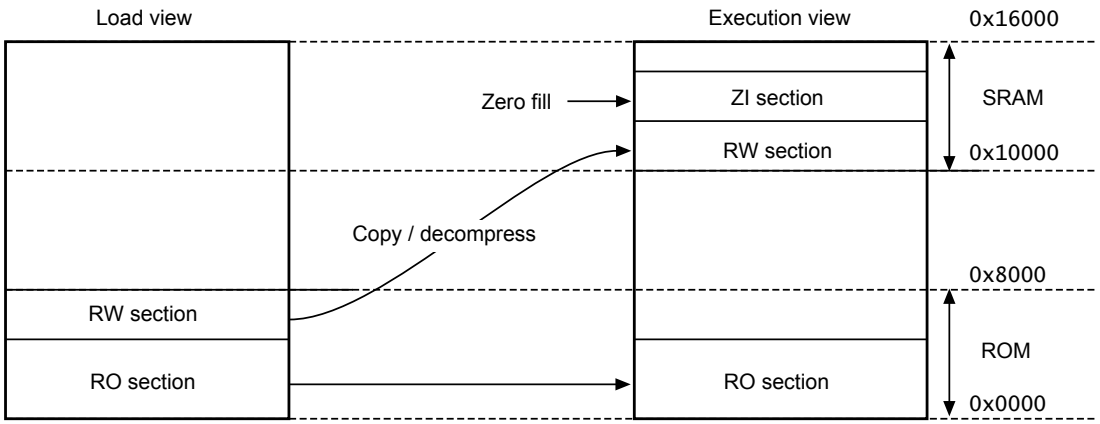

Figure 7-1 Simple scatter-loaded memory map

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

```
LOAD_ROM 0x0000 0x8000       ; Name of load region (LOAD_ROM),
                             ; Start address for load region (0x0000),
                             ; Maximum size of load region (0x8000)
{
    EXEC_ROM 0x0000 0x8000   ; Name of first exec region (EXEC_ROM),
```

```
                            ; Start address for exec region (0x0000),
                            ; Maximum size of first exec region (0x8000)
    {
        * (+RO)             ; Place all code and RO data into
                            ; this exec region
    }
    SRAM 0x10000 0x6000     ; Name of second exec region (SRAM),
                            ; Start address of second exec region (0x10000),
                            ; Maximum size of second exec region (0x6000)
    {
        * (+RW, +ZI)        ; Place all RW and ZI data into
                            ; this exec region
    }
}
```

The maximum size specifications for the regions are optional. However, if you include them, they enable the linker to check that a region does not overflow its boundary.

Apart from the limit checking, you can achieve the same result with the following linker command-line:

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

Related concepts

7.16 Scatter file to ELF mapping on page 7-162

7.1 The scatter-loading mechanism on page 7-105

7.1.2 When to use scatter-loading on page 7-105

Related reference

11.114 --ro_base=address on page 11-344

11.117 --rw_base=address on page 11-347

11.157 --xo_base=address on page 11-389

7.1.7 Scatter-loading images with a complex memory map

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

The following figure shows a complex memory map:

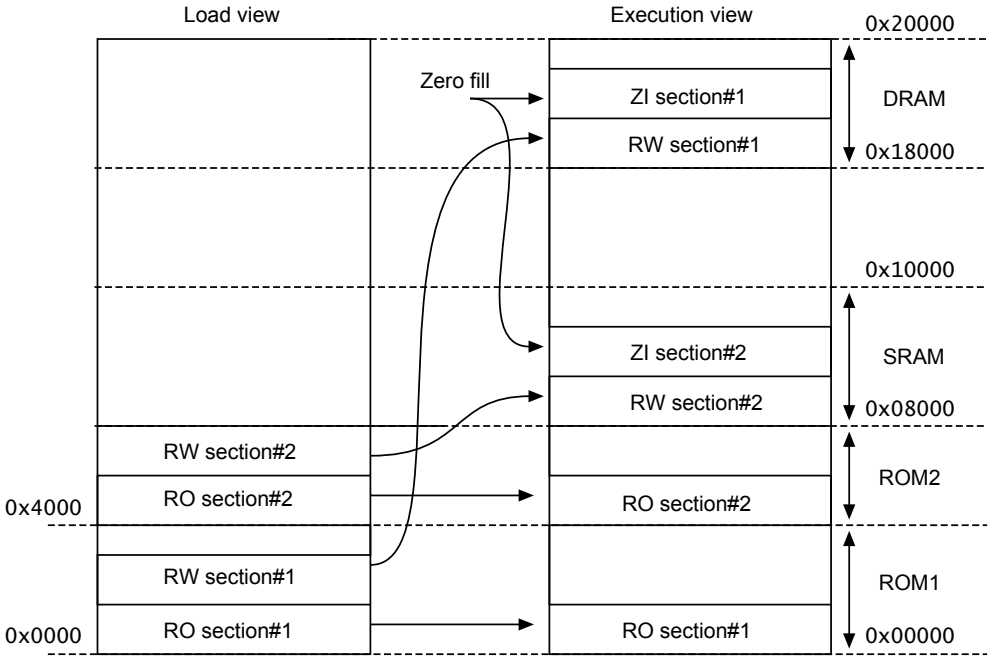

Figure 7-2 Complex memory map

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

```
LOAD_ROM_1 0x0000               ; Start address for first load region (0x0000)
{
    EXEC_ROM_1 0x0000           ; Start address for first exec region (0x0000)
    {
        program1.o (+RO)        ; Place all code and RO data from
                                ; program1.o into this exec region
    }
    DRAM 0x18000 0x8000         ; Start address for this exec region (0x18000),
                                ; Maximum size of this exec region (0x8000)
    {
        program1.o (+RW, +ZI)   ; Place all RW and ZI data from
                                ; program1.o into this exec region
    }
}
LOAD_ROM_2 0x4000               ; Start address for second load region (0x4000)
{
    EXEC_ROM_2 0x4000
    {
        program2.o (+RO)        ; Place all code and RO data from
                                ; program2.o into this exec region
    }
    SRAM 0x8000 0x8000
    {
        program2.o (+RW, +ZI)   ; Place all RW and ZI data from
                                ; program2.o into this exec region
    }
}
```

Caution

The scatter-loading description in this example specifies the location for code and data for `program1.o` and `program2.o` only. If you link an additional module, for example, `program3.o`, and use this description file, the location of the code and data for `program3.o` is not specified.

Unless you want to be very rigorous in the placement of code and data, Arm recommends that you use the `*` or `.ANY` specifier to place leftover code and data.

Related concepts

7.1 The scatter-loading mechanism on page 7-105

7.2.1 Effect of the ABSOLUTE attribute on a root region on page 7-111

7.2.2 Effect of the FIXED attribute on a root region on page 7-113

8.6.10 Scatter files containing relative base address load regions and a ZI execution region on page 8-192

7.16 Scatter file to ELF mapping on page 7-162

7.1.2 When to use scatter-loading on page 7-105

7.2 Root region and the initial entry point

The initial entry point of the image must be in a root region.

If the initial entry point is not in a root region, the link fails and the linker gives an error message.

Example

Root region with the same load and execution address.

```
LR_1 0x040000          ; load region starts at 0x40000
{                      ; start of execution region descriptions
    ER_RO 0x040000     ; load address = execution address
    {
        * (+RO)        ; all RO sections (must include section with
                       ; initial entry point)
    }
    …                 ; rest of scatter-loading description
}
```

This section contains the following subsections:

- *7.2.1 Effect of the ABSOLUTE attribute on a root region* on page 7-111.
- *7.2.2 Effect of the FIXED attribute on a root region* on page 7-113.
- *7.2.3 Methods of placing functions and data at specific addresses* on page 7-114.
- *7.2.4 Placing functions and data in a named section* on page 7-118.
- *7.2.5 Placing __at sections at a specific address* on page 7-120.
- *7.2.6 Restrictions on placing __at sections* on page 7-121.
- *7.2.7 Automatically placing __at sections* on page 7-121.
- *7.2.8 Manually placing __at sections* on page 7-123.
- *7.2.9 Placing a key in flash memory with an __at section* on page 7-124.

7.2.1 Effect of the ABSOLUTE attribute on a root region

You can use the `ABSOLUTE` attribute to specify a root region. This attribute is the default for an execution region.

To specify a root region, use `ABSOLUTE` as the attribute for the execution region. You can either specify the attribute explicitly or permit it to default, and use the same address for the first execution region and the enclosing load region.

To make the execution region address the same as the load region address, either:

- Specify the same numeric value for both the base address for the execution region and the base address for the load region.
- Specify a `+0` offset for the first execution region in the load region.

  If you specify an offset of zero (`+0`) for all subsequent execution regions in the load region, then all execution regions not following an execution region containing ZI are also root regions.

Example

The following example shows an implicitly defined root region:

```
LR_1 0x040000                ; load region starts at 0x40000
{                            ; start of execution region descriptions
    ER_RO 0x040000 ABSOLUTE  ; load address = execution address
    {
        * (+RO)              ; all RO sections (must include the section
                             ; containing the initial entry point)
    }
    …                       ; rest of scatter-loading description
}
```

Related concepts

7.2 Root region and the initial entry point on page 7-111

7.2.2 Effect of the FIXED attribute on a root region on page 7-113

8.3 Load region descriptions on page 8-167

8.4 Execution region descriptions on page 8-173
8.3.6 Considerations when using a relative address +offset for a load region on page 8-171
8.4.5 Considerations when using a relative address +offset for execution regions on page 8-179
8.3.4 Inheritance rules for load region address attributes on page 8-170
8.3.5 Inheritance rules for the RELOC address attribute on page 8-171
8.4.4 Inheritance rules for execution region address attributes on page 8-178

Related reference

8.3.3 Load region attributes on page 8-169
8.4.3 Execution region attributes on page 8-175

Related information

ENTRY directive

7.2.2 Effect of the FIXED attribute on a root region

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

Use the `FIXED` execution region attribute to ensure that the load address and execution address of a specific region are the same.

You can use the `FIXED` attribute to place any execution region at a specific address in ROM.

For example, the following memory map shows fixed execution regions:

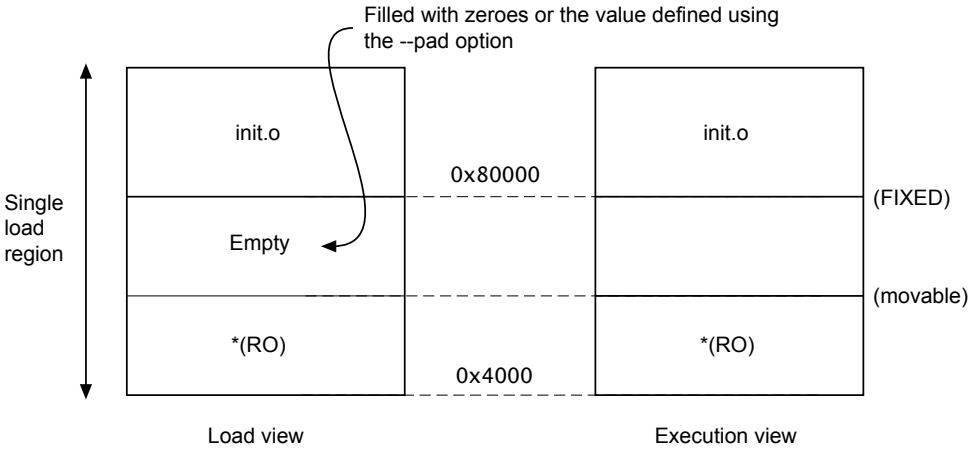

Figure 7-3 Memory map for fixed execution regions

The following example shows the corresponding scatter-loading description:

```
LR_1 0x040000              ; load region starts at 0x40000
{                          ; start of execution region descriptions
    ER_RO 0x040000         ; load address = execution address
    {
        * (+RO)            ; RO sections other than those in init.o
    }
    ER_INIT 0x080000 FIXED ; load address and execution address of this
                           ; execution region are fixed at 0x80000
    {
        init.o(+RO)        ; all RO sections from init.o
    }
    …                    ; rest of scatter-loading description
}
```

You can use this attribute to place a function or a block of data, for example a constant table or a checksum, at a fixed address in ROM. This makes it easier to access the function or block of data through pointers.

If you place two separate blocks of code or data at the start and end of ROM, some of the memory contents might be unused. For example, you might place some initialization code at the start of ROM and a checksum at the end of ROM. Use the * or `.ANY` module selector to flood fill the region between the end of the initialization block and the start of the data block.

To make your code easier to maintain and debug, use the minimum number of placement specifications in scatter files. Leave the detailed placement of functions and data to the linker.

Note

There are some situations where using `FIXED` and a single load region are not appropriate. Other techniques for specifying fixed locations are:

- If your loader can handle multiple load regions, place the RO code or data in its own load region.
- If you do not require the function or data to be at a fixed location in ROM, use `ABSOLUTE` instead of `FIXED`. The loader then copies the data from the load region to the specified address in RAM. `ABSOLUTE` is the default attribute.
- To place a data structure at the location of memory-mapped I/O, use two load regions and specify `UNINIT`. `UNINIT` ensures that the memory locations are not initialized to zero.

Example showing the misuse of the FIXED attribute

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

```
LR1 0x8000
{
    ER_LOW +0 0x1000
    {
        *(+RO)
    }
; At this point the next available Load and Execution address is 0x8000 + size of
; contents of ER_LOW. The maximum size is limited to 0x1000 so the next available Load
; and Execution address is at most 0x9000
    ER_HIGH 0xF0000000 FIXED
    {
        *(+RW,+ZI)
    }
; The required execution address and load address is 0xF0000000. The linker inserts
; 0xF0000000 - (0x8000 + size of(ER_LOW)) bytes of padding so that load address matches
; execution address
}
; The other common misuse of FIXED is to give a lower execution address than the next
; available load address.
LR_HIGH 0x100000000
{
    ER_LOW 0x1000 FIXED
    {
        *(+RO)
    }
; The next available load address in LR_HIGH is 0x10000000. The required Execution
; address is 0x1000. Because the next available load address in LR_HIGH must increase
; monotonically the linker cannot give ER_LOW a Load Address lower than 0x10000000
}
```

Related concepts

8.4 Execution region descriptions on page 8-173

8.3.4 Inheritance rules for load region address attributes on page 8-170

8.3.5 Inheritance rules for the RELOC address attribute on page 8-171

8.4.4 Inheritance rules for execution region address attributes on page 8-178

Related reference

8.3.3 Load region attributes on page 8-169

8.4.3 Execution region attributes on page 8-175

7.2.3 Methods of placing functions and data at specific addresses

There are various methods available to place functions and data at specific addresses.

Placing functions and data at specific addresses

To place a single function or data item at a fixed address, you must enable the linker to process the function or data separately from the rest of the input files.

Where they are required, the compiler normally produces RO, RW, and ZI sections from a single source file. These sections contain all the code and data from the source file.

Note

For images targeted at Armv7-M or Armv8-M, the compiler might generate *execute-only* (XO) sections.

Typically, you create a scatter file that defines an execution region at the required address with a section description that selects only one section.

To place a function or variable at a specific address, it must be placed in its own section. There are several ways to do this:

- Place the function or data item in its own source file.
- Use `__attribute__((section("name")))` to place functions and variables in a specially named section, `.ARM.__at_address`, where `address` is the address to place the function or variable. For example, `__attribute__((section(".ARM.__at_0x4000")))`.

  To place ZI data at a specific address, use the variable attribute `__attribute__((section("name")))` with the special name `.bss.ARM.__at_address`

  These specially named sections are called `__at` sections.
- Use the `.section` directive from assembly language. In assembly code, the smallest locatable unit is a `.section`.
- 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`.

Related concepts

7.3 Example of how to explicitly place a named section with scatter-loading on page 7-126
7.2.6 Restrictions on placing __at sections on page 7-121

Related tasks

7.2.5 Placing __at sections at a specific address on page 7-120

Related reference

11.5 --autoat, --no_autoat on page 11-223
11.85 --map, --no_map on page 11-314
11.120 --scatter=filename on page 11-350
11.92 -o filename, --output=filename on page 11-321

Related information

AREA directive

Placing a variable at a specific address without scatter-loading

This example shows how to modify your source code to place code and data at specific addresses, and does not require a scatter file.

To place code and data at specific addresses without a scatter file:

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

```
#include <stdio.h>

extern int sqr(int n1);
const int gValue __attribute__((section(".ARM.__at_0x5000"))) = 3; // Place at 0x5000
int main(void)
{
    int squared;
    squared=sqr(gValue);
    printf("Value squared is: %d\n", squared);
```

```
    return 0;
}
```

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

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

3. Compile and link the sources:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -c function.c
armclang --target=arm-arm-none-eabi -march=armv8-a -c main.c
armlink --map function.o main.o -o squared.axf
```

   The `--map` option displays the memory map of the image. Also, `--autoat` is the default.

In this example, `__attribute__((section(".ARM.__AT_0x5000")))` specifies that the global variable `gValue` is to be placed at the absolute address `0x5000`. `gValue` is placed in the execution region `ER$$.ARM.__AT_0x5000` and load region `LR$$.ARM.__AT_0x5000`.

The memory map shows:

```
…
  Load Region LR$$.ARM.__AT_0x5000 (Base: 0x00005000, Size: 0x00000004, Max: 0x00000004,
ABSOLUTE)

    Execution Region ER$$.ARM.__AT_0x5000 (Base: 0x00005000, Size: 0x00000004, Max:
0x00000004, ABSOLUTE, UNINIT)

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x00005000   0x00000004   Data   RO           18    .ARM.__AT_0x5000  main.o
```

Related reference

11.5 --autoat, --no_autoat on page 11-223
11.85 --map, --no_map on page 11-314
11.92 -o filename, --output=filename on page 11-321

Example of how to place a variable in a named section with scatter-loading

This example shows how to modify your source code to place code and data in a specific section using a scatter file.

To modify your source code to place code and data in a specific section using a scatter file:

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

```
#include <stdio.h>
extern int sqr(int n1);
int gSquared __attribute__((section("foo")));  // Place in section foo
int main(void)
{
    gSquared=sqr(3);
    printf("Value squared is: %d\n", gSquared);
    return 0;
}
```

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

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

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

```
LR1 0x0000 0x20000
{
    ER1 0x0 0x2000
    {
        *(+RO)                ; rest of code and read-only data
    }
    ER2 0x8000 0x2000
    {
        main.o
```

```
    }
    ER3 0x10000 0x2000
    {
        function.o
        *(foo)               ; Place gSquared in ER3
    }
    ; RW and ZI data to be placed at 0x200000
    RAM 0x200000 (0x1FF00-0x2000)
    {
        *(+RW, +ZI)
    }
    ARM_LIB_STACK 0x800000 EMPTY -0x10000
    {
    }
    ARM_LIB_HEAP  +0 EMPTY 0x10000
    {
    }
}
```

The `ARM_LIB_STACK` and `ARM_LIB_HEAP` regions are required because the program is being linked with the semihosting libraries.

4. Compile and link the sources:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -c function.c
armclang --target=arm-arm-none-eabi -march=armv8-a -c main.c
armlink --map --scatter=scatter.scat function.o main.o -o squared.axf
```

The `--map` option displays the memory map of the image. Also, `--autoat` is the default.

In this example, `__attribute__((section("foo")))` specifies that the global variable `gSquared` is to be placed in a section called `foo`. The scatter file specifies that the section `foo` is to be placed in the `ER3` execution region.

The memory map shows:

```
  Load Region LR1 (Base: 0x00000000, Size: 0x00001570, Max: 0x00020000, ABSOLUTE)
…
    Execution Region ER3 (Base: 0x00010000, Size: 0x00000010, Max: 0x00002000, ABSOLUTE)

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x00010000   0x0000000c   Code   RO            3    .text               function.o
    0x0001000c   0x00000004   Data   RW           15    foo                 main.o
…
```

Note

If you omit `*(foo)` from the scatter file, the section is placed in the region of the same type. That is `RAM` in this example.

Related reference

11.5 --autoat, --no_autoat on page 11-223
11.85 --map, --no_map on page 11-314
11.92 -o filename, --output=filename on page 11-321
11.120 --scatter=filename on page 11-350

Placing a variable at a specific address with scatter-loading

This example shows how to modify your source code to place code and data at a specific address using a scatter file.

To modify your source code to place code and data at a specific address using a scatter file:

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

```
#include <stdio.h>
extern int sqr(int n1);
// Place at address 0x10000
const int gValue __attribute__((section(".ARM.__at_0x10000"))) = 3;
int main(void)
{
    int squared;
    squared=sqr(gValue);
```

```
    printf("Value squared is: %d\n", squared);
    return 0;
}
```

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

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

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

```
LR1 0x0
{
    ER1 0x0
    {
        *(+RO)                      ; rest of code and read-only data
    }
    ER2 +0
    {
        function.o
        *(.ARM.__at_0x10000)        ; Place gValue at 0x10000
    }
    ; RW and ZI data to be placed at 0x200000
    RAM 0x200000 (0x1FF00-0x2000)
    {
        *(+RW, +ZI)
    }
    ARM_LIB_STACK 0x800000 EMPTY -0x10000
    {
    }
    ARM_LIB_HEAP  +0 EMPTY 0x10000
    {
    }
}
```

The `ARM_LIB_STACK` and `ARM_LIB_HEAP` regions are required because the program is being linked with the semihosting libraries.

4. Compile and link the sources:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -c function.c
armclang --target=arm-arm-none-eabi -march=armv8-a -c main.c
armlink --no_autoat --scatter=scatter.scat --map function.o main.o -o squared.axf
```

The `--map` option displays the memory map of the image.

The memory map shows that the variable is placed in the `ER2` execution region at address `0x10000`:

```
…
    Execution Region ER2 (Base: 0x00002a54, Size: 0x0000d5b0, Max: 0xffffffff, ABSOLUTE)

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x00002a54   0x0000001c   Code   RO            4    .text.sqr           function.o
    0x00002a70   0x0000d590   PAD
    0x00010000   0x00000004   Data   RO            9    .ARM.__at_0x10000   main.o
```

In this example, the size of `ER1` is unknown. Therefore, `gValue` might be placed in `ER1` or `ER2`. To make sure that `gValue` is placed in `ER2`, you must include the corresponding selector in `ER2` and link with the `--no_autoat` command-line option. If you omit `--no_autoat`, `gValue` is placed in a separate load region `LR$$.ARM.__at_0x10000` that contains the execution region `ER$$.ARM.__at_0x10000`.

Related reference

11.5 --autoat, --no_autoat on page 11-223
11.85 --map, --no_map on page 11-314
11.92 -o filename, --output=filename on page 11-321
11.120 --scatter=filename on page 11-350

7.2.4 Placing functions and data in a named section

You can place functions and data by separating them into their own objects without having to use toolchain-specific pragmas or attributes. Alternatively, you can specify a name of a section using the function or variable attribute, `__attribute__((section("name")))`.

You can use `__attribute__((section("name")))` to place a function or variable in a separate ELF section, where *name* is a name of your choice. You can then use a scatter file to place the named sections at specific locations.

You can place ZI data in a named section with `__attribute__((section(".bss.name")))`.

Use the following procedure to modify your source code to place functions and data in a specific section using a scatter file.

Procedure

1. Create a C source file `file.c` to specify a section name `foo` for a variable and a section name `.bss.mybss` for a zero-initialized variable `z`, for example:

```
#include "stdio.h"

int variable __attribute__((section("foo"))) = 10;
__attribute__((section(".bss.mybss"))) int z;

int main(void)
{
    int x = 4;
    int y = 7;
    z = x + y;
    printf("%d\n",variable);
    printf("%d\n",z);
    return 0;
}
```

2. Create a scatter file to place the named section, `scatter.scat`, for example:

```
LR_1 0x0
{
    ER_RO 0x0 0x4000
    {
        *(+RO)
    }
    ER_RW 0x4000 0x2000
    {
        *(+RW)
    }
    ER_ZI 0x6000 0x2000
    {
        *(+ZI)
    }
    ER_MYBSS 0x8000 0x2000
    {
        *(.bss.mybss)
    }

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

FLASH 0x24000000 0x4000000
{
    ; rest of code

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

}
```

   The `ARM_LIB_STACK` and `ARM_LIB_HEAP` regions are required because the program is being linked with the semihosting libraries.

   Note

   If you omit `file.o (foo)` from the scatter file, the linker places the section in the region of the same type. That is, `ER_RW` in this example.

3. Compile and link the C source:

```
armclang --target=arm-arm-eabi-none  -march=armv8-a file.c -g -c -O1 -o file.o
armlink --cpu=8-A.32 --scatter=scatter.scat --map file.o --output=file.axf
```

The `--map` option displays the memory map of the image.

Example:
In this example:

- `__attribute__((section("foo")))` specifies that the linker is to place the global variable `variable` in a section called `foo`.
- `__attribute__((section(".bss.mybss")))` specifies that the linker is to place the global variable `z` in a section called `.bss.mybss`.
- The scatter file specifies that the linker is to place the section `foo` in the `ADDER` execution region of the `FLASH` execution region.

The following example shows the output from `--map`:

```
…
    Execution Region ER_MYBSS (Base: 0x00008000, Size: 0x00000004, Max: 0x00002000,
ABSOLUTE)

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x00008000   0x00000004   Zero   RW            7    .bss.mybss          file.o
…
  Load Region FLASH (Base: 0x24000000, Size: 0x00000004, Max: 0x04000000, ABSOLUTE)

    Execution Region ADDER (Base: 0x08000000, Size: 0x00000004, Max: 0xffffffff, ABSOLUTE)

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x08000000   0x00000004   Data   RW            5    foo                 file.o
…
```

Note

- If scatter-loading is not used, the linker places the section `foo` in the default `ER_RW` execution region of the `LR_1` load region. It also places the section `.bss.mybss` in the default execution region `ER_ZI`.
- If you have a scatter file that does not include the `foo` selector, then the linker places the section in the defined RW execution region.

You can also place a function at a specific address using `.ARM.__at_address` as the section name. For example, to place the function `sqr` at `0x20000`, specify:

```
int sqr(int n1) __attribute__((section(".ARM.__at_0x20000")));

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

For more information, see *Placing functions and data at specific addresses* on page 7-114.

Related concepts

7.2.6 Restrictions on placing __at sections on page 7-121

Related tasks

7.2.5 Placing __at sections at a specific address on page 7-120

Related reference

11.5 --autoat, --no_autoat on page 11-223
11.120 --scatter=filename on page 11-350

7.2.5 Placing __at sections at a specific address

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

To place a section at a specific address, use the function or variable attribute `__attribute__((section("name")))` with the special name `.ARM.__at_address`.

To place ZI data at a specific address, use the variable attribute `__attribute__((section("name")))` with the special name `.bss.ARM.__at_address`

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

The following example shows how to assign a variable to a specific address in C or C++ code:

```
// place variable1 in a section called .ARM.__at_0x8000
int variable1 __attribute__((section(".ARM.__at_0x8000"))) = 10;
```

Note

The name of the section is only significant if you are trying to match the section by name in a scatter file. Without overlays, the linker automatically assigns `__at` sections when you use the `--autoat` command-line option. This option is the default. If you are using overlays, then you cannot use `--autoat` to place `__at` sections.

Related concepts

7.2.6 Restrictions on placing __at sections on page 7-121

Related tasks

Placing functions and data at specific addresses on page 7-114

7.2.4 Placing functions and data in a named section on page 7-118

7.2.7 Automatically placing __at sections on page 7-121

7.2.8 Manually placing __at sections on page 7-123

7.2.9 Placing a key in flash memory with an __at section on page 7-124

Related reference

11.5 --autoat, --no_autoat on page 11-223

7.2.6 Restrictions on placing __at sections

There are restrictions when placing `__at` sections at specific addresses.

The following restrictions apply:

- `__at` section address ranges must not overlap, unless the overlapping sections are placed in different overlay regions.
- `__at` sections are not permitted in position independent execution regions.
- You must not reference the linker-defined symbols `$$Base`, `$$Limit` and `$$Length` of an `__at` section.
- `__at` sections must not be used in *Base Platform Application Binary Interface* (BPABI) executables and BPABI *dynamically linked libraries* (DLLs).
- `__at` sections must have an address that is a multiple of their alignment.
- `__at` sections ignore any `+FIRST` or `+LAST` ordering constraints.

Related tasks

7.2.5 Placing __at sections at a specific address on page 7-120

Related information

Base Platform ABI for the Arm Architecture

7.2.7 Automatically placing __at sections

The automatic placement of `__at` sections is enabled by default. Use the linker command-line option, `--no_autoat` to disable this feature.

Note

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

When linking with the `--autoat` option, the linker does not place `__at` sections with scatter-loading selectors. Instead, the linker places the `__at` section in a compatible region. If no compatible region is found, the linker creates a load and execution region for the `__at` section.

All linker execution regions created by `--autoat` have the `UNINIT` scatter-loading attribute. If you require a ZI `__at` section to be zero-initialized, then it must be placed within a compatible region. A linker execution region created by `--autoat` must have a base address that is at least 4 byte-aligned. If any region is incorrectly aligned, the linker produces an error message.

A compatible region is one where:

- The `__at` address lies within the execution region base and limit, where limit is the base address + maximum size of execution region. If no maximum size is set, the linker sets the limit for placing `__at` sections as the current size of the execution region without `__at` sections plus a constant. The default value of this constant is `10240` bytes, but you can change the value using the `--max_er_extension` command-line option.
- The execution region meets at least one of the following conditions:
  - It has a selector that matches the `__at` section by the standard scatter-loading rules.
  - It has at least one section of the same type (RO or RW) as the `__at` section.
  - It does not have the `EMPTY` attribute.

    Note

    The linker considers an `__at` section with type RW compatible with RO.

The following example shows the sections `.ARM.__at_0x0000` type RO, `.ARM.__at_0x4000` type RW, and `.ARM.__at_0x8000` type RW:

```
// place the RO variable in a section called .ARM.__at_0x0000
const int foo __attribute__((section(".ARM.__at_0x0000"))) = 10;

// place the RW variable in a section called .ARM.__at_0x4000
int bar __attribute__((section(".ARM.__at_0x4000"))) = 100;

// place "variable" in a section called .ARM.__at_0x00008000
int variable __attribute__((section(".ARM.__at_0x00008000")));
```

The following scatter file shows how automatically to place these `__at` sections:

```
LR1 0x0
{
    ER_RO 0x0 0x4000
    {
        *(+RO)      ; .ARM.__at_0x0000 lies within the bounds of ER_RO
    }
    ER_RW 0x4000 0x2000
    {
        *(+RW)      ; .ARM.__at_0x4000 lies within the bounds of ER_RW
    }
    ER_ZI 0x6000 0x2000
    {
        *(+ZI)
    }
}
; The linker creates a load and execution region for the __at section
; .ARM.__at_0x8000 because it lies outside all candidate regions.
```

Related concepts

8.4 Execution region descriptions on page 8-173

7.2.6 Restrictions on placing __at sections on page 7-121

Related tasks

7.2.5 Placing __at sections at a specific address on page 7-120

7.2.8 Manually placing __at sections on page 7-123
7.2.9 Placing a key in flash memory with an __at section on page 7-124
7.2.4 Placing functions and data in a named section on page 7-118

Related reference

11.5 --autoat, --no_autoat on page 11-223
11.114 --ro_base=address on page 11-344
11.117 --rw_base=address on page 11-347
11.157 --xo_base=address on page 11-389
11.161 --zi_base=address on page 11-393
8.4.3 Execution region attributes on page 8-175
11.86 --max_er_extension=size on page 11-315

Related information

__attribute__((section("name"))) variable attribute

7.2.8 Manually placing __at sections

You can have direct control over the placement of `__at` sections, if required.

You can use the standard section-placement rules to place `__at` sections when using the `--no_autoat` command-line option.

Note

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

The following example shows the placement of read-only sections `.ARM.__at_0x2000` and the read-write section `.ARM.__at_0x4000`. Load and execution regions are not created automatically in manual mode. An error is produced if an `__at` section cannot be placed in an execution region.

The following example shows the placement of the variables in C or C++ code:

```
// place the RO variable in a section called .ARM.__at_0x2000
const int foo __attribute__((section(".ARM.__at_0x2000"))) = 100;
// place the RW variable in a section called .ARM.__at_0x4000
int bar __attribute__((section(".ARM.__at_0x4000")));
```

The following scatter file shows how to place `__at` sections manually:

```
LR1 0x0
{
    ER_RO 0x0 0x2000
    {
        *(+RO)               ; .ARM.__at_0x0000 is selected by +RO
    }
    ER_RO2 0x2000
    {
        *(.ARM.__at_0x02000)  ; .ARM.__at_0x2000 is selected by the section named
                              ; .ARM.__at_0x2000
    }
    ER2 0x4000
    {
        *(+RW, +ZI)           ; .ARM.__at_0x4000 is selected by +RW
    }
}
```

Related concepts

8.4 Execution region descriptions on page 8-173
7.2.6 Restrictions on placing __at sections on page 7-121

Related tasks

7.2.5 Placing __at sections at a specific address on page 7-120
7.2.7 Automatically placing __at sections on page 7-121
7.2.9 Placing a key in flash memory with an __at section on page 7-124
7.2.4 Placing functions and data in a named section on page 7-118

Related reference

11.5 --autoat, --no_autoat on page 11-223

8.4.3 Execution region attributes on page 8-175

Related information

__attribute__((section("name"))) variable attribute

7.2.9 Placing a key in flash memory with an __at section

Some flash devices require a key to be written to an address to activate certain features. An `__at` section provides a simple method of writing a value to a specific address.

Placing the flash key variable in C or C++ code

Assume that a device has flash memory from `0x8000` to `0x10000` and a key is required in address `0x8000`. To do this with an `__at` section, you must declare a variable so that the compiler can generate a section called `.ARM.__at_0x8000`.

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

Manually placing a flash execution region

The following example shows how to manually place a flash execution region with a scatter file:

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

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

Automatically placing a flash execution region

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

```
LR1 0x0
{
    ER_FLASH 0x8000 0x2000
    {
        *(+RO)                  ; other code and read-only data, the
                                ; __at section is automatically selected
    }
    ER2 0x4000
    {
        *(+RW +ZI)              ; Any other RW and ZI variables
    }
}
```

Related concepts

8.4 Execution region descriptions on page 8-173

3.3.2 Section placement with the FIRST and LAST attributes on page 3-49

Related tasks

7.2.5 Placing __at sections at a specific address on page 7-120

7.2.7 Automatically placing __at sections on page 7-121

7.2.8 Manually placing __at sections on page 7-123

Related reference

11.5 --autoat, --no_autoat on page 11-223

Related concepts

7.2.1 Effect of the ABSOLUTE attribute on a root region on page 7-111

7.2.2 Effect of the FIXED attribute on a root region on page 7-113

3.1 The structure of an Arm® ELF image on page 3-33

Related reference

7.8 Placement of Arm® C and C++ library code on page 7-144

7.3 Example of how to explicitly place a named section with scatter-loading

This example shows how to place a named section explicitly using scatter-loading.

Consider the following source files:

```
init.c
------
int foo() __attribute__((section("INIT")));
int foo() {
  return 1;
}

int bar() {
  return 2;
}

data.c
------
const long padding=123;
int z=5;
```

The following scatter file shows how to place a named section explicitly:

```
LR1 0x0 0x10000
{
    ; Root Region, containing init code
    ER1 0x0 0x2000
    {
        init.o (INIT, +FIRST)   ; place init code at exactly 0x0
        *(+RO)                  ; rest of code and read-only data
    }
    ; RW & ZI data to be placed at 0x400000
    RAM_RW 0x400000 (0x1FF00-0x2000)
    {
        *(+RW)
    }
    RAM_ZI +0
    {
        *(+ZI)
    }
    ; execution region at 0x1FF00
    ; maximum space available for table is 0xFF
    DATABLOCK 0x1FF00 0xFF
    {
        data.o(+RO-DATA)  ; place RO data between 0x1FF00 and 0x1FFFF
    }
}
```

In this example, the scatter-loading description places:

- The initialization code is placed in the `INIT` section in the `init.o` file. This example shows that the code from the `INIT` section is placed first, at address `0x0`, followed by the remainder of the RO code and all of the RO data except for the RO data in the object `data.o`.
- All global RW variables in RAM at `0x400000`.
- A table of `RO-DATA` from `data.o` at address `0x1FF00`.

The resulting image memory map is as follows:

```
Memory Map of the image

  Image entry point : Not specified.

  Load Region LR1 (Base: 0x00000000, Size: 0x00000018, Max: 0x00010000, ABSOLUTE)

    Execution Region ER1 (Base: 0x00000000, Size: 0x00000010, Max: 0x00002000, ABSOLUTE)

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x00000000   0x00000008   Code   RO            4    INIT               init.o
    0x00000008   0x00000008   Code   RO            1    .text              init.o
    0x00000010   0x00000000   Code   RO           16    .text              data.o


    Execution Region DATABLOCK (Base: 0x0001ff00, Size: 0x00000004, Max: 0x000000ff,
ABSOLUTE)

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object
```

```
0x0001ff00   0x00000004   Data   RO           19    .rodata            data.o


Execution Region RAM_RW (Base: 0x00400000, Size: 0x00000004, Max: 0x0001df00, ABSOLUTE)

Base Addr    Size         Type   Attr      Idx    E Section Name        Object

0x00400000   0x00000000   Data   RW            2    .data              init.o
0x00400000   0x00000004   Data   RW           17    .data              data.o


Execution Region RAM_ZI (Base: 0x00400004, Size: 0x00000000, Max: 0xffffffff, ABSOLUTE)

Base Addr    Size         Type   Attr      Idx    E Section Name        Object

0x00400004   0x00000000   Zero   RW            3    .bss               init.o
0x00400004   0x00000000   Zero   RW           18    .bss               data.o
```

Related concepts

7.2.2 Effect of the FIXED attribute on a root region on page 7-113

8.3 Load region descriptions on page 8-167

8.4 Execution region descriptions on page 8-173

8.3.4 Inheritance rules for load region address attributes on page 8-170

8.3.5 Inheritance rules for the RELOC address attribute on page 8-171

8.4.4 Inheritance rules for execution region address attributes on page 8-178

Related reference

8.3.3 Load region attributes on page 8-169

8.4.3 Execution region attributes on page 8-175

Related information

ENTRY

7.4 Placement of unassigned sections

The linker attempts to place input sections into specific execution regions. For any input sections that cannot be resolved, and where the placement of those sections is not important, you can specify where the linker is to place them.

To place sections that are not automatically assigned to specific execution regions, use the `.ANY` module selector in a scatter file.

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

The linker has default rules for placing unassigned sections when you specify multiple `.ANY` selectors. However, you can override the default rules using the following command-line options:

- `--any_contingency` to permit extra space in any execution regions containing `.ANY` sections for linker-generated content such as veneers and alignment padding.
- `--any_placement` to provide more control over the placement of unassigned sections.
- `--any_sort_order` to control the sort order of unassigned input sections.

In a scatter file, you can also:

- Assign a priority to a `.ANY` selector. This gives you more control over how the unassigned sections are divided between multiple execution regions. You can assign the same priority to more than one execution region.
- Specify the maximum size for an execution region that the linker can fill with unassigned sections.

This section contains the following subsections:

7.4.1 Default rules for placing unassigned sections

The linker has default rules for placing sections when using multiple `.ANY` selectors.

When more than one `.ANY` selector is present in a scatter file, the linker sorts sections in descending size order. It then takes the unassigned section with the largest size and assigns the section to the most specific `.ANY` execution region that has enough free space. For example, `.ANY(.text)` is judged to be more specific than `.ANY(+RO)`.

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

For example:

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

You can specify a maximum amount of space to use for unassigned sections with the execution region attribute `ANY_SIZE`.

Related concepts

7.14 How the linker resolves multiple matches when processing scatter files on page 7-159

7.4.8 Behavior when .ANY sections overflow because of linker-generated content on page 7-136

Related reference

11.2 --any_placement=algorithm on page 11-219

11.1 --any_contingency on page 11-218

7.4 Placement of unassigned sections on page 7-128

8.5.2 Syntax of an input section description on page 8-181

11.59 --info=topic[,topic,...] on page 11-281

7.4.2 Command-line options for controlling the placement of unassigned sections

You can modify how the linker places unassigned input sections when using multiple `.ANY` selectors by using a different placement algorithm or a different sort order.

The following command-line options are available:

- `--any_placement=`*`algorithm`*, where *`algorithm`* is one of `first_fit`, `worst_fit`, `best_fit`, or `next_fit`.
- `--any_sort_order=`*`order`*, where *`order`* is one of `cmdline` or `descending_size`.

Use `first_fit` when you want to fill regions in order.

Use `best_fit` when you want to fill regions to their maximum.

Use `worst_fit` when you want to fill regions evenly. With equal sized regions and sections `worst_fit` fills regions cyclically.

Use `next_fit` when you need a more deterministic fill pattern.

If the linker attempts to fill a region to its limit, as it does with `first_fit` and `best_fit`, it might overfill the region. This is because linker-generated content such as padding and veneers are not known until sections have been assigned to `.ANY` selectors. If this occurs you might see the following error:

`Error: L6220E: Execution region `*`regionname`*` size (`*`size`*` bytes) exceeds limit (`*`limit`*` bytes)`.

The `--any_contingency` option prevents the linker from filling the region up to its maximum. It reserves a portion of the region's size for linker-generated content and fills this contingency area only if no other regions have space. It is enabled by default for the `first_fit` and `best_fit` algorithms, because they are most likely to exhibit this behavior.

Related concepts

7.4.5 Examples of using placement algorithms for .ANY sections on page 7-131

7.4.6 Example of next_fit algorithm showing behavior of full regions, selectors, and priority on page 7-133

7.4.7 Examples of using sorting algorithms for .ANY sections on page 7-134

7.4.8 Behavior when .ANY sections overflow because of linker-generated content on page 7-136

Related reference

11.3 --any_sort_order=order on page 11-221

11.85 --map, --no_map on page 11-314

11.121 --section_index_display=type on page 11-352

11.142 --tiebreaker=option on page 11-374

11.2 --any_placement=algorithm on page 11-219

11.1 --any_contingency on page 11-218

7.4.3 Prioritizing the placement of unassigned sections

You can give a priority ordering when placing unassigned sections with multiple `.ANY` module selectors.

To prioritize the order of multiple `.ANY` sections use the `.ANY`*num* selector, where *num* is a positive integer starting at zero.

The highest priority is given to the selector with the highest integer.

The following example shows how to use `.ANY`*num*:

```
lr1 0x8000 1024
{
    er1 +0 512
    {
        .ANY1(+RO) ; evenly distributed with er3
    }
    er2 +0 256
    {
        .ANY2(+RO) ; Highest priority, so filled first
    }
    er3 +0 256
    {
        .ANY1(+RO) ; evenly distributed with er1
    }
}
```

Related concepts

7.4.5 Examples of using placement algorithms for .ANY sections on page 7-131

7.4.6 Example of next_fit algorithm showing behavior of full regions, selectors, and priority on page 7-133

7.4.7 Examples of using sorting algorithms for .ANY sections on page 7-134

7.4.8 Behavior when .ANY sections overflow because of linker-generated content on page 7-136

7.14 How the linker resolves multiple matches when processing scatter files on page 7-159

Related reference

11.3 --any_sort_order=order on page 11-221

11.85 --map, --no_map on page 11-314

11.121 --section_index_display=type on page 11-352

11.142 --tiebreaker=option on page 11-374

7.4.4 Specify the maximum region size permitted for placing unassigned sections

You can specify the maximum size in a region that `armlink` can fill with unassigned sections.

Use the execution region attribute `ANY_SIZE` *`max_size`* to specify the maximum size in a region that `armlink` can fill with unassigned sections.

Be aware of the following restrictions when using this keyword:

- *`max_size`* must be less than or equal to the region size.
- If you use `ANY_SIZE` on a region without a `.ANY` selector, it is ignored by `armlink`.

When `ANY_SIZE` is present, `armlink` does not attempt to calculate contingency and strictly follows the `.ANY` priorities.

When `ANY_SIZE` is not present for an execution region containing a `.ANY` selector, and you specify the `--any_contingency` command-line option, then `armlink` attempts to adjust the contingency for that execution region. The aims are to:

- Never overflow a `.ANY` region.
- Make sure there is a contingency reserved space left in the given execution region. This space is reserved for veneers and section padding.

If you specify `--any_contingency` on the command line, it is ignored for regions that have `ANY_SIZE` specified. It is used as normal for regions that do not have `ANY_SIZE` specified.

Example

The following example shows how to use `ANY_SIZE`:

```
LOAD_REGION 0x0 0x3000
{
    ER_1 0x0 ANY_SIZE 0xF00 0x1000
    {
        .ANY
    }
    ER_2 0x0 ANY_SIZE 0xFB0 0x1000
    {
        .ANY
    }
    ER_3 0x0 ANY_SIZE 0x1000 0x1000
    {
        .ANY
    }
}
```

In this example:

- `ER_1` has `0x100` reserved for linker-generated content.
- `ER_2` has `0x50` reserved for linker-generated content. That is about the same as the automatic contingency of `--any_contingency`.
- `ER_3` has no reserved space. Therefore, 100% of the region is filled, with no contingency for veneers. Omitting the `ANY_SIZE` parameter causes 98% of the region to be filled, with a two percent contingency for veneers.

Related concepts

7.4.5 Examples of using placement algorithms for .ANY sections on page 7-131

7.4.6 Example of next_fit algorithm showing behavior of full regions, selectors, and priority on page 7-133

7.4.7 Examples of using sorting algorithms for .ANY sections on page 7-134

7.4.8 Behavior when .ANY sections overflow because of linker-generated content on page 7-136

Related reference

11.3 --any_sort_order=order on page 11-221

11.85 --map, --no_map on page 11-314

11.1 --any_contingency on page 11-218

7.4.5 Examples of using placement algorithms for .ANY sections

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

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

Table 7-1 Input section properties for placement of .ANY sections

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

The scatter file that the examples use is:

```
LR 0x100
{
  ER_1 0x100 0x10
  {
```

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

Note

These examples have `--any_contingency` disabled.

Example for first_fit, next_fit, and best_fit

This example shows the image memory map where several sections of equal size are assigned to two regions with one selector. The selectors are equally specific, equivalent to `.ANY(+R0)` and have no priority.

```
    Execution Region ER_1 (Base: 0x00000100, Size: 0x00000010, Max: 0x00000010, ABSOLUTE)

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x00000100   0x00000004   Code   RO            1    sec1               sections.o
    0x00000104   0x00000004   Code   RO            2    sec2               sections.o
    0x00000108   0x00000004   Code   RO            3    sec3               sections.o
    0x0000010c   0x00000004   Code   RO            4    sec4               sections.o


    Execution Region ER_2 (Base: 0x00000200, Size: 0x00000008, Max: 0x00000010, ABSOLUTE)

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x00000200   0x00000004   Code   RO            5    sec5               sections.o
    0x00000204   0x00000004   Code   RO            6    sec6               sections.o
```

In this example:

- For `first_fit`, the linker first assigns all the sections it can to `ER_1`, then moves on to `ER_2` because that is the next available region.
- For `next_fit`, the linker does the same as `first_fit`. However, when `ER_1` is full it is marked as `FULL` and is not considered again. In this example, `ER_1` is full. `ER_2` is then considered.
- For `best_fit`, the linker assigns `sec1` to `ER_1`. It then has two regions of equal priority and specificity, but `ER_1` has less space remaining. Therefore, the linker assigns `sec2` to `ER_1`, and continues assigning sections until `ER_1` is full.

Example for worst_fit

This example shows the image memory map when using the `worst_fit` algorithm.

```
    Execution Region ER_1 (Base: 0x00000100, Size: 0x0000000c, Max: 0x00000010, ABSOLUTE)

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x00000100   0x00000004   Code   RO            1    sec1               sections.o
    0x00000104   0x00000004   Code   RO            3    sec3               sections.o
    0x00000108   0x00000004   Code   RO            5    sec5               sections.o


    Execution Region ER_2 (Base: 0x00000200, Size: 0x0000000c, Max: 0x00000010, ABSOLUTE)

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x00000200   0x00000004   Code   RO            2    sec2               sections.o
    0x00000204   0x00000004   Code   RO            4    sec4               sections.o
    0x00000208   0x00000004   Code   RO            6    sec6               sections.o
```

The linker first assigns `sec1` to `ER_1`. It then has two equally specific and priority regions. It assigns `sec2` to the one with the most free space, `ER_2` in this example. The regions now have the same amount of space remaining, so the linker assigns `sec3` to the first one that appears in the scatter file, that is `ER_1`.

Note

The behavior of `worst_fit` is the default behavior in this version of the linker, and it is the only algorithm available in earlier linker versions.

Related concepts

7.4.2 Command-line options for controlling the placement of unassigned sections on page 7-129

7.4.6 Example of next_fit algorithm showing behavior of full regions, selectors, and priority on page 7-133

7.4.4 Specify the maximum region size permitted for placing unassigned sections on page 7-130

Related tasks

7.4.3 Prioritizing the placement of unassigned sections on page 7-129

Related reference

11.120 --scatter=filename on page 11-350

7.4.6 Example of next_fit algorithm showing behavior of full regions, selectors, and priority

This example shows the operation of the `next_fit` placement algorithm for `RO-CODE` sections in `sections.o`.

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

Table 7-2 Input section properties for placement of sections with next_fit

Name	Size
sec1	0x14
sec2	0x14
sec3	0x10
sec4	0x4
sec5	0x4
sec6	0x4

The scatter file used for the examples is:

```
LR 0x100
{
  ER_1 0x100 0x20
  {
     .ANY1(+RO-CODE)
  }
  ER_2 0x200 0x20
  {
     .ANY2(+RO)
  }
  ER_3 0x300 0x20
  {
     .ANY3(+RO)
  }
}
```

Note

This example has `--any_contingency` disabled.

The `next_fit` algorithm is different to the others in that it never revisits a region that is considered to be full. This example also shows the interaction between priority and specificity of selectors. This is the same for all the algorithms.

```
Execution Region ER_1 (Base: 0x00000100, Size: 0x00000014, Max: 0x00000020, ABSOLUTE)

Base Addr    Size         Type   Attr      Idx    E Section Name        Object

0x00000100   0x00000014   Code   RO            1    sec1                sections.o


Execution Region ER_2 (Base: 0x00000200, Size: 0x0000001c, Max: 0x00000020, ABSOLUTE)

Base Addr    Size         Type   Attr      Idx    E Section Name        Object

0x00000200   0x00000010   Code   RO            3    sec3                sections.o
0x00000210   0x00000004   Code   RO            4    sec4                sections.o
0x00000214   0x00000004   Code   RO            5    sec5                sections.o
0x00000218   0x00000004   Code   RO            6    sec6                sections.o


Execution Region ER_3 (Base: 0x00000300, Size: 0x00000014, Max: 0x00000020, ABSOLUTE)

Base Addr    Size         Type   Attr      Idx    E Section Name        Object

0x00000300   0x00000014   Code   RO            2    sec2                sections.o
```

In this example:

- The linker places `sec1` in `ER_1` because `ER_1` has the most specific selector. `ER_1` now has `0x6` bytes remaining.
- The linker then tries to place `sec2` in `ER_1`, because it has the most specific selector, but there is not enough space. Therefore, `ER_1` is marked as full and is not considered in subsequent placement steps. The linker chooses `ER_3` for `sec2` because it has higher priority than `ER_2`.
- The linker then tries to place `sec3` in `ER_3`. It does not fit, so `ER_3` is marked as full and the linker places `sec3` in `ER_2`.
- The linker now processes `sec4`. This is `0x4` bytes so it can fit in either `ER_1` or `ER_3`. Because both of these sections have previously been marked as full, they are not considered. The linker places all remaining sections in `ER_2`.
- If another section `sec7` of size `0x8` exists, and is processed after `sec6` the example fails to link. The algorithm does not attempt to place the section in `ER_1` or `ER_3` because they have previously been marked as full.

Related concepts

7.4.4 Specify the maximum region size permitted for placing unassigned sections on page 7-130

7.4.2 Command-line options for controlling the placement of unassigned sections on page 7-129

7.4.5 Examples of using placement algorithms for .ANY sections on page 7-131

7.14 How the linker resolves multiple matches when processing scatter files on page 7-159

7.4.8 Behavior when .ANY sections overflow because of linker-generated content on page 7-136

Related tasks

7.4.3 Prioritizing the placement of unassigned sections on page 7-129

Related reference

11.120 --scatter=filename on page 11-350

7.4.7 Examples of using sorting algorithms for .ANY sections

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

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

Table 7-3 Input section properties and ordering for sections_a.o and sections_b.o

sections_a.o		sections_b.o	
Name	**Size**	**Name**	**Size**
seca_1	0x4	secb_1	0x4
seca_2	0x4	secb_2	0x4
seca_3	0x10	secb_3	0x10
seca_4	0x14	secb_4	0x14

Descending size example

The following linker command-line options are used for this example:

```
--any_sort_order=descending_size sections_a.o sections_b.o --scatter scatter.txt
```

The following table shows the order that the sections are processed by the `.ANY` assignment algorithm.

Table 7-4 Sort order for descending_size algorithm

Name	Size
seca_4	0x14
secb_4	0x14
seca_3	0x10
secb_3	0x10
seca_1	0x4
seca_2	0x4
secb_1	0x4
secb_2	0x4

With `--any_sort_order=descending_size`, sections of the same size use the creation index as a tiebreaker.

Command-line example

The following linker command-line options are used for this example:

```
--any_sort_order=cmdline sections_a.o sections_b.o --scatter scatter.txt
```

The following table shows the order that the sections are processed by the `.ANY` assignment algorithm.

Table 7-5 Sort order for cmdline algorithm

Name	Size
seca_1	0x4
seca_2	0x4
seca_3	0x10
seca_4	0x14
secb_1	0x4
secb_2	0x4

Table 7-5 Sort order for cmdline algorithm (continued)

Name	Size
secb_3	0x10
secb_4	0x14

That is, the input sections are sorted by command-line index.

Related concepts

7.4.2 Command-line options for controlling the placement of unassigned sections on page 7-129

7.4.4 Specify the maximum region size permitted for placing unassigned sections on page 7-130

Related tasks

7.4.3 Prioritizing the placement of unassigned sections on page 7-129

Related reference

11.3 --any_sort_order=order on page 11-221

11.120 --scatter=filename on page 11-350

7.4.8 Behavior when .ANY sections overflow because of linker-generated content

Because linker-generated content might cause `.ANY` sections to overflow, a contingency algorithm is included in the linker.

The linker does not know the address of a section until it is assigned to a region. Therefore, when filling `.ANY` regions, the linker cannot calculate the contingency space and cannot determine if calling functions require veneers. The linker provides a contingency algorithm that gives a worst-case estimate for padding and an extra two percent for veneers. To enable this algorithm, use the `--any_contingency` command-line option.

The following diagram represents an example image layout during `.ANY` placement:

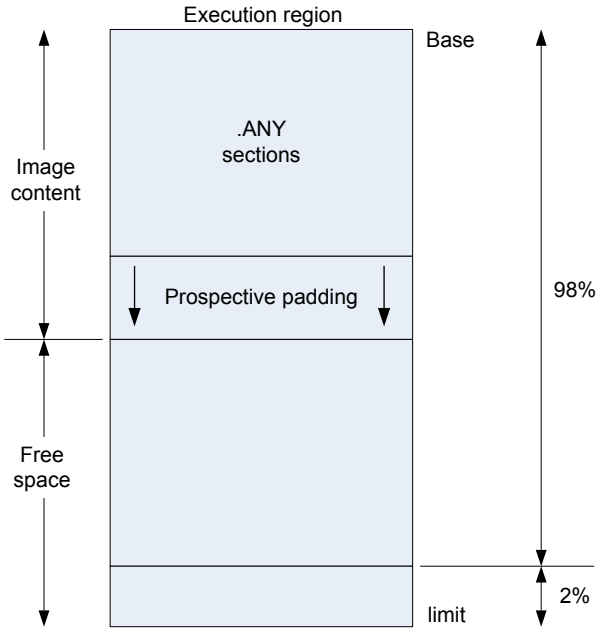

Figure 7-4 .ANY contingency

The downward arrows for prospective padding show that the prospective padding continues to grow as more sections are added to the `.ANY` selector.

Prospective padding is dealt with before the two percent veneer contingency.

When the prospective padding is cleared, the priority is set to zero. When the two percent is cleared, the priority is decremented again.

You can also use the `ANY_SIZE` keyword on an execution region to specify the maximum amount of space in the region to set aside for `.ANY` section assignments.

You can use the `armlink` command-line option `--info=any` to get extra information on where the linker has placed sections. This information can be useful when trying to debug problems.

Example

1. Create the following `foo.c` program:

```
#include "stdio.h"

int array[10] __attribute__ ((section ("ARRAY")));

struct S {
    char A[8];
    char B[4];
};
struct S s;

struct S* get()
{
    return &s;
}

int sqr(int n1);
int gSquared __attribute__((section(".ARM.__at_0x5000")));  // Place at 0x5000
int sqr(int n1)
{
    return n1*n1;
}

int main(void) {
    int i;
    for (i=0; i<10; i++) {
        array[i]=i*i;
        printf("%d\n", array[i]);
    }
    gSquared=sqr(i);
    printf("%d squared is: %d\n", i, gSquared);

    return sizeof(array);
}
```

2. Create the following `scatter.scat` file:

```
LOAD_REGION 0x0 0x3000
{
   ER_1 0x0 0x1000   {
      .ANY
   }
   ER_2 (ImageLimit(ER_1)) 0x1500   {
      .ANY
   }
   ER_3 (ImageLimit(ER_2)) 0x500
   {
      .ANY
   }
   ER_4 (ImageLimit(ER_3)) 0x1000
   {
       *(+RW,+ZI)
   }
   ARM_LIB_STACK 0x800000 EMPTY -0x10000
   {
   }
   ARM_LIB_HEAP  +0 EMPTY 0x10000
   {
   }
}
```

3. Compile and link the program as follows:

```
armclang -c --target=arm-arm-none-eabi -mcpu=cortex-m4 -o foo.o foo.c
armlink --cpu=cortex-m4 --any_contingency --scatter=scatter.scat --info=any -o foo.axf
foo.o
```

The following shows an example of the information generated:

```
================================================================================


Sorting unassigned sections by descending size for .ANY placement.
Using Worst Fit .ANY placement algorithm.
.ANY contingency enabled.

Exec Region    Event                          Idx          Size        Section
Name                  Object
ER_2           Assignment: Worst fit          144
0x0000041a  .text                          c_wu.l(_printf_fp_dec.o)
ER_2           Assignment: Worst fit          261          0x00000338  CL$
$btod_div_common           c_wu.l(btod.o)
ER_1           Assignment: Worst fit          146
0x000002fc  .text                          c_wu.l(_printf_fp_hex.o)
ER_2           Assignment: Worst fit          260          0x00000244  CL$
$btod_mult_common          c_wu.l(btod.o)
...
ER_1           Assignment: Worst fit          3
0x00000090  .text                          foo.o
...
ER_3           Assignment: Worst fit          100
0x0000000a  .ARM.Collect$$_printf_percent$$00000007  c_wu.l(_printf_ll.o)
ER_3           Info: .ANY limit reached       -            -
-                                 -
ER_1           Assignment: Highest priority  423
0x0000000a  .text                          c_wu.l(defsig_exit.o)
...
.ANY contingency summary
Exec Region    Contingency     Type
ER_1           161             Auto
ER_2           180             Auto
ER_3           73              Auto

================================================================================


Sorting unassigned sections by descending size for .ANY placement.
Using Worst Fit .ANY placement algorithm.
.ANY contingency enabled.

Exec Region    Event                          Idx          Size        Section
Name                  Object
ER_2           Info: .ANY limit reached       -            -
-                                 -
ER_1           Info: .ANY limit reached       -            -
-                                 -
ER_3           Info: .ANY limit reached       -            -
-                                 -
ER_2           Assignment: Worst fit          533          0x00000034  !!!
scatter                    c_wu.l(__scatter.o)
ER_2           Assignment: Worst fit          535          0x0000001c  !!
handler_zi                  c_wu.l(__scatter_zi.o)
```

Related concepts

7.4.2 Command-line options for controlling the placement of unassigned sections on page 7-129

7.4.4 Specify the maximum region size permitted for placing unassigned sections on page 7-130

Related tasks

7.4.3 Prioritizing the placement of unassigned sections on page 7-129

Related reference

11.1 --any_contingency on page 11-218

11.59 --info=topic[,topic,...] on page 11-281

8.5.2 Syntax of an input section description on page 8-181

8.4.3 Execution region attributes on page 8-175

7.5 Placing veneers with a scatter file

You can place veneers at a specific location with a linker-generated symbol.

Veneers allow switching between A32 and T32 code or allow a longer program jump than can be specified in a single instruction.

Procedure

1. To place veneers at a specific location, include the linker-generated symbol `Veneer$$Code` in a scatter file. At most, one execution region in the scatter file can have the `*(Veneer$$Code)` section selector.

   If it is safe to do so, the linker places veneer input sections into the region identified by the `*(Veneer$$Code)` section selector. It might not be possible for a veneer input section to be assigned to the region because of address range problems or execution region size limitations. If the veneer cannot be added to the specified region, it is added to the execution region containing the relocated input section that generated the veneer.

   Note

   Instances of `*(IWV$$Code)` in scatter files from earlier versions of Arm tools are automatically translated into `*(Veneer$$Code)`. Use `*(Veneer$$Code)` in new descriptions.

   `*(Veneer$$Code)` is ignored when the amount of code in an execution region exceeds 4MB of 16-bit T32 code, 16MB of 32-bit T32 code, and 32MB of A32 code.

   Note

   There are no state-change veneers in A64.

Related concepts

3.6 Linker-generated veneers on page 3-53

7.6 Placement of CMSE veneer sections for a Secure image

`armlink` automatically generates all CMSE veneer sections for a Secure image.

The linker:

- Creates `__at` sections that are called `Veneer$$CMSE_AT_`*`address`* for secure gateway veneers that you specify in a user-defined input import library.
- Produces one normal section `Veneer$$CMSE` to hold all other secure gateway veneers.

Placement of secure gateway veneers generated from input import libraries

The following example shows the placement of secure gateway veneers for functions `entry1` and `entry2` that are specified in the input import library:

```
...

** Section #4 'ER$$Veneer$$CMSE_AT_0x00004000' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR +
SHF_ARM_NOREAD]
    Size   : 32 bytes (alignment 32)
    Address: 0x00004000

    $t
    entry1
        0x00004000:    e97fe97f    ....    SG        ; [0x3e08]
        0x00004004:    f004b85a    ..Z.    B.W      __acle_se_entry1 ; 0x80bc
    entry2
        0x00004008:    e97fe97f    ....    SG        ; [0x3e10]
        0x0000400c:    f004b868    ..h.    B.W      __acle_se_entry2 ; 0x80e0

...
```

The same rules and options that apply to normal `__at` sections apply to `__at` sections created for secure gateway veneers. The same rules and options also apply to the automatic placement of these sections when you specify `--autoat`.

Placement of secure gateway veneers that are not specified in the input import library

Secure gateway veneers that do not have their addresses specified in an input import library get generated in the `Veneer$$CMSE` input section. You must place this section as required. If you create a simple image, that is without using a scatter file, the sections get placed in the ER_XO execution region, and the respective ER_XO output section.

The following example shows the placement of secure gateway veneers for functions `entry3` and `entry4` that are not specified in the input import library:

```
...

** Section #1 'ER_XO' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR + SHF_ARM_NOREAD]
    Size   : 32 bytes (alignment 32)
    Address: 0x00008000

    $t
    entry3
        0x00008000:    e97fe97f    ....    SG
        0x00008004:    f000b87e    ..~.    B.W      __acle_se_entry3 ; 0x8104
    entry4
        0x00008008:    e97fe97f    ....    SG
        0x0000800c:    f000b894    ....    B.W      __acle_se_entry4 ; 0x8138

...
```

Placement of secure gateway veneers with a scatter file

To make sure all the secure gateway veneers are in a single section, you must place them using a scatter file.

Secure gateway veneers that are not specified in the input import library are new veneers. New veneers get generated in the `Veneer$$CMSE` input section. You can place this section in the scatter file as required. Veneers that are already present in the input import library are placed at the address that is specified in this library. This placement is done by creating `Veneer$$CMSE_AT_`*`address`* sections for them. These

sections use the same facility that is used by other AT sections. Therefore, if you use `--no_autoat`, you can place these sections either by using the `--autoat` mechanism or by manually placing them using a scatter file.

For a Non-secure callable region of size `0x1000` bytes with a base address of `0x4000` a suitable example of a scatter file load and execution region to match the veneers is:

```
LOAD_NSCR 0x4000 0x1000
{
    EXEC_NSCR 0x4000 0x1000
    {
        *(Veneer$$CMSE)
    }
}
```

The secure gateway veneers are placed as follows:

```
...

** Section #7 'EXEC_NSCR' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR + SHF_ARM_NOREAD]
    Size   : 64 bytes (alignment 32)
    Address: 0x00004000

    $t
    entry1
        0x00004000:    e97fe97f    ....    SG
        0x00004004:    f7fcb850    ..P.    B        __acle_se_entry1 ; 0xa8
    entry2
        0x00004008:    e97fe97f    ....    SG
        0x0000400c:    f7fcb85e    ..^.    B        __acle_se_entry2 ; 0xcc

...

    entry3
        0x00004020:    e97fe97f    ....    SG
        0x00004024:    f7fcb864    ..d.    B        __acle_se_entry3 ; 0xf0
    entry4
        0x00004028:    e97fe97f    ....    SG
        0x0000402c:    f7fcb87a    ..z.    B        __acle_se_entry4 ; 0x124

...
```

Related concepts

3.6.6 Generation of secure gateway veneers on page 3-56

7.2.6 Restrictions on placing __at sections on page 7-121

Related tasks

7.2.5 Placing __at sections at a specific address on page 7-120

7.2.7 Automatically placing __at sections on page 7-121

7.2.8 Manually placing __at sections on page 7-123

7.7 Reserving an empty block of memory

You can reserve an empty block of memory with a scatter file, such as the area used for the stack.

To reserve an empty block of memory, add an execution region in the scatter file and assign the `EMPTY` attribute to that region.

This section contains the following subsections:

7.7.1 Characteristics of a reserved empty block of memory

An empty block of memory that is reserved with a scatter-loading description has certain characteristics.

The block of memory does not form part of the load region, but is assigned for use at execution time. Because it is created as a dummy ZI region, the linker uses the following symbols to access it:

- `Image$$`*`region_name`*`$$ZI$$Base`.
- `Image$$`*`region_name`*`$$ZI$$Limit`.
- `Image$$`*`region_name`*`$$ZI$$Length`.

If the length is given as a negative value, the address is taken to be the end address of the region. This address must be an absolute address and not a relative one.

7.7.2 Example of reserving an empty block of memory

This example shows how to reserve and empty block of memory for stack and heap using a scatter-loading description. It also shows the related symbols that the linker generates.

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

```
LR_1 0x80000                            ; load region starts at 0x80000
{
    STACK 0x800000 EMPTY -0x10000       ; region ends at 0x800000 because of the
                                        ; negative length. The start of the region
                                        ; is calculated using the length.
    {
                                        ; Empty region for placing the stack
    }

    HEAP +0 EMPTY 0x10000               ; region starts at the end of previous
                                        ; region. End of region calculated using
                                        ; positive length
    {
                                        ; Empty region for placing the heap
    }
    …                                   ; rest of scatter-loading description
}
```

Note

The dummy ZI region that is created for an `EMPTY` execution region is not initialized to zero at runtime.

If the address is in relative (`+`*`offset`*) form and the length is negative, the linker generates an error.

The following figure shows a diagrammatic representation for this example.

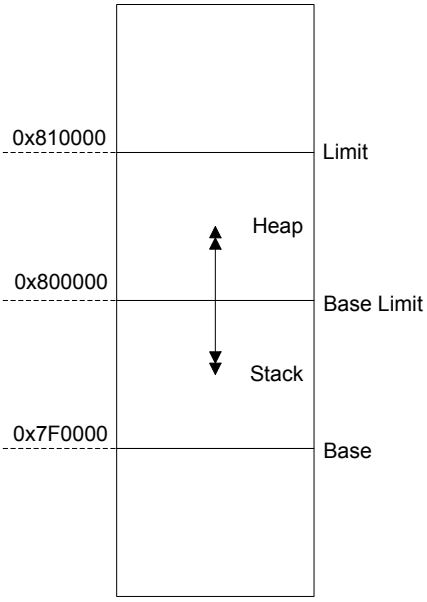

Figure 7-5 Reserving a region for the stack

In this example, the linker generates the following symbols:

```
Image$$STACK$$ZI$$Base      = 0x7f0000
Image$$STACK$$ZI$$Limit     = 0x800000
Image$$STACK$$ZI$$Length    = 0x10000
Image$$HEAP$$ZI$$Base       = 0x800000
Image$$HEAP$$ZI$$Limit      = 0x810000
Image$$HEAP$$ZI$$Length     = 0x10000
```

Note

The `EMPTY` attribute applies only to an execution region. The linker generates a warning and ignores an `EMPTY` attribute that is used in a load region definition.

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

Related concepts

8.4 Execution region descriptions on page 8-173

Related reference

6.3.2 Image$$ execution region symbols on page 6-90

8.4.3 Execution region attributes on page 8-175

7.8 Placement of Arm® C and C++ library code

You can place code from the Arm standard C and C++ libraries using a scatter file.

Use `*armlib*` or `*libcxx*` so that the linker can resolve library naming in your scatter file.

Some Arm C and C++ library sections must be placed in a root region, for example `__main.o`, `__scatter*.o`, `__dc*.o`, and `*Region$$Table`. This list can change between releases. The linker can place all these sections automatically in a future-proof way with `InRoot$$Sections`.

Note

For AArch64, `__rtentry*.o` is moved to a root region.

This section contains the following subsections:

- *7.8.1 Placing code in a root region* on page 7-144.
- *7.8.2 Placing Arm® C library code* on page 7-144.
- *7.8.3 Placing Arm® C++ library code* on page 7-145.

7.8.1 Placing code in a root region

Some code must always be placed in a root region. You do this in a similar way to placing a named section.

To place all sections that must be in a root region, use the section selector `InRoot$$Sections`. For example :

```
ROM_LOAD 0x0000 0x4000
{
  ROM_EXEC 0x0000 0x4000      ; root region at 0x0
  {
    vectors.o (Vect, +FIRST)  ; Vector table
    * (InRoot$$Sections)      ; All library sections that must be in a
                              ; root region, for example, __main.o,
                              ; __scatter*.o, __dc*.o, and *Region$$Table
  }
  RAM 0x10000 0x8000
  {
    * (+RO, +RW, +ZI)         ; all other sections
  }
}
```

Related concepts

7.2.1 Effect of the ABSOLUTE attribute on a root region on page 7-111

7.2.2 Effect of the FIXED attribute on a root region on page 7-113

7.2 Root region and the initial entry point on page 7-111

Related tasks

7.8.2 Placing Arm® C library code on page 7-144

7.8.3 Placing Arm® C++ library code on page 7-145

7.8.2 Placing Arm® C library code

You can place C library code using a scatter file.

To place C library code, specify the library path and library name as the module selector. You can use wildcard characters if required. For example:

```
LR1 0x0
{
    ROM1 0
    {
        * (InRoot$$Sections)
        * (+RO)
    }
    ROM2 0x1000
    {
        *armlib/c_* (+RO)                  ; all Arm-supplied C library functions
    }
```

```
    RAM1 0x3000
    {
        *armlib* (+RO)                      ; all other Arm-supplied library code
                                            ; for example, floating-point libraries
    }
    RAM2 0x4000
    {
        * (+RW, +ZI)
    }
}
```

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

Related tasks

7.8.1 Placing code in a root region on page 7-144

7.8.3 Placing Arm® C++ library code on page 7-145

Related information

C and C++ library naming conventions

7.8.3 Placing Arm® C++ library code

You can place C++ library code using a scatter file.

To place C++ library code, specify the library path and library name as the module selector. You can use wildcard characters if required.

Procedure

1. Create the following C++ program, `foo.cpp`:

```
#include <iostream>

using namespace std;

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

2. To place the C++ library code, define the following scatter file, `scatter.scat`:

```
LR 0x8000
{
    ER1 +0
    {
        *armlib*(+RO)
    }
    ER2 +0
    {
        *libcxx*(+RO)
    }
    ER3 +0
    {
        *(+RO)

        ; All .ARM.exidx* sections must be coalesced into a single contiguous
        ; .ARM.exidx section because the unwinder references linker-generated
        ; Base and Limit symbols for this section.
        *(0x70000001)  ; SHT_ARM_EXIDX sections

        ; All .init_array sections must be coalesced into a single contiguous
        ; .init_array section because the initialization code references
        ; linker-generated Base and Limit for this section.
        *(.init_array)
    }
    ER4 +0
    {
        *(+RW,+ZI)
    }
}
```

The name `*armlib*` matches *`install_directory`*`\lib\armlib`, indicating the Arm C library files that are located in the `armlib` directory.

The name `*libcxx*` matches *`install_directory`*`\lib\libcxx`, indicating the C++ library files that are located in the `libcxx` directory.

3. Compile and link the sources:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -c foo.cpp
armclang --target=arm-arm-none-eabi -march=armv8-a -c main.c
armlink --scatter=scatter.scat --map main.o foo.o -o foo.axf
```

The `--map` option displays the memory map of the image.

Related tasks

7.8.1 Placing code in a root region on page 7-144

7.8.2 Placing Arm® C library code on page 7-144

Related information

C and C++ library naming conventions

7.9 Aligning regions to page boundaries

You can produce an ELF file with each execution region starting at a page boundary.

The linker provides the following built-in functions to help create load and execution regions on page boundaries:

- `AlignExpr`, to specify an address expression.
- `GetPageSize`, to obtain the page size for use in `AlignExpr`. If you use `GetPageSize`, you must also use the `--paged` linker command-line option.
- `SizeOfHeaders()`, to return the size of the ELF header and Program Header table.

Note

- Alignment on an execution region causes both the load address and execution address to be aligned.
- The default page size is `0x8000`. To change the page size, specify the `--pagesize` linker command-line option.

To produce an ELF file with each execution region starting on a new page, and with code starting on the next page boundary after the header information:

```
LR1 0x0 + SizeOfHeaders()
{
    ER_RO +0
    {
        *(+RO)
    }
    ER_RW AlignExpr(+0, GetPageSize())
    {
        *(+RW)
    }
    ER_ZI AlignExpr(+0, GetPageSize())
    {
        *(+ZI)
    }
}
```

If you set up your ELF file in this way, then you can memory-map it onto an operating system in such a way that:

- RO and RW data can be given different memory protections, because they are placed in separate pages.
- The load address everything expects to run at is related to its offset in the ELF file by specifying `SizeOfHeaders()` for the first load region.

Related concepts

3.4 Linker support for creating demand-paged files on page 3-51

8.6 Expression evaluation in scatter files on page 8-186

7.12 Example of using expression evaluation in a scatter file to avoid padding on page 7-151

8.6.9 Example of aligning a base address in execution space but still tightly packed in load space on page 8-192

Related tasks

7.10 Aligning execution regions and input sections on page 7-148

Related reference

8.6.6 AlignExpr(expr, align) function on page 8-190

8.6.7 GetPageSize() function on page 8-191

11.99 --pagesize=pagesize on page 11-328

8.3.3 Load region attributes on page 8-169

8.4.3 Execution region attributes on page 8-175

11.98 --paged on page 11-327

7.10 Aligning execution regions and input sections

There are situations when you want to align code and data sections. How you deal with them depends on whether you have access to the source code.

Aligning when it is convenient for you to modify the source and recompile

When it is convenient for you to modify the original source code, you can align at compile time with the `__align(n)` keyword, for example.

Aligning when it is not convenient for you to modify the source and recompile

It might not be convenient for you to modify the source code for various reasons. For example, your build process might link the same object file into several images with different alignment requirements.

When it is not convenient for you to modify the source code, then you must use the following alignment specifiers in a scatter file:

`ALIGNALL`

Increases the section alignment of all the sections in an execution region, for example:

```
ER_DATA … ALIGNALL 8
{
    … ;selectors
}
```

`OVERALIGN`

Increases the alignment of a specific section, for example:

```
ER_DATA …
{
    *.o(.bar, OVERALIGN 8)
    … ;selectors
}
```

Related concepts

8.5 Input section descriptions on page 8-181

Related tasks

7.9 Aligning regions to page boundaries on page 7-147

Related reference

8.4.3 Execution region attributes on page 8-175

7.11 Preprocessing a scatter file

You can pass a scatter file through a C preprocessor. This permits access to all the features of the C preprocessor.

Use the first line in the scatter file to specify a preprocessor command that the linker invokes to process the file. The command is of the form:

```
#! preprocessor [preprocessor_flags]
```

Most typically the command is `#! armclang --target=arm-arm-none-eabi -march=armv8-a -E -x c`. This passes the scatter file through the `armclang` preprocessor.

You can:

- Add preprocessing directives to the top of the scatter file.
- Use simple expression evaluation in the scatter file.

For example, a scatter file, `file.scat`, might contain:

```
#! armclang --target=arm-arm-none-eabi -march=armv8-a -E -x c
#define ADDRESS 0x20000000
#include "include_file_1.h"

LR1 ADDRESS
{
    …
}
```

The linker parses the preprocessed scatter file and treats the directives as comments.

You can also use the `--predefine` command-line option to assign values to constants. For this example:

1. Modify `file.scat` to delete the directive `#define ADDRESS 0x20000000`.
2. Specify the command:

```
armlink --predefine="-DADDRESS=0x20000000" --scatter=file.scat
```

This section contains the following subsections:

- *7.11.1 Default behavior for armclang -E in a scatter file* on page 7-149.
- *7.11.2 Using other preprocessors in a scatter file* on page 7-149.

7.11.1 Default behavior for `armclang` -E in a scatter file

`armlink` behaves in the same way as `armclang` when invoking other Arm tools.

`armlink` searches for the `armclang` binary in the following order:

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

`armlink` invokes `armclang` with the `-I`*`scatter_file_path`* option so that any relative `#includes` work. The linker only adds this option if the full name of the preprocessor tool given is `armclang` or `armclang.exe`. This means that if an absolute path or a relative path is given, the linker does not give the `-I`*`scatter_file_path`* option to the preprocessor. This also happens with the `--cpu` option.

On Windows, `.exe` suffixes are handled, so `armclang.exe` is considered the same as `armclang`. Executable names are case insensitive, so `ARMCLANG` is considered the same as `armclang`. The portable way to write scatter file preprocessing lines is to use correct capitalization and omit the `.exe` suffix.

7.11.2 Using other preprocessors in a scatter file

You must ensure that the preprocessing command line is appropriate for execution on the host system.

This means:

- The string must be correctly quoted for the host system. The portable way to do this is to use double-quotes.
- Single quotes and escaped characters are not supported and might not function correctly.
- The use of a double-quote character in a path name is not supported and might not work.

These rules also apply to any strings passed with the `--predefine` option.

All preprocessor executables must accept the `-o` *`file`* option to mean output to file and accept the input as a filename argument on the command line. These options are automatically added to the user command line by `armlink`. Any options to redirect preprocessing output in the user-specified command line are not supported.

Related concepts

8.6 Expression evaluation in scatter files on page 8-186

Related reference

11.106 --predefine="string" on page 11-336

11.120 --scatter=filename on page 11-350

7.12 Example of using expression evaluation in a scatter file to avoid padding

This example shows how to use expression evaluation in a scatter file to avoid padding.

Using certain scatter-loading attributes in a scatter file can result in a large amount of padding in the image.

To remove the padding caused by the `ALIGN`, `ALIGNALL`, and `FIXED` attributes, use expression evaluation to specify the start address of a load region and execution region. The built-in function `AlignExpr` is available to help you specify address expressions.

Example

The following scatter file produces an image with padding:

```
LR1 0x4000
{
    ER1 +0 ALIGN 0x8000
    {
        …
    }
}
```

In this example, the `ALIGN` keyword causes `ER1` to be aligned to a `0x8000` boundary in both the load and the execution view. To align in the load view, the linker must insert `0x4000` bytes of padding.

The following scatter file produces an image without padding:

```
LR1 0x4000
{
    ER1 AlignExpr(+0, 0x8000)
    {
        …
    }
}
```

Using `AlignExpr` the result of `+0` is aligned to a `0x8000` boundary. This creates an execution region with a load address of `0x4000` but an Execution Address of `0x8000`.

Related concepts

8.6.9 Example of aligning a base address in execution space but still tightly packed in load space on page 8-192

Related reference

8.6.6 AlignExpr(expr, align) function on page 8-190

8.4.3 Execution region attributes on page 8-175

7.13 Equivalent scatter-loading descriptions for simple images

Although you can use command-line options to scatter-load simple images, you can also use a scatter file.

This section contains the following subsections:

7.13.1 Command-line options for creating simple images

The command-line options `--reloc`, `--ro_base`, `--rw_base`, `--ropi`, `--rwpi`, `--split`, and `--xo_base` create the simple image types.

The simple image types are:

- Type 1 image, one load region and contiguous execution regions.
- Type 2 image, one load region and non-contiguous execution regions.
- Type 3 image, two load regions and non-contiguous execution regions.

You can create the same image types by using the `--scatter` command-line option and a file containing one of the corresponding scatter-loading descriptions.

Note

The option `--reloc` is not supported for AArch64 state.

Related concepts

7.13.2 Type 1 image, one load region and contiguous execution regions on page 7-152

8.3 Load region descriptions on page 8-167

7.13.3 Type 2 image, one load region and non-contiguous execution regions on page 7-154

7.13.4 Type 3 image, multiple load regions and non-contiguous execution regions on page 7-155

Related reference

11.111 --reloc on page 11-341

11.114 --ro_base=address on page 11-344

11.115 --ropi on page 11-345

11.117 --rw_base=address on page 11-347

11.118 --rwpi on page 11-348

11.120 --scatter=filename on page 11-350

11.127 --split on page 11-359

11.157 --xo_base=address on page 11-389

8.3.3 Load region attributes on page 8-169

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

A Type 1 image consists of a single load region in the load view and up to four execution regions in the execution view. The execution regions are placed contiguously in the memory map.

By default, the ER_RO, ER_RW, and ER_ZI execution regions are present. If an image contains any *execute-only* (XO) sections, then an ER_XO execution region is also present.

`--ro_base` *`address`* specifies the load and execution address of the region containing the RO output section. The following example shows the scatter-loading description equivalent to using `--ro_base 0x040000`:

```
LR_1 0x040000     ; Define the load region name as LR_1, the region starts at 0x040000.
{
```

```
    ER_RO +0      ; First execution region is called ER_RO, region starts at end of
                  ; previous region. Because there is no previous region, the
                  ; address is 0x040000.
    {
        * (+RO)   ; All RO sections go into this region, they are placed
                  ; consecutively.
    }
    ER_RW +0      ; Second execution region is called ER_RW, the region starts at the
                  ; end of the previous region.
                  ; The address is 0x040000 + size of ER_RO region.
    {
        * (+RW)   ; All RW sections go into this region, they are placed
                  ; consecutively.
    }
    ER_ZI +0      ; Last execution region is called ER_ZI, the region starts at the
                  ; end of the previous region at 0x040000 + the size of the ER_RO
                  ; regions + the size of the ER_RW regions.
    {
        * (+ZI)   ; All ZI sections are placed consecutively here.
    }
}
```

In this example:

- This description creates an image with one load region called `LR_1` that has a load address of `0x040000`.
- The image has three execution regions, named `ER_RO`, `ER_RW`, and `ER_ZI`, that contain the RO, RW, and ZI output sections respectively. RO and RW are root regions. ZI is created dynamically at runtime. The execution address of `ER_RO` is `0x040000`. All three execution regions are placed contiguously in the memory map by using the *`+offset`* form of the base designator for the execution region description. This enables an execution region to be placed immediately following the end of the preceding execution region.

Use the `--reloc` option to make relocatable images. Used on its own, `--reloc` makes an image similar to simple type 1, but the single load region has the `RELOC` attribute.

Note

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

ROPI example variant (AArch32 only)

In this variant, the execution regions are placed contiguously in the memory map. However, `--ropi` marks the load and execution regions containing the RO output section as position-independent.

The following example shows the scatter-loading description equivalent to using `--ro_base 0x010000 --ropi`:

```
LR_1 0x010000 PI        ; The first load region is at 0x010000.
{
    ER_RO +0            ; The PI attribute is inherited from parent.
                        ; The default execution address is 0x010000, but the code
                        ; can be moved.
    {
        * (+RO)         ; All the RO sections go here.
    }
    ER_RW +0 ABSOLUTE   ; PI attribute is overridden by ABSOLUTE.
    {
        * (+RW)         ; The RW sections are placed next. They cannot be moved.
    }
    ER_ZI +0            ; ER_ZI region placed after ER_RW region.
    {
        * (+ZI)         ; All the ZI sections are placed consecutively here.
    }
}
```

`ER_RO`, the RO execution region, inherits the `PI` attribute from the load region `LR_1`. The next execution region, `ER_RW`, is marked as `ABSOLUTE` and uses the *`+offset`* form of base designator. This prevents

`ER_RW` from inheriting the `PI` attribute from `ER_RO`. Also, because the `ER_ZI` region has an offset of `+0`, it inherits the `ABSOLUTE` attribute from the `ER_RW` region.

Note

If an image contains execute-only sections, ROPI is not supported. If you use `--ropi` to link such an image, `armlink` gives an error.

Note

XO memory is supported only for Armv7-M and Armv8-M architectures.

Related concepts

7.13.1 Command-line options for creating simple images on page 7-152

8.3 Load region descriptions on page 8-167

8.3.6 Considerations when using a relative address +offset for a load region on page 8-171

8.4.5 Considerations when using a relative address +offset for execution regions on page 8-179

Related reference

11.114 --ro_base=address on page 11-344

11.115 --ropi on page 11-345

8.3.3 Load region attributes on page 8-169

11.111 --reloc on page 11-341

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

A Type 2 image consists of a single load region in the load view and three execution regions in the execution view. It is similar to images of Type 1 except that the RW execution region is not contiguous with the RO execution region.

`--ro_base=`*`address`* specifies the load and execution address of the region containing the RO output section. `--rw_base=`*`address`* specifies the execution address for the RW execution region.

For images that contain *execute-only* (XO) sections, the XO execution region is placed at the address specified by `--ro_base`. The RO execution region is placed contiguously and immediately after the XO execution region.

If you use `--xo_base` *`address`*, then the XO execution region is placed in a separate load region at the specified address.

Note

XO memory is supported only for Armv7-M and Armv8-M architectures.

Example for single load region and multiple execution regions

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

```
LR_1 0x010000        ; Defines the load region name as LR_1
{
    ER_RO +0         ; The first execution region is called ER_RO and starts at end
                     ; of previous region. Because there is no previous region, the
                     ; address is 0x010000.
    {
        * (+RO)      ; All RO sections are placed consecutively into this region.
    }
    ER_RW 0x040000   ; Second execution region is called ER_RW and starts at 0x040000.
    {
        * (+RW)      ; All RW sections are placed consecutively into this region.
    }
    ER_ZI +0         ; The last execution region is called ER_ZI.
                     ; The address is 0x040000 + size of ER_RW region.
    {
        * (+ZI)      ; All ZI sections are placed consecutively here.
```

```
    }
}
```

In this example:

- This description creates an image with one load region, named `LR_1`, with a load address of `0x010000`.
- The image has three execution regions, named `ER_RO`, `ER_RW`, and `ER_ZI`, that contain the RO, RW, and ZI output sections respectively. The RO region is a root region. The execution address of `ER_RO` is `0x010000`.
- The `ER_RW` execution region is not contiguous with `ER_RO`. Its execution address is `0x040000`.
- The `ER_ZI` execution region is placed immediately following the end of the preceding execution region, `ER_RW`.

RWPI example variant (AArch32 only)

This is similar to images of Type 2 with `--rw_base` where the RW execution region is separate from the RO execution region. However, `--rwpi` marks the execution regions containing the RW output section as position-independent.

The following example shows the scatter-loading description equivalent to using `--ro_base=0x010000 --rw_base=0x018000 --rwpi`:

```
LR_1 0x010000           ; The first load region is at 0x010000.
{
    ER_RO +0            ; Default ABSOLUTE attribute is inherited from parent.
                        ; The execution address is 0x010000. The code and RO data
                        ; cannot be moved.
    {
        * (+RO)         ; All the RO sections go here.
    }
    ER_RW 0x018000 PI   ; PI attribute overrides ABSOLUTE
    {
        * (+RW)         ; The RW sections are placed at 0x018000 and they can be
                        ; moved.
    }
    ER_ZI +0            ; ER_ZI region placed after ER_RW region.
    {
        * (+ZI)         ; All the ZI sections are placed consecutively here.
    }
}
```

`ER_RO`, the RO execution region, inherits the `ABSOLUTE` attribute from the load region `LR_1`. The next execution region, `ER_RW`, is marked as `PI`. Also, because the `ER_ZI` region has an offset of `+0`, it inherits the `PI` attribute from the `ER_RW` region.

Similar scatter-loading descriptions can also be written to correspond to the usage of other combinations of `--ropi` and `--rwpi` with Type 2 and Type 3 images.

Related concepts

8.3 Load region descriptions on page 8-167

8.3.6 Considerations when using a relative address +offset for a load region on page 8-171

8.4.5 Considerations when using a relative address +offset for execution regions on page 8-179

Related reference

11.114 --ro_base=address on page 11-344

11.117 --rw_base=address on page 11-347

11.157 --xo_base=address on page 11-389

8.3.3 Load region attributes on page 8-169

7.13.4 Type 3 image, multiple load regions and non-contiguous execution regions

A Type 3 image consists of multiple load regions in load view and multiple execution regions in execution view. They are similar to images of Type 2 except that the single load region in Type 2 is now split into multiple load regions.

You can relocate and split load regions using the following linker options:

--reloc
The combination `--reloc --split` makes an image similar to simple Type 3, but the two load regions now have the `RELOC` attribute.

--ro_base=*address1*
Specifies the load and execution address of the region containing the RO output section.

--rw_base=*address2*
Specifies the load and execution address for the region containing the RW output section.

--xo_base=*address3*
Specifies the load and execution address for the region containing the *execute-only* (XO) output section, if present.

--split
Splits the default single load region that contains the RO and RW output sections into two load regions. One load region contains the RO output section and one contains the RW output section.

Note

For images containing XO sections, and if `--xo_base` is not used, an XO execution region is placed at the address specified by `--ro_base`. The RO execution region is placed immediately after the XO region.

Note

XO memory is supported only for Armv7-M and Armv8-M architectures.

Example for multiple load regions

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

```
LR_1 0x010000     ; The first load region is at 0x010000.
{
    ER_RO +0      ; The address is 0x010000.
    {
        * (+RO)
    }
}
LR_2 0x040000     ; The second load region is at 0x040000.
{
    ER_RW +0      ; The address is 0x040000.
    {
        * (+RW)   ; All RW sections are placed consecutively into this region.
    }
    ER_ZI +0      ; The address is 0x040000 + size of ER_RW region.
    {
        * (+ZI)   ; All ZI sections are placed consecutively into this region.
    }
}
```

In this example:

- This description creates an image with two load regions, named `LR_1` and `LR_2`, that have load addresses `0x010000` and `0x040000`.
- The image has three execution regions, named `ER_RO`, `ER_RW` and `ER_ZI`, that contain the RO, RW, and ZI output sections respectively. The execution address of `ER_RO` is `0x010000`.
- The `ER_RW` execution region is not contiguous with `ER_RO`, because its execution address is `0x040000`.
- The `ER_ZI` execution region is placed immediately after `ER_RW`.

Example for multiple load regions with an XO region

The following example shows the scatter-loading description equivalent to using `--ro_base=0x010000 --rw_base=0x040000 --split` when an object file has XO sections:

```
LR_1 0x010000     ; The first load region is at 0x010000.
{
```

```
    ER_XO +0      ; The address is 0x010000.
    {
        * (+XO)
    }
    ER_RO +0      ; The address is 0x010000 + size of ER_XO region.
    {
        * (+RO)
    }
}
LR_2 0x040000     ; The second load region is at 0x040000.
{
    ER_RW +0      ; The address is 0x040000.
    {
        * (+RW)   ; All RW sections are placed consecutively into this region.
    }
    ER_ZI +0      ; The address is 0x040000 + size of ER_RW region.
    {
        * (+ZI)   ; All ZI sections are placed consecutively into this region.
    }
}
```

In this example:

- This description creates an image with two load regions, named `LR_1` and `LR_2`, that have load addresses `0x010000` and `0x040000`.
- The image has four execution regions, named `ER_XO`, `ER_RO`, `ER_RW` and `ER_ZI`, that contain the XO, RO, RW, and ZI output sections respectively. The execution address of `ER_XO` is placed at the address specified by `--ro_base`, `0x010000`. `ER_RO` is placed immediately after `ER_XO`.
- The `ER_RW` execution region is not contiguous with `ER_RO`, because its execution address is `0x040000`.
- The `ER_ZI` execution region is placed immediately after `ER_RW`.

Note

If you also specify `--xo_base`, then the `ER_XO` execution region is placed in a load region separate from the `ER_RO` execution region, at the specified address.

Relocatable load regions example variant

This Type 3 image also consists of two load regions in load view and three execution regions in execution view. However, `--reloc` specifies that the two load regions now have the `RELOC` attribute.

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

```
LR_1 0x010000 RELOC
{
     ER_RO + 0
     {
         * (+RO)
     }
}
LR2 0x040000 RELOC
{
     ER_RW + 0
     {
         * (+RW)
     }
     ER_ZI +0
     {
         * (+ZI)
     }
}
```

Related concepts

8.3 Load region descriptions on page 8-167

8.3.6 Considerations when using a relative address +offset for a load region on page 8-171

8.4.5 Considerations when using a relative address +offset for execution regions on page 8-179

8.3.4 Inheritance rules for load region address attributes on page 8-170

8.3.5 Inheritance rules for the RELOC address attribute on page 8-171

8.4.4 Inheritance rules for execution region address attributes on page 8-178

Related reference

11.111 --reloc on page 11-341
11.114 --ro_base=address on page 11-344
11.117 --rw_base=address on page 11-347
11.127 --split on page 11-359
11.157 --xo_base=address on page 11-389
8.3.3 Load region attributes on page 8-169

7.14 How the linker resolves multiple matches when processing scatter files

An input section must be unique. In the case of multiple matches, the linker attempts to assign the input section to a region based on the attributes of the input section description.

The linker assignment of the input section is based on a *module_select_pattern* and *input_section_selector* pair that is the most specific. However, if a unique match cannot be found, the linker faults the scatter-loading description.

The following variables describe how the linker matches multiple input sections:

- *m1* and *m2* represent module selector patterns.
- *s1* and *s2* represent input section selectors.

For example, if input section A matches *m1,s1* for execution region R1, and A matches *m2,s2* for execution region R2, the linker:

- Assigns A to R1 if *m1,s1* is more specific than *m2,s2*.
- Assigns A to R2 if *m2,s2* is more specific than *m1,s1*.
- Diagnoses the scatter-loading description as faulty if *m1,s1* is not more specific than *m2,s2* and *m2,s2* is not more specific than *m1,s1*.

`armlink` uses the following strategy to determine the most specific *module_select_pattern*, *input_section_selector* pair:

Resolving the priority of two module_selector, section_selector pairs m1, s1 and m2, s2

The strategy starts with two *module_select_pattern, input_section_selector* pairs. *m1,s1* is more specific than *m2,s2* only if any of the following are true:

1. *s1* is either a literal input section name, that is it contains no pattern characters, or a section type and *s2* matches input section attributes.
2. *m1* is more specific than *m2*.
3. *s1* is more specific than *s2*.

The conditions are tested in order so condition 1 takes precedence over condition 2 and 3, and condition 2 takes precedence over condition 3.

Resolving the priority of two module selectors m1 and m2 in isolation

For the module selector patterns, *m1* is more specific than *m2* if the text string *m1* matches pattern *m2* and the text string *m2* does not match pattern *m1*.

Resolving the priority of two section selectors s1 and s2 in isolation

For the input section selectors:

- If one of *s1* or *s2* matches the input section name or type and the other matches the input section attributes, *s1* and *s2* are unordered and the description is diagnosed as faulty.
- If both *s1* and *s2* match the input section name or type, the following relationships determine whether s1 is more specific than s2:
  - Section type is more specific than section name.
  - If both *s1* and *s2* match input section type, *s1* and *s2* are unordered and the description is diagnosed as faulty.
  - If *s1* and *s2* are both patterns matching section names, the same definition as for module selector patterns is used.
- If both *s1* and *s2* match input section attributes, the following relationships determine whether *s1* is more specific than *s2*s:
  - `ENTRY` is more specific than `RO-CODE`, `RO-DATA`, `RW-CODE`, or `RW-DATA`.
  - `RO-CODE` is more specific than `RO`.
  - `RO-DATA` is more specific than `RO`.
  - `RW-CODE` is more specific than `RW`.
  - `RW-DATA` is more specific than `RW`.
  - There are no other members of the (*s1* more specific than *s2*) relationship between section attributes.

This matching strategy has the following consequences:

- Descriptions do not depend on the order they are written in the file.
- Generally, the more specific the description of an object, the more specific the description of the input sections it contains.
- The *input_section_selector*s are not examined unless:
  - Object selection is inconclusive.
  - One selector specifies a literal input section name or a section type and the other selects by attribute. In this case, the explicit input section name or type is more specific than any attribute. This is true even if the object selector associated with the input section name is less specific than that of the attribute.

The `.ANY` module selector is available to assign any sections that cannot be resolved from the scatter-loading description.

Example

The following example shows multiple execution regions and pattern matching:

```
LR_1 0x040000
{
    ER_ROM 0x040000              ; The startup exec region address is the same
    {                            ; as the load address.
        application.o (+ENTRY)   ; The section containing the entry point from
    }                            ; the object is placed here.
    ER_RAM1 0x048000
    {
        application.o (+RO-CODE) ; Other RO code from the object goes here
    }
    ER_RAM2 0x050000
    {
        application.o (+RO-DATA) ; The RO data goes here
    }
    ER_RAM3 0x060000
    {
        application.o (+RW)      ; RW code and data go here
    }
    ER_RAM4 +0                   ; Follows on from end of ER_R3
    {
        *.o (+RO, +RW, +ZI)      ; Everything except for application.o goes here
    }
}
```

Related concepts

8.5 Input section descriptions on page 8-181

Related reference

7.4 Placement of unassigned sections on page 7-128

8.2 Syntax of a scatter file on page 8-166

8.5.2 Syntax of an input section description on page 8-181

7.15 How the linker resolves path names when processing scatter files

The linker matches wildcard patterns in scatter files against any combination of forward slashes and backslashes it finds in path names.

This might be useful where the paths are taken from environment variables or multiple sources, or where you want to use the same scatter file to build on Windows or Unix platforms.

Note

Use forward slashes in path names to ensure they are understood on Windows and Unix platforms.

Related reference

8.2 Syntax of a scatter file on page 8-166

7.16 Scatter file to ELF mapping

Shows how scatter file components map onto ELF.

ELF executable files contain segments:

- A load region is represented by an ELF program segment with type PT_LOAD.
- An execution region is represented by one or more of the following ELF sections:
  - XO.
  - RO.
  - RW.
  - ZI.

Note

If XO and RO are mixed within an execution region, that execution region is treated as RO.

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

```
LOAD 0x8000
{
    EXEC_ROM +0
    {
        *(+RO)
    }
    RAM +0
    {
        *(+RW,+ZI)
    }
    HEAP +0x100 EMPTY 0x100
    {
    }
    STACK +0 EMPTY 0x400
    {
    }
}
```

This scatter file creates a single program segment with type PT_LOAD for the load region with address `0x8000`.

A single output section with type SHT_PROGBITS is created to represent the contents of EXEC_ROM. Two output sections are created to represent RAM. The first has a type SHT_PROGBITS and contains the initialized read/write data. The second has a type of SHT_NOBITS and describes the zero-initialized data.

The heap and stack are described in the ELF file by SHT_NOBITS sections.

Enter the following `fromelf` command to see the scatter-loaded sections in the image:

```
fromelf --text -v my_image.axf
```

To display the symbol table, enter the command:

```
fromelf --text -s -v my_image.axf
```

The following is an example of the `fromelf` output showing the `LOAD`, `EXEC_ROM`, `RAM`, `HEAP`, and `STACK` sections:

```
…
==============================================================================
** Program header #0
    Type          : PT_LOAD (1)
    File Offset   : 52 (0x34)
    Virtual Addr  : 0x00008000
    Physical Addr : 0x00008000
    Size in file  : 764 bytes (0x2fc)
    Size in memory: 2140 bytes (0x85c)
    Flags         : PF_X + PF_W + PF_R + PF_ARM_ENTRY (0x80000007)
    Alignment     : 4
==============================================================================
** Section #1
```

```
    Name        : EXEC_ROM
…
    Addr        : 0x00008000
    File Offset : 52 (0x34)
    Size        : 740 bytes (0x2e4)
…
====================================
** Section #2
    Name        : RAM
…
    Addr        : 0x000082e4
    File Offset : 792 (0x318)
    Size        : 20 bytes (0x14)
…
====================================
** Section #3
    Name        : RAM
…
    Addr        : 0x000082f8
    File Offset : 812 (0x32c)
    Size        : 96 bytes (0x60)
…
====================================
** Section #4
    Name        : HEAP
…
    Addr        : 0x00008458
    File Offset : 812 (0x32c)
    Size        : 256 bytes (0x100)
…
====================================
** Section #5
    Name        : STACK
…
    Addr        : 0x00008558
    File Offset : 812 (0x32c)
    Size        : 1024 bytes (0x400)
…
```

Related concepts

7.1.1 Overview of scatter-loading on page 7-105

7.1.6 Scatter-loading images with a simple memory map on page 7-108

Chapter 8
Scatter File Syntax

Describes the format of scatter files.

It contains the following sections:

8.1 BNF notation used in scatter-loading description syntax

Scatter-loading description syntax uses standard BNF notation.

The following table summarizes the *Backus-Naur Form* (BNF) symbols that are used for describing the syntax of scatter-loading descriptions.

Table 8-1 BNF notation

Symbol	Description
"	Quotation marks indicate that a character that is normally part of the BNF syntax is used as a literal character in the definition. The definition `B"+"C`, for example, can only be replaced by the pattern `B+C`. The definition `B+C` can be replaced by, for example, patterns `BC`, `BBC`, or `BBBC`.
A ::= *B*	Defines *A* as *B*. For example, `A::= B"+" | C` means that *A* is equivalent to either `B+` or `C`. The `::=` notation defines a higher level construct in terms of its components. Each component might also have a `::=` definition that defines it in terms of even simpler components. For example, `A::= B` and `B::= C | D` means that the definition *A* is equivalent to the patterns `C` or `D`.
[*A*]	Optional element *A*. For example, `A::= B[C]D` means that the definition *A* can be expanded into either `BD` or `BCD`.
A+	Element *A* can have one or more occurrences. For example, `A::= B+` means that the definition *A* can be expanded into `B`, `BB`, or `BBB`.
*A**	Element *A* can have zero or more occurrences.
A | *B*	Either element *A* or *B* can occur, but not both.
(*A B*)	Element *A* and *B* are grouped together. This is particularly useful when the | operator is used or when a complex pattern is repeated. For example, `A::=(B C)+ (D | E)` means that the definition *A* can be expanded into any of `BCD`, `BCE`, `BCBCD`, `BCBCE`, `BCBCBCD`, or `BCBCBCE`.

Related reference

8.2 Syntax of a scatter file on page 8-166

8.2 Syntax of a scatter file

A scatter file contains one or more load regions. Each load region can contain one or more execution regions.

The following figure shows the components and organization of a typical scatter file:

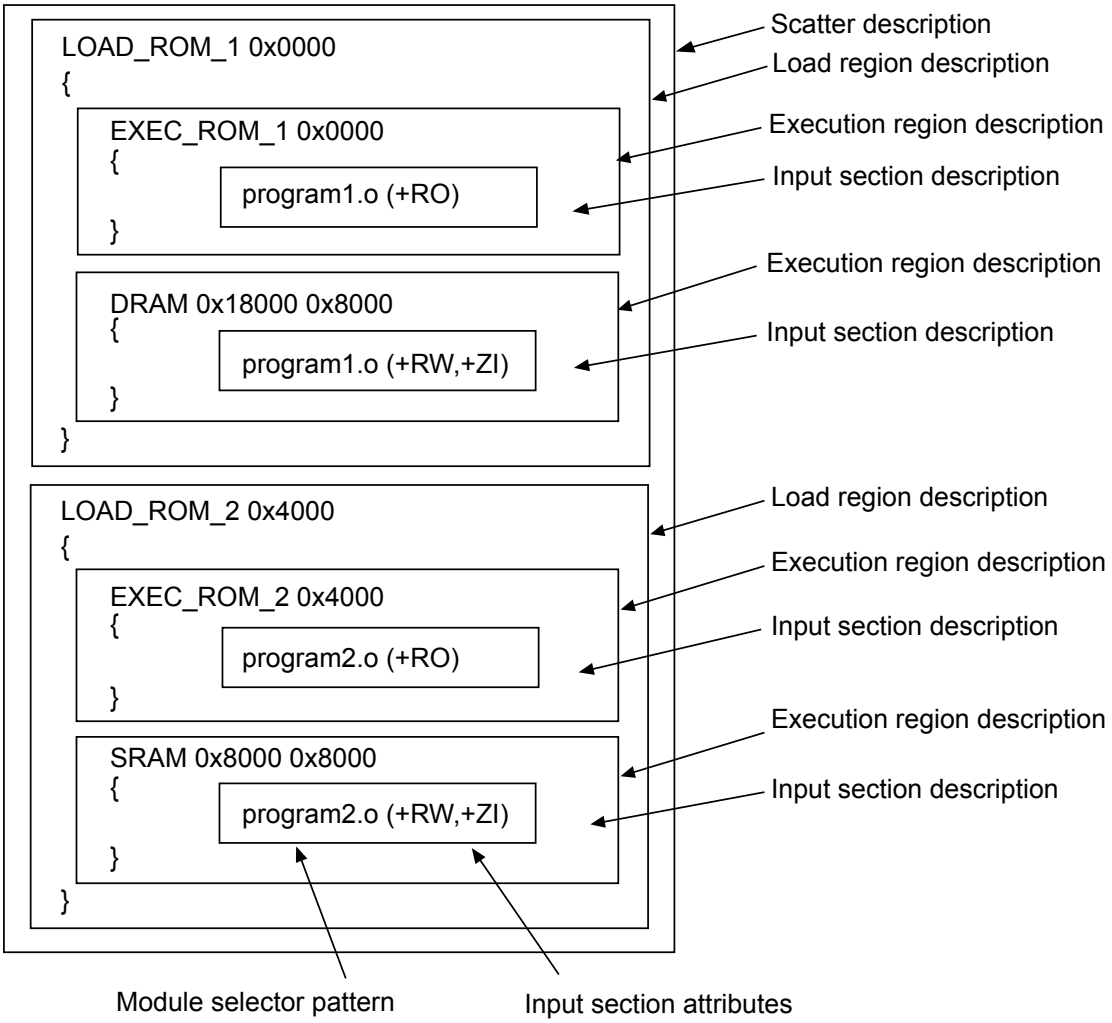

Figure 8-1 Components of a scatter file

Related concepts

8.3 Load region descriptions on page 8-167

8.4 Execution region descriptions on page 8-173

Related reference

Chapter 7 Scatter-loading Features on page 7-104

8.3 Load region descriptions

A load region description specifies the region of memory where its child execution regions are to be placed.

This section contains the following subsections:

8.3.1 Components of a load region description

The components of a load region description allow you to uniquely identify a load region and to control what parts of an ELF file are placed in that region.

A load region description has the following components:

- A name (used by the linker to identify different load regions).
- A base address (the start address for the code and data in the load view).
- Attributes that specify the properties of the load region.
- An optional maximum size specification.
- One or more execution regions.

The following figure shows an example of a typical load region description:

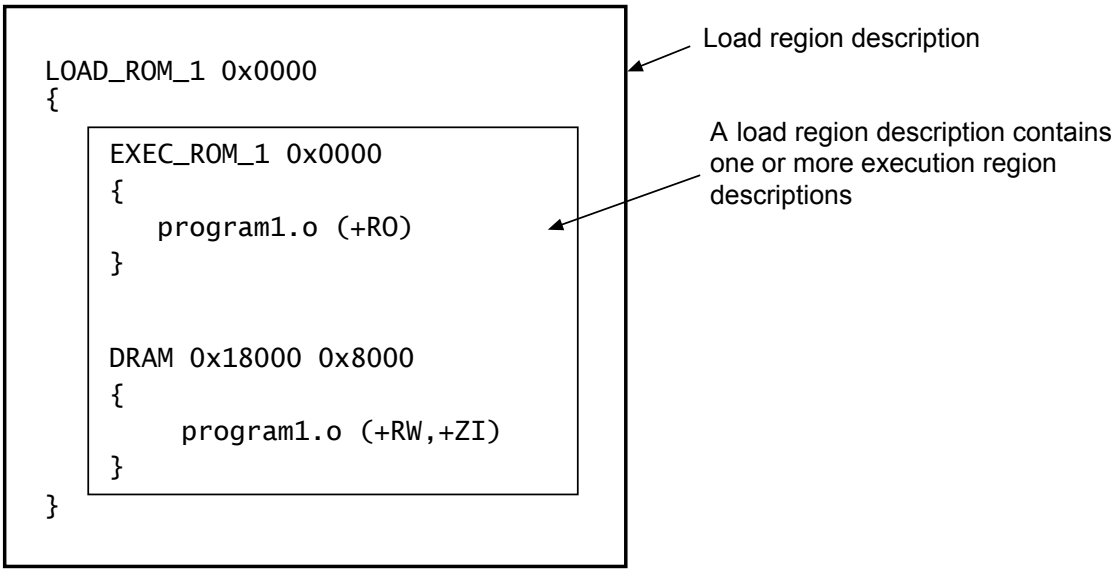

Figure 8-2 Components of a load region description

Related concepts

8.3.4 Inheritance rules for load region address attributes on page 8-170

8.3.5 Inheritance rules for the RELOC address attribute on page 8-171

8.4.4 Inheritance rules for execution region address attributes on page 8-178

8.6 Expression evaluation in scatter files on page 8-186

Related tasks

7.9 Aligning regions to page boundaries on page 7-147

Related reference

8.3.2 Syntax of a load region description on page 8-168

8.3.3 Load region attributes on page 8-169
Chapter 7 Scatter-loading Features on page 7-104

8.3.2 Syntax of a load region description

A load region can contain one or more execution region descriptions.

The syntax of a load region description, in *Backus-Naur Form* (BNF), is:

```
load_region_description ::=
  load_region_name  (base_address  | ("+" offset)) [attribute_list] [max_size]
       "{"
          execution_region_description+
       "}"
```

where:

load_region_name

Names the load region. You can use a quoted name. The name is case-sensitive only if you use any region-related linker-defined symbols.

base_address

Specifies the address where objects in the region are to be linked. ***base_address*** must satisfy the alignment constraints of the load region.

+offset

Describes a base address that is ***offset*** bytes beyond the end of the preceding load region. The value of ***offset*** must be zero modulo four. If this is the first load region, then ***+offset*** means that the base address begins ***offset*** bytes from zero.

If you use ***+offset***, then the load region might inherit certain attributes from a previous load region.

attribute_list

The attributes that specify the properties of the load region contents.

max_size

Specifies the maximum size of the load region. This is the size of the load region before any decompression or zero initialization take place. If the optional ***max_size*** value is specified, `armlink` generates an error if the region has more than ***max_size*** bytes allocated to it.

execution_region_description

Specifies the execution region name, address, and contents.

Note

The BNF definitions contain additional line returns and spaces to improve readability. They are not required in scatter-loading descriptions and are ignored if present in a scatter file.

Related concepts

8.3.5 Inheritance rules for the RELOC address attribute on page 8-171
8.3.6 Considerations when using a relative address +offset for a load region on page 8-171
8.3.4 Inheritance rules for load region address attributes on page 8-170
8.6 Expression evaluation in scatter files on page 8-186

Related reference

8.3.1 Components of a load region description on page 8-167
8.3.3 Load region attributes on page 8-169
8.1 BNF notation used in scatter-loading description syntax on page 8-165
8.2 Syntax of a scatter file on page 8-166

6.3 Region-related symbols on page 6-90

8.3.3 Load region attributes

A load region has attributes that allow you to control where parts of your image are loaded in the target memory.

The load region attributes are:

ABSOLUTE
The content is placed at a fixed address that does not change after linking. The load address of the region is specified by the base designator. This is the default, unless you use `PI` or `RELOC`.

ALIGN *alignment*

Increase the alignment constraint for the load region from 4 to *alignment*. *alignment* must be a positive power of 2. If the load region has a *base_address* then this must be *alignment* aligned. If the load region has a *+offset* then the linker aligns the calculated base address of the region to an *alignment* boundary.

This can also affect the offset in the ELF file. For example, the following causes the data for `FOO` to be written out at 4k offset into the ELF file:

```
FOO +4 ALIGN 4096
```

NOCOMPRESS
RW data compression is enabled by default. The `NOCOMPRESS` keyword enables you to specify that the contents of a load region must not be compressed in the final image.

OVERLAY

The `OVERLAY` keyword enables you to have multiple load regions at the same address. Arm tools do not provide an overlay mechanism. To use multiple load regions at the same address, you must provide your own overlay manager.

The content is placed at a fixed address that does not change after linking. The content might overlap with other regions designated as `OVERLAY` regions.

PI

This region is position independent. The content does not depend on any fixed address and might be moved after linking without any extra processing.

Note

`PI` is not supported for AArch64 state.

Note

This attribute is not supported if an image contains execute-only sections.

PROTECTED
The `PROTECTED` keyword prevents:

- Overlapping of load regions.
- Veneer sharing.
- String sharing with the `--merge` option.

RELOC

Note

- This attribute is deprecated when *Base Platform* on page 10-208 is not enabled.
- `RELOC` is not supported for AArch64 state.

This region is relocatable. The content depends on fixed addresses. Relocation information is output to enable the content to be moved to another location by another tool.

Related concepts

8.6.9 Example of aligning a base address in execution space but still tightly packed in load space on page 8-192

3.3.3 Section alignment with the linker on page 3-50

3.6.5 Reuse of veneers when scatter-loading on page 3-55

8.3.6 Considerations when using a relative address +offset for a load region on page 8-171

8.3.4 Inheritance rules for load region address attributes on page 8-170

8.3.5 Inheritance rules for the RELOC address attribute on page 8-171

3.6.2 Veneer sharing on page 3-53

3.6.4 Generation of position independent to absolute veneers on page 3-55

4.3 Optimization with RW data compression on page 4-68

Related tasks

7.9 Aligning regions to page boundaries on page 7-147

Related reference

11.89 --merge, --no_merge on page 11-318

8.3.1 Components of a load region description on page 8-167

8.3.2 Syntax of a load region description on page 8-168

8.3.4 Inheritance rules for load region address attributes

A load region can inherit the attributes of a previous load region.

For a load region to inherit the attributes of a previous load region, specify a `+offset` base address for that region. A load region cannot inherit attributes if:

- You explicitly set the attribute of that load region.
- The load region immediately before has the `OVERLAY` attribute.

You can explicitly set a load region with the `ABSOLUTE`, `PI`, `RELOC`, or `OVERLAY` address attributes.

Note

`PI` and `RELOC` are not supported for AArch64 state.

The following inheritance rules apply when no address attribute is specified:

- The `OVERLAY` attribute cannot be inherited. A region with the `OVERLAY` attribute cannot inherit.
- A base address load or execution region always defaults to `ABSOLUTE`.
- A `+offset` load region inherits the address attribute from the previous load region or `ABSOLUTE` if no previous load region exists.

Example

This example shows the inheritance rules for setting the address attributes of load regions:

```
LR1 0x8000 PI
{
    …
}
LR2 +0              ; LR2 inherits PI from LR1
{
    …
}
LR3 0x1000          ; LR3 does not inherit because it has no relative base
                      address, gets default of ABSOLUTE
{
    …
}
LR4 +0              ; LR4 inherits ABSOLUTE from LR3
{
    …
}
LR5 +0 RELOC        ; LR5 does not inherit because it explicitly sets RELOC
{
    …
```

```
}
LR6 +0 OVERLAY     ; LR6 does not inherit, an OVERLAY cannot inherit
{
    …
}
LR7 +0             ; LR7 cannot inherit OVERLAY, gets default of ABSOLUTE
{
    …
}
```

Related concepts

8.4.4 Inheritance rules for execution region address attributes on page 8-178

8.3.6 Considerations when using a relative address +offset for a load region on page 8-171

8.3.5 Inheritance rules for the RELOC address attribute on page 8-171

Related reference

8.3.1 Components of a load region description on page 8-167

8.4.1 Components of an execution region description on page 8-173

8.3.2 Syntax of a load region description on page 8-168

8.3.5 Inheritance rules for the RELOC address attribute

You can explicitly set the `RELOC` attribute for a load region. However, an execution region can only inherit the `RELOC` attribute from the parent load region.

Note

`RELOC` is not supported for AArch64 state.

Example

This example shows the inheritance rules for setting the address attributes with `RELOC`:

```
LR1 0x8000 RELOC
{
    ER1 +0 ; inherits RELOC from LR1
    {
        …
    }
    ER2 +0 ; inherits RELOC from ER1
    {
        …
    }
    ER3 +0 RELOC ; Error cannot explicitly set RELOC on an execution region
    {
        …
    }
}
```

Related concepts

10.1 Restrictions on the use of scatter files with the Base Platform model on page 10-209

8.3.4 Inheritance rules for load region address attributes on page 8-170

8.4.4 Inheritance rules for execution region address attributes on page 8-178

8.4.5 Considerations when using a relative address +offset for execution regions on page 8-179

8.3.6 Considerations when using a relative address +offset for a load region on page 8-171

2.5 Base Platform linking model on page 2-30

Related reference

8.3.1 Components of a load region description on page 8-167

8.3.2 Syntax of a load region description on page 8-168

8.4.1 Components of an execution region description on page 8-173

8.3.6 Considerations when using a relative address +offset for a load region

There are some considerations to be aware of when using a relative address for a load region.

When using *+offset* to specify a load region base address:

- If the *+offset* load region LR2 follows a load region LR1 containing ZI data, then LR2 overlaps the ZI data. To fix this, use the `ImageLimit()` function to specify the base address of LR2.
- A *+offset* load region LR2 inherits the attributes of the load region LR1 immediately before it, unless:
  — LR1 has the `OVERLAY` attribute.
  — LR2 has an explicit attribute set.

  If a load region is unable to inherit an attribute, then it gets the attribute `ABSOLUTE`.
- A gap might exist in a ROM image between a *+offset* load region and a preceding region when the preceding region has RW data compression applied. This is because the linker calculates the *+offset* based on the uncompressed size of the preceding region. However, this gap disappears when the RW data is decompressed at load time.

Related concepts

8.3.4 Inheritance rules for load region address attributes on page 8-170

8.6.3 Execution address built-in functions for use in scatter files on page 8-187

Related reference

8.2 Syntax of a scatter file on page 8-166

8.4 Execution region descriptions

An execution region description specifies the region of memory where parts of your image are to be placed at run-time.

This section contains the following subsections:

8.4.1 Components of an execution region description

The components of an execution region description allow you to uniquely identify each execution region and its position in the parent load region, and to control what parts of an ELF file are placed in that execution region.

An execution region description has the following components:

- A name (used by the linker to identify different execution regions).
- A base address (either absolute or relative).
- Attributes that specify the properties of the execution region.
- An optional maximum size specification.
- One or more input section descriptions (the modules placed into this execution region).

The following figure shows the components of a typical execution region description:

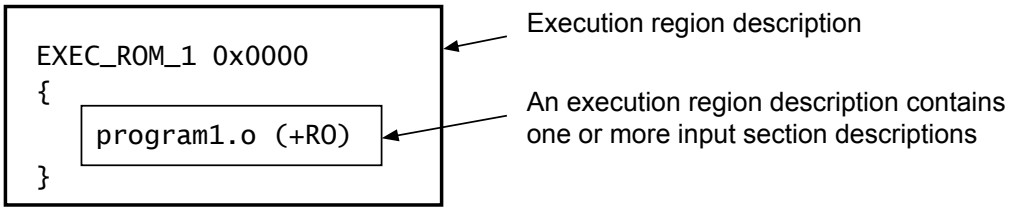

Figure 8-3 Components of an execution region description

Related concepts

8.3.4 Inheritance rules for load region address attributes on page 8-170

8.3.5 Inheritance rules for the RELOC address attribute on page 8-171

8.4.4 Inheritance rules for execution region address attributes on page 8-178

8.6 Expression evaluation in scatter files on page 8-186

8.5 Input section descriptions on page 8-181

Related tasks

7.9 Aligning regions to page boundaries on page 7-147

Related reference

8.4.2 Syntax of an execution region description on page 8-173

8.4.3 Execution region attributes on page 8-175

Chapter 7 Scatter-loading Features on page 7-104

8.3.3 Load region attributes on page 8-169

8.4.2 Syntax of an execution region description

An execution region specifies where the input sections are to be placed in target memory at run-time.

The syntax of an execution region description, in *Backus-Naur Form* (BNF), is:

```
execution_region_description ::=
  exec_region_name (base_address | "+" offset) [attribute_list] [max_size | length]
```

```
"{"
    input_section_description*
"}"
```

where:

exec_region_name

Names the execution region. You can use a quoted name. The name is case-sensitive only if you use any region-related linker-defined symbols.

base_address

Specifies the address where objects in the region are to be linked. ***base_address*** must be word-aligned.

Note

Using `ALIGN` on an execution region causes both the load address and execution address to be aligned.

+offset

Describes a base address that is ***offset*** bytes beyond the end of the preceding execution region. The value of ***offset*** must be zero modulo four.

If this is the first execution region in the load region then ***+offset*** means that the base address begins ***offset*** bytes after the base of the containing load region.

If you use ***+offset***, then the execution region might inherit certain attributes from the parent load region, or from a previous execution region within the same load region.

attribute_list

The attributes that specify the properties of the execution region contents.

max_size

For an execution region marked `EMPTY` or `FILL` the ***max_size*** value is interpreted as the length of the region. Otherwise the ***max_size*** value is interpreted as the maximum size of the execution region.

[-]length

Can only be used with `EMPTY` to represent a stack that grows down in memory. If the length is given as a negative value, the ***base_address*** is taken to be the end address of the region.

input_section_description

Specifies the content of the input sections.

Note

The BNF definitions contain additional line returns and spaces to improve readability. They are not required in scatter-loading descriptions and are ignored if present in a scatter file.

Related concepts

8.4.5 Considerations when using a relative address +offset for execution regions on page 8-179

8.6 Expression evaluation in scatter files on page 8-186

2.5 Base Platform linking model on page 2-30

10.1 Restrictions on the use of scatter files with the Base Platform model on page 10-209

8.3.4 Inheritance rules for load region address attributes on page 8-170

8.3.5 Inheritance rules for the RELOC address attribute on page 8-171

8.5 Input section descriptions on page 8-181

Related tasks

7.9 Aligning regions to page boundaries on page 7-147

Related reference

8.4.1 Components of an execution region description on page 8-173

8.4.3 Execution region attributes on page 8-175

Chapter 7 Scatter-loading Features on page 7-104

6.3 Region-related symbols on page 6-90

8.4.3 Execution region attributes

An execution region has attributes that allow you to control where parts of your image are loaded in the target memory at runtime.

The execution region attributes are:

`ABSOLUTE`

The content is placed at a fixed address that does not change after linking. A base designator specifies the execution address of the region.

`ALIGN` ***`alignment`***

Increase the alignment constraint for the execution region from 4 to ***`alignment`***. ***`alignment`*** must be a positive power of 2. If the execution region has a ***`base_address`***, then the address must be ***`alignment`*** aligned. If the execution region has a ***`+offset`***, then the linker aligns the calculated base address of the region to an ***`alignment`*** boundary.

Note

`ALIGN` on an execution region causes both the load address and execution address to be aligned. This alignment can result in padding being added to the ELF file. To align only the execution address, use the `AlignExpr` expression on the base address.

`ALIGNALL` ***`value`***

Increases the alignment of sections within the execution region.

The value must be a positive power of 2 and must be greater than or equal to 4.

`ANY_SIZE` ***`max_size`***

Specifies the maximum size within the execution region that `armlink` can fill with unassigned sections. You can use a simple expression to specify the ***`max_size`***. That is, you cannot use functions such as `ImageLimit()`.

Note

Specifying `ANY_SIZE` overrides any effects that `--any_contingency` has on the region.

Be aware of the following restrictions when using this keyword:

- ***`max_size`*** must be less than or equal to the region size.
- You can use `ANY_SIZE` on a region without a `.ANY` selector but `armlink` ignores it.

AUTO_OVERLAY

Use to indicate regions of memory where armlink assigns the overlay sections for loading into at runtime. Overlay sections are those named .ARM.overlay*N* in the input object.

The execution region must not have any section selectors.

The addresses that you give for the execution regions are the addresses that armlink expects the overlaid code to be loaded at when running. The load region containing the execution regions is where armlink places the overlay contents.

By default, the overlay manager loads overlays by copying them into RAM from some other memory that is not suitable for direct execution. For example, very slow Flash or memory from which instruction fetches are not enabled. You can keep your unloaded overlays in peripheral storage that is not mapped into the address space of the processor. To keep such overlays in peripheral storage, you must extract the data manually from the linked image.

armlink allocates every overlay to one of the AUTO_OVERLAY execution regions, and has to be loaded into only that region to run correctly.

You must use the --overlay_veneers command-line option when linking with a scatter file containing the AUTO_OVERLAY attribute.

Note

With the AUTO_OVERLAY attribute, armlink decides how your code sections get allocated to overlay regions. With the OVERLAY attribute, you must manually arrange the allocation of the code sections.

EMPTY [-]*length*

Reserves an empty block of memory of a given size in the execution region, typically used by a heap or stack. No section can be placed in a region with the EMPTY attribute.

length represents a stack that grows down in memory. If the length is given as a negative value, the *base_address* is taken to be the end address of the region.

FILL *value*

Creates a linker generated region containing a *value*. If you specify FILL, you must give a value, for example: FILL 0xFFFFFFFF. The FILL attribute replaces the following combination: EMPTY ZEROPAD PADVALUE.

In certain situations, such as a simulation, filling a region with a value is preferable to spending a long time in a zeroing loop.

FIXED

Fixed address. The linker attempts to make the execution address equal the load address. If it succeeds, then the region is a root region. If it does not succeed, then the linker produces an error.

Note

The linker inserts padding with this attribute.

NOCOMPRESS

RW data compression is enabled by default. The NOCOMPRESS keyword enables you to specify that RW data in an execution region must not be compressed in the final image.

OVERLAY

Use for sections with overlaying address ranges. If consecutive execution regions have the same +*offset*, then they are given the same base address.

The content is placed at a fixed address that does not change after linking. The content might overlap with other regions designated as OVERLAY regions.

PADVALUE *value*

Defines the *value* to use for padding. If you specify PADVALUE, you must give a value, for example:

```
EXEC 0x10000 PADVALUE 0xFFFFFFFF EMPTY ZEROPAD 0x2000
```

This example creates a region of size 0x2000 full of 0xFFFFFFFF.

PADVALUE must be a word in size. PADVALUE attributes on load regions are ignored.

PI

This region contains only position independent sections. The content does not depend on any fixed address and might be moved after linking without any extra processing.

Note

PI is not supported for AArch64 state.

Note

This attribute is not supported if an image contains execute-only sections.

SORTTYPE *algorithm*

Specifies the sorting *algorithm* for the execution region, for example:

```
ER1 +0 SORTTYPE CallTree
```

Note

This attribute overrides any sorting algorithm that you specify with the --sort command-line option.

UNINIT

Use to create execution regions containing uninitialized data or memory-mapped I/O. Only ZI output sections are affected. For example, in the following ER_RW region only the ZI part is uninitialized:

```
LR 0x8000
{
    ER_RO +0
    {
        *(+RO)
    }
    ER_RW 0x10000 UNINIT
    {
        *(+RW,+ZI)
    }
}
```

Note

Arm Compiler does not support systems with ECC or parity protection where the memory is not initialized.

ZEROPAD

Zero-initialized sections are written in the ELF file as a block of zeros and, therefore, do not have to be zero-filled at runtime.

This attribute sets the load length of a ZI output section to `Image$$`*`region_name`*`$$ZI$$Length`.

Only root execution regions can be zero-initialized using the `ZEROPAD` attribute. Using the `ZEROPAD` attribute with a non-root execution region generates a warning and the attribute is ignored.

In certain situations, such as a simulation, filling a region with a value is preferable to spending a long time in a zeroing loop.

Related concepts

7.4.8 Behavior when .ANY sections overflow because of linker-generated content on page 7-136

3.3.3 Section alignment with the linker on page 3-50

7.12 Example of using expression evaluation in a scatter file to avoid padding on page 7-151

8.6.9 Example of aligning a base address in execution space but still tightly packed in load space on page 8-192

8.4.5 Considerations when using a relative address +offset for execution regions on page 8-179

8.6 Expression evaluation in scatter files on page 8-186

4.3 Optimization with RW data compression on page 4-68

8.4.4 Inheritance rules for execution region address attributes on page 8-178

Related tasks

7.9 Aligning regions to page boundaries on page 7-147

7.10 Aligning execution regions and input sections on page 7-148

Related reference

8.4.2 Syntax of an execution region description on page 8-173

6.3.3 Load$$ execution region symbols on page 6-91

8.6.6 AlignExpr(expr, align) function on page 8-190

8.1 BNF notation used in scatter-loading description syntax on page 8-165

11.1 --any_contingency on page 11-218

6.3.2 Image$$ execution region symbols on page 6-90

8.5.2 Syntax of an input section description on page 8-181

11.94 --overlay_veneers on page 11-323

11.126 --sort=algorithm on page 11-357

Related information

Overlay support in Arm Compiler

8.4.4 Inheritance rules for execution region address attributes

An execution region can inherit the attributes of a previous execution region.

For an execution region to inherit the attributes of a previous execution region, specify a `+`*`offset`* base address for that region. The first `+`*`offset`* execution region can inherit the attributes of the parent load region. An execution region cannot inherit attributes if:

- You explicitly set the attribute of that execution region.
- The previous execution region has the `AUTO_OVERLAY` or `OVERLAY` attribute.

You can explicitly set an execution region with the `ABSOLUTE`, `AUTO_OVERLAY`, `PI`, or `OVERLAY` attributes. However, an execution region can only inherit the `RELOC` attribute from the parent load region.

Note

PI and RELOC are not supported for AArch64 state.

The following inheritance rules apply when no address attribute is specified:

- The OVERLAY attribute cannot be inherited. A region with the OVERLAY attribute cannot inherit.
- A base address load or execution region always defaults to ABSOLUTE.
- A *+offset* execution region inherits the address attribute from the previous execution region or parent load region if no previous execution region exists.

Example

This example shows the inheritance rules for setting the address attributes of execution regions:

```
LR1 0x8000 PI
{
    ER1 +0         ; ER1 inherits PI from LR1
    {
        …
    }
    ER2 +0         ; ER2 inherits PI from ER1
    {
        …
    }
    ER3 0x10000    ; ER3 does not inherit because it has no relative base
                     address and gets the default of ABSOLUTE
    {
        …
    }
    ER4 +0         ; ER4 inherits ABSOLUTE from ER3
    {
        …
    }
    ER5 +0 PI      ; ER5 does not inherit, it explicitly sets PI
    {
        …
    }
    ER6 +0 OVERLAY ; ER6 does not inherit, an OVERLAY cannot inherit
    {
        …
    }
    ER7 +0        ; ER7 cannot inherit OVERLAY, gets the default of ABSOLUTE
    {
        …
    }
}
```

Related concepts

8.3.6 Considerations when using a relative address +offset for a load region on page 8-171

8.3.4 Inheritance rules for load region address attributes on page 8-170

8.4.5 Considerations when using a relative address +offset for execution regions on page 8-179

Related reference

8.3.1 Components of a load region description on page 8-167

8.4.1 Components of an execution region description on page 8-173

8.4.2 Syntax of an execution region description on page 8-173

8.4.5 Considerations when using a relative address +offset for execution regions

There are some considerations to be aware of when using a relative address for execution regions.

When using *+offset* to specify an execution region base address:

- The first execution region inherits the attributes of the parent load region, unless an attribute is explicitly set on that execution region.
- A *+offset* execution region ER2 inherits the attributes of the execution region ER1 immediately before it, unless:
  - ER1 has the OVERLAY attribute.
  - ER2 has an explicit attribute set.

If an execution region is unable to inherit an attribute, then it gets the attribute `ABSOLUTE`.

- If the parent load region has the `RELOC` attribute, then all execution regions within that load region must have a `+offset` base address.

Related concepts

8.4.4 Inheritance rules for execution region address attributes on page 8-178

8.3.5 Inheritance rules for the RELOC address attribute on page 8-171

Related reference

8.2 Syntax of a scatter file on page 8-166

8.5 Input section descriptions

An input section description is a pattern that identifies input sections.

This section contains the following subsections:

8.5.1 Components of an input section description

The components of an input section description allow you to identify the parts of an ELF file that are to be placed in an execution region.

An input section description identifies input sections by:

- Module name (object filename, library member name, or library filename). The module name can use wildcard characters.
- Input section name, type, or attributes such as `READ-ONLY`, or `CODE`. You can use wildcard characters for the input section name.
- Symbol name.

The following figure shows the components of a typical input section description.

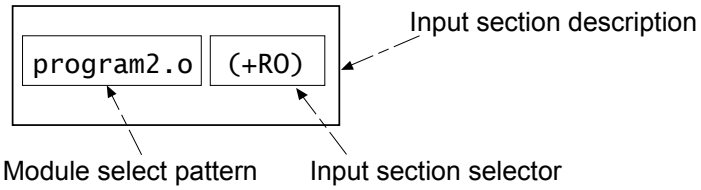

Figure 8-4 Components of an input section description

Note

Ordering in an execution region does not affect the ordering of sections in the output image.

Input section descriptions when linking partially-linked objects

You cannot specify partially-linked objects in an input section description, only the combined object file.

For example, if you link the partially linked objects `obj1.o`, `obj2.o`, and `obj3.o` together to produce `obj_all.o`, the component object names are discarded in the resulting object. Therefore, you cannot refer to one of the objects by name, for example, `obj1.o`. You can refer only to the combined object `obj_all.o`.

Related reference

8.5.2 Syntax of an input section description on page 8-181

8.2 Syntax of a scatter file on page 8-166

11.100 --partial on page 11-329

8.5.2 Syntax of an input section description

An input section description specifies what input sections are loaded into the parent execution region.

The syntax of an input section description, in *Backus-Naur Form* (BNF), is:

```
input_section_description ::=
              module_select_pattern [ "(" input_section_selector ( ","
input_section_selector )* ")" ]

input_section_selector ::=
              "+" input_section_attr |
```

```
                input_section_pattern |
                input_section_type |
                input_symbol_pattern |
                section_properties
```

Where:

module_select_pattern

A pattern that is constructed from literal text. An input section matches a module selector pattern when *module_select_pattern* matches one of the following:

- The name of the object file containing the section.
- The name of the library member (without leading path name).
- The full name of the library (including path name) the section is extracted from. If the names contain spaces, use wild characters to simplify searching. For example, use `*libname.lib` to match `C:\lib dir\libname.lib`.

The wildcard character `*` matches zero or more characters and `?` matches any single character.

Matching is not case-sensitive, even on hosts with case-sensitive file naming.

Use `*.o` to match all objects. Use `*` to match all object files and libraries.

You can use quoted filenames, for example `"file one.o"`.

You cannot have two `*` selectors in a scatter file. You can, however, use two modified selectors, for example `*A` and `*B`, and you can use a `.ANY` selector together with a `*` module selector. The `*` module selector has higher precedence than `.ANY`. If the portion of the file containing the `*` selector is removed, the `.ANY` selector then becomes active.

input_section_attr

An attribute selector that is matched against the input section attributes. Each *input_section_attr* follows a `+`.

The selectors are not case-sensitive. The following selectors are recognized:

- `RO-CODE`.
- `RO-DATA`.
- `RO`, selects both `RO-CODE` and `RO-DATA`.
- `RW-DATA`.
- `RW-CODE`.
- `RW`, selects both `RW-CODE` and `RW-DATA`.
- `XO`.
- `ZI`.
- `ENTRY`, that is, a section containing an `ENTRY` point.

The following synonyms are recognized:

- `CODE` for `RO-CODE`.
- `CONST` for `RO-DATA`.
- `TEXT` for `RO`.
- `DATA` for `RW`.
- `BSS` for `ZI`.

The following pseudo-attributes are recognized:

- `FIRST`.
- `LAST`.

Use `FIRST` and `LAST` to mark the first and last sections in an execution region if the placement order is important. For example, if a specific input section must be first in the region and an input section containing a checksum must be last.

Caution

`FIRST` and `LAST` must not violate the basic attribute sorting order. For example, `FIRST RW` is placed after any read-only code or read-only data.

There can be only one `FIRST` or one `LAST` attribute for an execution region, and it must follow a single *`input_section_selector`*. For example:

`*(section, +FIRST)`
This pattern is correct.

`*(+FIRST, section)`
This pattern is incorrect and produces an error message.

`input_section_pattern`
A pattern that is matched, without case sensitivity, against the input section name. It is constructed from literal text. The wildcard character `*` matches 0 or more characters, and `?` matches any single character. You can use a quoted input section name.

Note

If you use more than one *`input_section_pattern`*, ensure that there are no duplicate patterns in different execution regions to avoid ambiguity errors.

`input_section_type`
A number that is compared against the input section type. The number can be decimal or hexadecimal.

`input_symbol_pattern`

You can select the input section by the global symbol name that the section defines. The global name enables you to choose individual sections with the same name from partially linked objects.

The `:gdef:` prefix distinguishes a global symbol pattern from a section pattern. For example, use `:gdef:mysym` to select the section that defines `mysym`. The following example shows a scatter file in which `ExecReg1` contains the section that defines global symbol `mysym1`, and the section that contains global symbol `mysym2`:

```
LoadRegion 0x8000
{
    ExecReg1 +0
    {
        *(:gdef:mysym1)
        *(:gdef:mysym2)
    }
    ; rest of scatter-loading description
}
```

You can use a quoted global symbol pattern. The `:gdef:` prefix can be inside or outside the quotes.

Note

If you use more than one *`input_symbol_pattern`*, ensure that there are no duplicate patterns in different execution regions to avoid ambiguity errors.

section_properties

A section property can be `+FIRST`, `+LAST`, and `OVERALIGN` *`value`*.

The value for `OVERALIGN` must be a positive power of 2 and must be greater than or equal to 4.

Note

- The order of input section descriptors is not significant.
- Only input sections that match both *`module_select_pattern`* and at least one *`input_section_attr`* or *`input_section_pattern`* are included in the execution region.

  If you omit `(+` *`input_section_attr`*`)` and `(`*`input_section_pattern`*`)`, the default is `+RO`.
- Do not rely on input section names that the compiler generates, or that are used by Arm library code. If, for example, different compiler options are used, the input section names can change between compilations. In addition, section naming conventions that are used by the compiler are not guaranteed to remain constant between releases.
- The BNF definitions contain extra line returns and spaces to improve readability. If present in a scatter file, they are not required in scatter-loading descriptions and are ignored.

Related concepts

7.4.8 Behavior when .ANY sections overflow because of linker-generated content on page 7-136

8.5.3 Examples of module and input section specifications on page 8-184

7.4.5 Examples of using placement algorithms for .ANY sections on page 7-131

7.4.6 Example of next_fit algorithm showing behavior of full regions, selectors, and priority on page 7-133

7.4.7 Examples of using sorting algorithms for .ANY sections on page 7-134

Related tasks

7.10 Aligning execution regions and input sections on page 7-148

Related reference

8.5.1 Components of an input section description on page 8-181

8.1 BNF notation used in scatter-loading description syntax on page 8-165

8.2 Syntax of a scatter file on page 8-166

7.4 Placement of unassigned sections on page 7-128

8.5.3 Examples of module and input section specifications

Examples of *`module_select_pattern`* specifications and *`input_section_selector`* specifications.

Examples of *`module_select_pattern`* specifications are:

- `*` matches any module or library.
- `*.o` matches any object module.
- `math.o` matches the `math.o` module.
- `*armlib*` matches all C libraries supplied by Arm.
- `"file 1.o"` matches the file `file 1.o`.
- `*math.lib` matches any library path ending with `math.lib`, for example, `C:\apps\lib\math\satmath.lib`.

Examples of *`input_section_selector`* specifications are:

- `+RO` is an input section attribute that matches all RO code and all RO data.
- `+RW,+ZI` is an input section attribute that matches all RW code, all RW data, and all ZI data.
- `BLOCK_42` is an input section pattern that matches sections named `BLOCK_42`. There can be multiple ELF sections with the same `BLOCK_42` name that possess different attributes, for example `+RO-CODE,+RW`.

Related reference

8.5.1 Components of an input section description on page 8-181

8.5.2 Syntax of an input section description on page 8-181

8.6 Expression evaluation in scatter files

Scatter files frequently contain numeric constants. These can be specific values, or the result of an expression.

This section contains the following subsections:

8.6.1 Expression usage in scatter files

You can use expressions for various load and execution region attributes.

Expressions can be used in the following places:

- Load and execution region *`base_address`*.
- Load and execution region *`+offset`*.
- Load and execution region *`max_size`*.
- Parameter for the `ALIGN`, `FILL` or `PADVALUE` keywords.
- Parameter for the `ScatterAssert` function.

Example of specifying the maximum size in terms of an expression

```
LR1 0x8000 (2 * 1024)
{
    ER1 +0 (1 * 1024)
    {
        *(+RO)
    }
    ER2 +0 (1 * 1024)
    {
        *(+RW,+ZI)
    }
}
```

Related concepts

8.6.2 Expression rules in scatter files on page 8-187

8.6.3 Execution address built-in functions for use in scatter files on page 8-187

8.6.4 ScatterAssert function and load address related functions on page 8-189

8.6.5 Symbol related function in a scatter file on page 8-190

8.3.6 Considerations when using a relative address +offset for a load region on page 8-171

8.4.5 Considerations when using a relative address +offset for execution regions on page 8-179

8.6.9 Example of aligning a base address in execution space but still tightly packed in load space on page 8-192

Related reference

8.2 Syntax of a scatter file on page 8-166

8.3.2 Syntax of a load region description on page 8-168

8.4.2 Syntax of an execution region description on page 8-173

8.6.2 Expression rules in scatter files

Expressions follow the C-Precedence rules.

Expressions are made up of the following:

- Decimal or hexadecimal numbers.
- Arithmetic operators: `+`, `-`, `/`, `*`, `~`, `OR`, and `AND`

  The `OR` and `AND` operators map to the C operators | and & respectively.
- Logical operators: `LOR`, `LAND`, and `!`

  The `LOR` and `LAND` operators map to the C operators `||` and && respectively.
- Relational operators: `<`, `<=`, `>`, `>=`, and `==`

  Zero is returned when the expression evaluates to false and nonzero is returned when true.
- Conditional operator: *`Expression ? Expression1 : Expression2`*

  This matches the C conditional operator. If *`Expression`* evaluates to nonzero then *`Expression1`* is evaluated otherwise *`Expression2`* is evaluated.

  Note

  When using a conditional operator in a *`+offset`* context on an execution region or load region description, the final expression is considered relative only if both *`Expression1`* and *`Expression2`*, are considered relative. For example:

```
er1 0x8000
{
    …
}
er2 ((ImageLimit(er1) < 0x9000) ? +0 : +0x1000)    ; er2 has a relative address
{
    …
}
er3 ((ImageLimit(er2) < 0x10000) ? 0x0 : +0)       ; er3 has an absolute address
{
    …
}
```

- Functions that return numbers.

All operators match their C counterparts in meaning and precedence.

Expressions are not case-sensitive and you can use parentheses for clarity.

Related concepts

8.6.1 Expression usage in scatter files on page 8-186

8.6.3 Execution address built-in functions for use in scatter files on page 8-187

8.6.4 ScatterAssert function and load address related functions on page 8-189

8.6.5 Symbol related function in a scatter file on page 8-190

8.3.6 Considerations when using a relative address +offset for a load region on page 8-171

8.4.5 Considerations when using a relative address +offset for execution regions on page 8-179

8.6.9 Example of aligning a base address in execution space but still tightly packed in load space on page 8-192

Related reference

8.2 Syntax of a scatter file on page 8-166

8.3.2 Syntax of a load region description on page 8-168

8.4.2 Syntax of an execution region description on page 8-173

8.6.3 Execution address built-in functions for use in scatter files

Built-in functions are provided for use in scatter files to calculate execution addresses.

The execution address related functions can only be used when specifying a *base_address*, *+offset* value, or *max_size*. They map to combinations of the linker defined symbols shown in the following table.

Table 8-2 Execution address related functions

Function	Linker defined symbol value
ImageBase(*region_name*)	Image$$*region_name*$$Base
ImageLength(*region_name*)	Image$$*region_name*$$Length + Image$$*region_name*$$ZI$$Length
ImageLimit(*region_name*)	Image$$*region_name*$$Base + Image$$*region_name*$$Length + Image$$*region_name*$$ZI$$Length

The parameter *region_name* can be either a load or an execution region name. Forward references are not permitted. The *region_name* can only refer to load or execution regions that have already been defined.

Note

You cannot use these functions when using the `.ANY` selector pattern. This is because a `.ANY` region uses the maximum size when assigning sections. The maximum size might not be available at that point, because the size of all regions is not known until after the `.ANY` assignment.

The following example shows how to use `ImageLimit(region_name)` to place one execution region immediately after another:

```
LR1 0x8000
{
    ER1 0x100000
    {
        *(+RO)
    }
}
LR2 0x100000
{
    ER2 (ImageLimit(ER1))               ; Place ER2 after ER1 has finished
    {
        *(+RW +ZI)
    }
}
```

Using +*offset* with expressions

A *+offset* value for an execution region is defined in terms of the previous region. You can use this as an input to other expressions such as `AlignExpr`. For example:

```
LR1 0x4000
{
    ER1 AlignExpr(+0, 0x8000)
    {
        …
    }
}
```

By using `AlignExpr`, the result of `+0` is aligned to a `0x8000` boundary. This creates an execution region with a load address of `0x4000` but an execution address of `0x8000`.

Related concepts

8.6.1 Expression usage in scatter files on page 8-186
8.6.2 Expression rules in scatter files on page 8-187
8.6.4 ScatterAssert function and load address related functions on page 8-189
8.6.5 Symbol related function in a scatter file on page 8-190
8.3.6 Considerations when using a relative address +offset for a load region on page 8-171

8.6.10 Scatter files containing relative base address load regions and a ZI execution region on page 8-192
8.4.5 Considerations when using a relative address +offset for execution regions on page 8-179
8.6.9 Example of aligning a base address in execution space but still tightly packed in load space on page 8-192

Related reference

8.2 Syntax of a scatter file on page 8-166
8.3.2 Syntax of a load region description on page 8-168
8.4.2 Syntax of an execution region description on page 8-173
8.6.6 AlignExpr(expr, align) function on page 8-190
6.3.2 Image$$ execution region symbols on page 6-90

8.6.4 ScatterAssert function and load address related functions

The `ScatterAssert` function allows you to perform more complex size checks than those permitted by the *`max_size`* attribute.

The `ScatterAssert(`*`expression`*`)` function can be used at the top level, or within a load region. It is evaluated after the link has completed and gives an error message if *`expression`* evaluates to false.

The load address related functions can only be used within the `ScatterAssert` function. They map to the three linker defined symbol values:

Table 8-3 Load address related functions

Function	Linker defined symbol value
`LoadBase(`*`region_name`*`)`	`Load$$`*`region_name`*`$$Base`
`LoadLength(`*`region_name`*`)`	`Load$$`*`region_name`*`$$Length`
`LoadLimit(`*`region_name`*`)`	`Load$$`*`region_name`*`$$Limit`

The parameter *`region_name`* can be either a load or an execution region name. Forward references are not permitted. The *`region_name`* can only refer to load or execution regions that have already been defined.

The following example shows how to use the `ScatterAssert` function to write more complex size checks than those permitted by the *`max_size`* attribute of the region:

```
LR1 0x8000
{
    ER0 +0
    {
        *(+RO)
    }
    ER1 +0
    {
        file1.o(+RW)
    }
    ER2 +0
    {
        file2.o(+RW)
    }
    ScatterAssert((LoadLength(ER1) + LoadLength(ER2)) < 0x1000)
                                        ; LoadLength is compressed size
    ScatterAssert((ImageLength(ER1) + ImageLength(ER2)) < 0x2000)
                                        ; ImageLength is uncompressed size
}
ScatterAssert(ImageLength(LR1) < 0x3000)
                                        ; Check uncompressed size of load region LR1
```

Related concepts

8.6.1 Expression usage in scatter files on page 8-186

8.6.2 Expression rules in scatter files on page 8-187
8.6.3 Execution address built-in functions for use in scatter files on page 8-187
8.6.5 Symbol related function in a scatter file on page 8-190
8.6.9 Example of aligning a base address in execution space but still tightly packed in load space on page 8-192

Related reference

8.2 Syntax of a scatter file on page 8-166
8.3.2 Syntax of a load region description on page 8-168
8.4.2 Syntax of an execution region description on page 8-173
6.3.3 Load$$ execution region symbols on page 6-91

8.6.5 Symbol related function in a scatter file

The symbol related function `defined` allows you to assign different values depending on whether or not a global symbol is defined.

The symbol related function, `defined(`*`global_symbol_name`*`)` returns zero if *`global_symbol_name`* is not defined and nonzero if it is defined.

Example

The following scatter file shows an example of conditionalizing a base address based on the presence of the symbol `version1`:

```
LR1 0x8000
{
    ER1 (defined(version1) ? 0x8000 : 0x10000)   ; Base address is 0x8000
                                                 ; if version1 is defined
                                                 ; 0x10000 if not
    {
        *(+RO)
    }
    ER2 +0
    {
        *(+RW +ZI)
    }
}
```

Related concepts

8.6.1 Expression usage in scatter files on page 8-186
8.6.2 Expression rules in scatter files on page 8-187
8.6.3 Execution address built-in functions for use in scatter files on page 8-187
8.6.4 ScatterAssert function and load address related functions on page 8-189
8.6.9 Example of aligning a base address in execution space but still tightly packed in load space on page 8-192

Related reference

8.2 Syntax of a scatter file on page 8-166
8.3.2 Syntax of a load region description on page 8-168
8.4.2 Syntax of an execution region description on page 8-173

8.6.6 AlignExpr(expr, align) function

Aligns an address expression to a specified boundary.

This function returns:

```
(expr + (align-1)) & ~(align-1))
```

Where:

- `expr` is a valid address expression.
- `align` is the alignment, and must be a positive power of 2.

It increases `expr` until:

```
expr ≡ 0 (mod align)
```

Example

This example aligns the address of `ER2` on an 8-byte boundary:

```
ER +0
{
    …
}
ER2 AlignExpr(+0x8000,8)
{
    …
}
```

Relationship with the ALIGN keyword

The following relationship exists between `ALIGN` and `AlignExpr`:

`ALIGN` keyword

Load and execution regions already have an `ALIGN` keyword:

- For load regions the `ALIGN` keyword aligns the base of the load region in load space and in the file to the specified alignment.
- For execution regions the `ALIGN` keyword aligns the base of the execution region in execution and load space to the specified alignment.

`AlignExpr`

Aligns the expression it operates on, but has no effect on the properties of the load or execution region.

Related reference

8.4.3 Execution region attributes on page 8-175

8.6.7 GetPageSize() function

Returns the page size when an image is demand paged, and is useful when used with the `AlignExpr` function.

When you link with the `--paged` command-line option, returns the value of the internal page size that `armlink` uses in its alignment calculations. Otherwise, it returns zero.

By default the internal page size is set to `8000`, but you can change it with the `--pagesize` command-line option.

Example

This example aligns the base address of `ER` to a Page Boundary:

```
ER AlignExpr(+0, GetPageSize())
{
    …
}
```

Related concepts

8.6.9 Example of aligning a base address in execution space but still tightly packed in load space on page 8-192

Related reference

11.99 --pagesize=pagesize on page 11-328

8.6.6 AlignExpr(expr, align) function on page 8-190

8.6.8 SizeOfHeaders() function

Returns the size of ELF header plus the estimated size of the Program Header table.

This is useful when writing demand paged images to start code and data immediately after the ELF header and Program Header table.

Example

This example sets the base of `LR1` to start immediately after the ELF header and Program Headers:

```
LR1 SizeOfHeaders()
{
    …
}
```

Related concepts

8.6.9 Example of aligning a base address in execution space but still tightly packed in load space on page 8-192

3.4 Linker support for creating demand-paged files on page 3-51

Related tasks

7.9 Aligning regions to page boundaries on page 7-147

8.6.9 Example of aligning a base address in execution space but still tightly packed in load space

This example shows how to use a combination of preprocessor macros and expressions to copy tightly packed execution regions to execution addresses in a page-boundary.

Using the `ALIGN` scatter-loading keyword aligns the load addresses of `ER2` and `ER3` as well as the execution addresses

Aligning a base address in execution space but still tightly packed in load space

```
#! armclang -E#define START_ADDRESS  0x100000
#define PAGE_ALIGNMENT 0x100000

LR1 0x8000
{
    ER0 +0
    {
        *(InRoot$$Sections)
    }
    ER1 START_ADDRESS
    {
        file1.o(*)
    }
    ER2 AlignExpr(ImageLimit(ER1), PAGE_ALIGNMENT)
    {
        file2.o(*)
    }
    ER3 AlignExpr(ImageLimit(ER2), PAGE_ALIGNMENT)
    {
        file3.o(*)
    }
}
```

Related reference

8.3.3 Load region attributes on page 8-169

8.4.3 Execution region attributes on page 8-175

8.6.7 GetPageSize() function on page 8-191

8.6.8 SizeOfHeaders() function on page 8-191

8.3.2 Syntax of a load region description on page 8-168

8.4.2 Syntax of an execution region description on page 8-173

8.6.6 AlignExpr(expr, align) function on page 8-190

8.6.10 Scatter files containing relative base address load regions and a ZI execution region

You might want to place *zero-initialized* (ZI) data in one load region, and use a relative base address for the next load region.

To place ZI data in load region LR1, and use a relative base address for the next load region LR2, for example:

```
LR1 0x8000
{
    er_progbits +0
    {
        *(+RO,+RW) ; Takes space in the Load Region
    }
    er_zi +0
    {
        *(+ZI) ; Takes no space in the Load Region
    }
}
LR2 +0 ; Load Region follows immediately from LR1
{
    er_moreprogbits +0
    {
        file1.o(+RO) ; Takes space in the Load Region
    }
}
```

Because the linker does not adjust the base address of LR2 to account for ZI data, the execution region `er_zi` overlaps the execution region `er_moreprogbits`. This generates an error when linking.

To correct this, use the `ImageLimit()` function with the name of the ZI execution region to calculate the base address of LR2. For example:

```
LR1 0x8000
{
    er_progbits +0
    {
        *(+RO,+RW) ; Takes space in the Load Region
    }
    er_zi +0
    {
        *(+ZI) ; Takes no space in the Load Region
    }
}
LR2 ImageLimit(er_zi) ; Set the address of LR2 to limit of er_zi
{
    er_moreprogbits +0
    {
        file1.o(+RO) ; Takes space in the Load Region
    }
}
```

Related concepts

8.6 Expression evaluation in scatter files on page 8-186

8.6.1 Expression usage in scatter files on page 8-186

8.6.2 Expression rules in scatter files on page 8-187

8.6.3 Execution address built-in functions for use in scatter files on page 8-187

Related reference

8.2 Syntax of a scatter file on page 8-166

8.3.2 Syntax of a load region description on page 8-168

8.4.2 Syntax of an execution region description on page 8-173

6.3.2 Image$$ execution region symbols on page 6-90

Chapter 9
BPABI Shared Libraries and Executables

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

It contains the following sections:

9.1 About the Base Platform Application Binary Interface (BPABI)

The *Base Platform Application Binary Interface* (BPABI) is a meta-standard for third parties to generate their own platform-specific image formats.

Many embedded systems use an *operating system* (OS) to manage the resources on a device. In many cases this is a large, single executable with a *Real-Time Operating System* (RTOS) that tightly integrates with the applications.

To run an application or use a shared library on a platform OS, you must conform to the *Application Binary Interface* (ABI) for the platform and also the ABI for the Arm architecture. This can involve substantial changes to the linker output, for example, a custom file format. To support such a wide variety of platforms, the ABI for the Arm architecture provides the BPABI.

The BPABI provides a base standard from which a platform ABI can be derived. The linker produces a BPABI conforming ELF image or shared library. A platform specific tool called a post-linker translates this ELF output file into a platform-specific file format. Post linker tools are provided by the platform OS vendor. The following figure shows the BPABI tool flow.

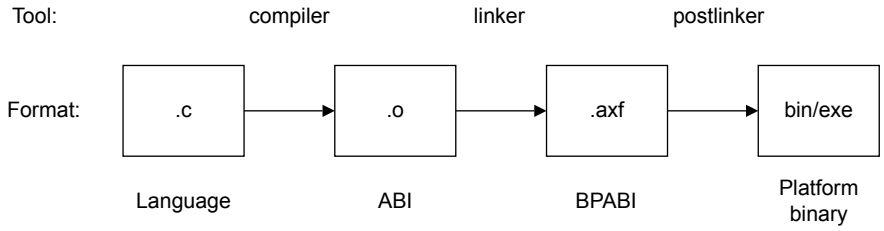

Figure 9-1 BPABI tool flow

Related concepts

9.2 Platforms supported by the BPABI on page 9-196

Related information

Base Platform ABI for the Arm Architecture

AN242 Dynamic Linking with the Arm Compiler toolchain

9.2 Platforms supported by the BPABI

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

The platform models are:

Bare metal

The bare metal model is designed for an offline dynamic loader or a simple module loader. References between modules are resolved by the loader directly without any additional support structures.

DLL-like

The *dynamically linked library* (DLL) like model sacrifices transparency between the dynamic and static library in return for better load and run-time efficiency.

——— **Note** ———

The DLL-like model is not supported for AArch64 state.

———————————

Linker support for the BPABI

The Arm linker supports all three BPABI models enabling you to link a collection of objects and libraries into a:

- Bare metal executable image.
- BPABI DLL shared object.
- BPABI executable file.

Related concepts

9.1 About the Base Platform Application Binary Interface (BPABI) on page 9-195

Related reference

11.32 --dll on page 11-254

9.3 Features common to all BPABI models

Some features are common to all BPABI models.

The linker enables you to build *Base Platform Application Binary Interface* (BPABI) shared libraries and to link objects against shared libraries. The following features are common to all BPABI models:

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

This section contains the following subsections:

9.3.1 About importing and exporting symbols for BPABI models

How symbols are imported and exported depends on the platform model.

In traditional linking, all symbols must be defined at link time for linking into a single executable file containing all the required code and data. In platforms that support dynamic linking, symbol binding can be delayed to load-time or in some cases, run-time. Therefore, the application can be split into a number of modules, where a module is either an executable or a shared library. Any symbols that are defined in modules other than the current module are placed in the dynamic symbol table. Any functions that are suitable for dynamically linking to at load or runtime are also listed in the dynamic symbol table.

There are two ways to control the contents of the dynamic symbol table:

- Automatic rules that infer the contents from the ELF symbol visibility property.
- Manual directives that are present in a steering file.

Related concepts

9.3.3 Automatic import and export for BPABI models on page 9-198

9.3.1 About importing and exporting symbols for BPABI models on page 9-197

9.3.2 Symbol visibility for BPABI models on page 9-197

9.3.4 Manual import and export for BPABI models on page 9-198

9.3.5 Symbol versioning for BPABI models on page 9-199

9.3.6 RW compression for BPABI models on page 9-199

9.5.3 The symbol versioning script file on page 9-206

Related reference

9.4.3 Linker command-line options for bare metal and DLL-like models on page 9-201

9.3.2 Symbol visibility for BPABI models

For *Base Platform Application Binary Interface* (BPABI) models, each symbol has a visibility property that can be controlled by compiler switches, a steering file, or attributes in the source code.

If a symbol is a reference, the visibility controls the definitions that the linker can use to define the symbol.

If a symbol is a definition, the visibility controls whether the symbol can be made visible outside the current module.

The visibility options defined by the ELF specification are:

Table 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_HIDDEN` `STV_INTERNAL`	Symbol must be resolved within the module.	Symbol is not visible outside the module.

Symbol preemption can happen in *dynamically linked library* (DLL) like implementations of the BPABI. The platform owner defines how this works. See the documentation for your specific platform for more information.

Related concepts

4.3 Optimization with RW data compression on page 4-68

9.5.3 The symbol versioning script file on page 9-206

Related reference

9.4.3 Linker command-line options for bare metal and DLL-like models on page 9-201

11.88 --max_visibility=type on page 11-317

11.95 --override_visibility on page 11-324

12.1 EXPORT steering file command on page 12-395

12.3 IMPORT steering file command on page 12-397

12.5 REQUIRE steering file command on page 12-399

11.147 --use_definition_visibility on page 11-379

Related information

EXPORT or GLOBAL

9.3.3 Automatic import and export for BPABI models

The linker can automatically import and export symbols for BPABI models.

This behavior depends on a combination of the symbol visibility in the input object file, if the output is an executable or a shared library. This depends on what type of linking model is being used.

Related concepts

9.3 Features common to all BPABI models on page 9-197

9.5 Symbol versioning on page 9-205

Related reference

9.4.3 Linker command-line options for bare metal and DLL-like models on page 9-201

9.3.4 Manual import and export for BPABI models

You can directly control the import and export of symbols with a linker steering file.

You can use linker steering files to:

- Manually control dynamic import and export.
- Override the automatic rules.

The steering file commands available to control the dynamic symbol table contents are:

- `EXPORT`.
- `IMPORT`.
- `REQUIRE`.

Related concepts

6.6 Edit the symbol tables with a steering file on page 6-100

Related reference

12.1 EXPORT steering file command on page 12-395

12.3 IMPORT steering file command on page 12-397

12.5 REQUIRE steering file command on page 12-399

9.3.5 Symbol versioning for BPABI models

Symbol versioning provides a way to tightly control the interface of a shared library.

When a symbol is imported from a shared library that has versioned symbols, `armlink` binds to the most recent (default) version of the symbol. At load or run-time when the platform OS resolves the symbol version, it always resolves to the version selected by `armlink`, even if there is a more recent version available. This process is automatic.

When a symbol is exported from an executable or a shared library, it can be given a version. `armlink` supports explicit symbol versioning where you use a script to precisely define the versions.

Related concepts

9.5 Symbol versioning on page 9-205

9.3.6 RW compression for BPABI models

The decompressor for compressed RW data is tightly integrated into the start-up code in the Arm C library.

When running an application on a platform OS, this functionality must be provided by the platform or platform libraries. Therefore, RW compression is turned off when linking a *Base Platform Application Binary Interface* (BPABI) file because there is no decompressor. It is not possible to turn compression back on again.

Related concepts

4.3 Optimization with RW data compression on page 4-68

9.4 Bare metal and DLL-like memory models

If you are developing applications or DLLs for a specific platform OS that are based around the BPABI, there are some features that you must be aware of.

You must use the following information in conjunction with the platform documentation:

- BPABI standard memory model.
- Mandatory symbol versioning in the BPABI DLL-like model.
- Automatic dynamic symbol table rules in the BPABI DLL-like model.
- Addressing modes in the BPABI DLL-like model.
- C++ initialization in the BPABI DLL-like model.

If you are implementing a platform OS, you must use this information in conjunction with the BPABI specification.

Note

The DLL-like model is not supported for AArch64 state.

This section contains the following subsections:

9.4.1 BPABI standard memory model

Base Platform Application Binary Interface (BPABI) files have a standard memory model that is described in the BPABI specification.

When you use the `--bpabi` command-line option, the linker automatically applies the standard memory model and ignores any scatter file that you specify on the command-line. This is equivalent to the following image layout:

```
LR_1 <read-only base address>
{
    ER_RO  +0
    {
        *(+RO)
    }
}
LR_2 <read-write base address>
{
    ER_RW  +0
    {
        *(+RW)
    }
    ER_ZI  +0
    {
        *(+ZI)
    }
}
```

The BPABI model is also referred to as the bare metal and DLL-like memory model.

Note

The DLL-like model is not supported for AArch64 state.

Related concepts

9.4.2 Customization of the BPABI standard memory model on page 9-201

9.4.2 Customization of the BPABI standard memory model

You can customize the BPABI standard memory model with the memory map related command-line options.

Note

If you specify the option `--ropi`, `LR_1` is marked as position-independent. Likewise, if you specify the option `--rwpi`, `LR_2` is marked as position-independent.

Note

In most cases, you must specify the `--ro_base` and `--rw_base` switches, because the default values, `0x8000` and `0` respectively, might not be suitable for your platform. These addresses do not have to reflect the addresses to which the image is relocated at run time.

If you require a more complicated memory layout, use the Base Platform linking model, `--base_platform`.

Related concepts

2.5 Base Platform linking model on page 2-30

Related reference

11.11 --bpabi on page 11-230

11.7 --base_platform on page 11-225

11.114 --ro_base=address on page 11-344

11.115 --ropi on page 11-345

11.116 --rosplit on page 11-346

11.117 --rw_base=address on page 11-347

11.118 --rwpi on page 11-348

11.157 --xo_base=address on page 11-389

9.4.3 Linker command-line options for bare metal and DLL-like models

There are linker command-line options available for building bare metal executables and *dynamically linked library* (DLL) like models for a platform OS.

The command-line options are:

Table 9-2 Turning on BPABI support

Command-line options	Description
`--base_platform`	To use scatter-loading with *Base Platform Application Binary Interface* (BPABI).
`--bpabi`	To produce a BPABI executable.
`--bpabi --dll`	To produce a BPABI DLL.

Note

The DLL-like model is not supported for AArch64 state.

Additional linker command-line options for the BPABI DLL-like model

There are additional linker command-line options available for the BPABI DLL-like model.

The additional command-line options are:

- `--export_all`, `--no_export_all`.
- `--pltgot=type`.
- `--pltgot_opts=mode`.
- `--ro_base=address`.
- `--ropi`.
- `--rosplit`.
- `--rw_base=address`.
- `--rwpi`.
- `--symver_script=filename`.
- `--symver_soname`.

Related concepts

9.4.1 BPABI standard memory model on page 9-200

9.4.5 Automatic dynamic symbol table rules in the BPABI DLL-like model on page 9-203

9.4.6 Addressing modes in the BPABI DLL-like model on page 9-203

9.4.4 Mandatory symbol versioning in the BPABI DLL-like model on page 9-202

Related reference

9.4.3 Linker command-line options for bare metal and DLL-like models on page 9-201

11.7 --base_platform on page 11-225

11.11 --bpabi on page 11-230

11.32 --dll on page 11-254

11.44 --export_all, --no_export_all on page 11-266

11.104 --pltgot=type on page 11-334

11.105 --pltgot_opts=mode on page 11-335

11.115 --ropi on page 11-345

11.116 --rosplit on page 11-346

11.117 --rw_base=address on page 11-347

11.118 --rwpi on page 11-348

11.139 --symver_script=filename on page 11-371

11.140 --symver_soname on page 11-372

Chapter 11 Linker Command-line Options on page 11-214

Related information

Base Platform ABI for the Arm Architecture

9.4.4 Mandatory symbol versioning in the BPABI DLL-like model

The *Base Platform Application Binary Interface* (BPABI) DLL-like model requires static binding to ensure a symbol can be searched for at run-time.

This is because a post-linker might translate the symbolic information in a BPABI DLL to an import or export table that is indexed by an ordinal. In which case, it is not possible to search for a symbol at run-time.

Static binding is enforced in the BPABI with the use of symbol versioning. The command-line option `--symver_soname` is on by default for BPABI files, this means that all exported symbols are given a version based on the name of the DLL.

Note

The DLL-like model is not supported for AArch64 state.

Related concepts

9.5 Symbol versioning on page 9-205

Related reference

11.139 --symver_script=filename on page 11-371

11.140 --symver_soname on page 11-372

9.4.5 Automatic dynamic symbol table rules in the BPABI DLL-like model

There are rules that apply to dynamic symbol tables for the *Base Platform Application Binary Interface* (BPABI) DLL-like model.

The following rules apply:

Executable

An undefined symbol reference is an undefined symbol error.

Global symbols with `STV_HIDDEN` or `STV_INTERNAL` visibility are never exported to the dynamic symbol table.

Global symbols with `STV_PROTECTED` or `STV_DEFAULT` visibility are not exported to the dynamic symbol table unless `--export_all` or `--export_dynamic` is set.

DLL

An undefined symbol reference is an undefined symbol error.

Global symbols with `STV_HIDDEN` or `STV_INTERNAL` visibility are never exported to the dynamic symbol table.

Note

`STV_HIDDEN` or `STV_INTERNAL` global symbols that are required for relocation can be placed in the dynamic symbol table, however the linker changes them into local symbols to prevent them from being accessed from outside the shared library.

Global symbols with `STV_PROTECTED` or `STV_DEFAULT` visibility are always exported to the dynamic symbol table.

Note

The DLL-like model is not supported for AArch64 state.

You can manually export and import symbols using the `EXPORT` and `IMPORT` steering file commands. Use the `--edit` command-line option to specify a steering file command.

Related concepts

6.6 Edit the symbol tables with a steering file on page 6-100

Related reference

6.6.2 Steering file command summary on page 6-100

6.6.3 Steering file format on page 6-101

11.36 --edit=file_list on page 11-258

11.44 --export_all, --no_export_all on page 11-266

11.45 --export_dynamic, --no_export_dynamic on page 11-267

12.1 EXPORT steering file command on page 12-395

12.3 IMPORT steering file command on page 12-397

9.4.6 Addressing modes in the BPABI DLL-like model

The main difference between the bare metal and *Base Platform Application Binary Interface* (BPABI) DLL-like models is the addressing mode used when accessing imported and own-program code and data.

There are four options available that correspond to categories in the BPABI specification:

- None.
- Direct references.
- Indirect references.
- Relative static base address references.

You can control the selection of the required addressing mode with the following command-line options:

- `--pltgot`.
- `--pltgot_opts`.

Note

The DLL-like model is not supported for AArch64 state.

Related reference

11.104 --pltgot=type on page 11-334

11.105 --pltgot_opts=mode on page 11-335

9.4.7 C++ initialization in the BPABI DLL-like model

A *dynamically linked library* (DLL) supports the initialization of static constructors with a table that contains references to initializer functions that perform the initialization.

The table is stored in an ELF section with a special section type of `SHT_INIT_ARRAY`. For each of these initializers there is a relocation of type `R_ARM_TARGET1` to a function that performs the initialization.

The ELF *Application Binary Interface* (ABI) specification describes `R_ARM_TARGET1` as either a relative form, or an absolute form.

The Arm C libraries use the relative form. For example, if the linker detects a definition of the Arm C library `__cpp_initialize__aeabi`, it uses the relative form of `R_ARM_TARGET1` otherwise it uses the absolute form.

Note

The DLL-like model is not supported for AArch64 state.

Related concepts

9.4.1 BPABI standard memory model on page 9-200

9.4.4 Mandatory symbol versioning in the BPABI DLL-like model on page 9-202

9.4.5 Automatic dynamic symbol table rules in the BPABI DLL-like model on page 9-203

9.4.6 Addressing modes in the BPABI DLL-like model on page 9-203

Related reference

9.4.3 Linker command-line options for bare metal and DLL-like models on page 9-201

Related information

Initialization of the execution environment and execution of the application

C++ initialization, construction and destruction

9.5 Symbol versioning

Symbol versioning records extra information about symbols imported from, and exported by, a dynamic shared object.

A dynamic loader uses this extra information to ensure that all the symbols required by an image are available at load time.

This section contains the following subsections:

9.5.1 Overview of symbol versioning

Symbol versioning enables shared object creators to produce new versions of symbols for use by all new clients, while maintaining compatibility with clients linked against old versions of the shared object.

Version

Symbol versioning adds the concept of a *version* to the dynamic symbol table. A version is a name that symbols are associated with. When a dynamic loader tries to resolve a symbol reference associated with a version name, it can only match against a symbol definition with the same version name.

Note

A version might be associated with previous version names to show the revision history of the shared object.

Default version

While a shared object might have multiple versions of the same symbol, a client of the shared object can only bind against the latest version.

This is called the *default version* of the symbol.

Creation of versioned symbols

By default, the linker does not create versioned symbols for a non *Base Platform Application Binary Interface* (BPABI) shared object.

Related concepts

9.5.3 The symbol versioning script file on page 9-206

Related information

--symbolversions, --no_symbolversions fromelf option

9.5.2 Embedded symbols

You can add specially-named symbols to input objects that cause the linker to create symbol versions.

These symbols are of the form:

- `name@version` for a non-default version of a symbol.
- `name@@version` for a default version of a symbol.

You must define these symbols, at the address of the function or data, as that you want to export. The symbol name is divided into two parts, a symbol name *`name`* and a version definition *`version`*. The *`name`* is added to the dynamic symbol table and becomes part of the interface to the shared object. Version creates a version called *`ver`* if it does not already exist and associates *`name`* with the version called *`ver`*.

The following example places the symbols `foo@ver1`, `foo@@ver2`, and `bar@@ver1` into the object symbol table:

```
int old_function(void) __asm__("foo@ver1");
int new_function(void) __asm__("foo@@ver2");
int other_function(void) __asm__("bar@@ver1");
```

The linker reads these symbols and creates version definitions `ver1` and `ver2`. The symbol `foo` is associated with a non-default version of `ver1`, and with a default version of `ver2`. The symbol `bar` is associated with a default version of `ver1`.

There is no way to create associations between versions with this method.

Related information

Writing A32/T32 Assembly Language

9.5.3 The symbol versioning script file

You can embed the commands to produce symbol versions in a script file.

You specify a symbol versioning script file with the command-line option `--symver_script=`*`file`*. Using this option automatically enables symbol versioning.

The script file supports the same syntax as the GNU *ld* linker.

Using a script file enables you to associate a version with an earlier version.

You can provide a steering file in addition to the embedded symbol method. If you choose to do this then your script file must match your embedded symbols and use the *Backus-Naur Form* (BNF) notation:

```
version_definition ::=
  version_name "{" symbol_association* "}" [depend_version] ";"
symbol_association ::=
  "local:" | "global:" | symbol_name ";"
```

Where:

- *`version_name`* is a string containing the name of the version.
- *`depend_version`* is a string containing the name of a version that this *`version_name`* depends on. This version must have already been defined in the script file.
- `"local:"` indicates that all subsequent `symbol_name`s in this version definition are local to the shared object and are not versioned.
- `"global:"` indicates that all subsequent `symbol_name`s belong to this version definition.

  There is an implicit `"global:"` at the start of every version definition.
- *`symbol_name`* is the name of a global symbol in the static symbol table.

Version names have no specific meaning, but they are significant in that they make it into the output. In the output, they are a part of the version specification of the library and a part of the version requirements of a program that links against such a library. The following example shows the use of version names:

```
VERSION_1
{
    ...
};
VERSION_2
{
    ...
} VERSION_1;
```

Note

If you use a script file then the version definitions and symbols associated with them must match. The linker warns you if it detects any mismatch.

Related concepts

9.5.1 Overview of symbol versioning on page 9-205

9.5.5 Linker options for enabling implicit symbol versioning on page 9-207
9.5.4 Example of creating versioned symbols on page 9-207

Related reference

11.139 --symver_script=filename on page 11-371

9.5.4 Example of creating versioned symbols

This example shows how to create versioned symbols in code and with a script file.

The following example places the symbols `foo@ver1`, `foo@@ver2`, and `bar@@ver1` into the object symbol table:

```
int old_function(void) __asm__("foo@ver1");
int new_function(void) __asm__("foo@@ver2");
int other_function(void) __asm__("bar@@ver1");
```

The corresponding script file includes the addition of dependency information so that `ver2` depends on `ver1` is:

```
ver1
{
    global:
        foo; bar;
    local:
        *;
};
ver2
{
    global:
        foo;
} ver1;
```

Related concepts

9.5 Symbol versioning on page 9-205
9.5.5 Linker options for enabling implicit symbol versioning on page 9-207

Related reference

11.139 --symver_script=filename on page 11-371

9.5.5 Linker options for enabling implicit symbol versioning

If you have to version your symbols to force static binding, but you do not care about the version number that they are given, you can use implicit symbol versioning.

Use the command-line option `--symver_soname` to turn on implicit symbol versioning.

Where a symbol has no defined version, the linker uses the `SONAME` of the file being linked.

This option can be combined with embedded symbols or a script file. `armlink` adds the `SONAME { *; };` definition to its internal representation of a symbol versioning script.

Related concepts

9.5.3 The symbol versioning script file on page 9-206
9.5 Symbol versioning on page 9-205
9.5.2 Embedded symbols on page 9-205

Related reference

11.140 --symver_soname on page 11-372

Chapter 10
Features of the Base Platform Linking Model

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

Note

The Base Platform linking model is not supported for AArch64 state.

It contains the following sections:

10.1 Restrictions on the use of scatter files with the Base Platform model

The Base Platform model supports scatter files, with some restrictions.

Although there are no restrictions on the keywords you can use in a scatter file, there are restrictions on the types of scatter files you can use:

- A load region marked with the `RELOC` attribute must contain only execution regions with a relative base address of `+offset`. The following examples show valid and invalid scatter files using the `RELOC` attribute and `+offset` relative base address:

  Valid scatter file example using

  ```
  # This is valid. All execution regions have +offset addresses.
  LR1 0x8000 RELOC
  {
      ER_RELATIVE +0
      {
          *(+RO)
      }
  }
  ```

  Invalid scatter file example using

  ```
  # This is not valid. One execution region has an absolute base address.
  LR1 0x8000 RELOC
  {
      ER_RELATIVE +0
      {
          *(+RO)
      }
      ER_ABSOLUTE 0x1000
      {
          *(+RW)
      }
  }
  ```

- Any load region that requires a PLT section must contain at least one execution region containing code, that is not marked `OVERLAY`. This execution region holds the PLT section. An `OVERLAY` region cannot be used as the PLT must remain in memory at all times. The following examples show valid and invalid scatter files that define execution regions requiring a PLT section:

  Valid scatter file example for a load region that requires a PLT section

  ```
  # This is valid. ER_1 contains code and is not OVERLAY.
  LR_NEEDING_PLT 0x8000
  {
      ER_1 +0
      {
          *(+RO)
      }
  }
  ```

  Invalid scatter file example for a load region that requires a PLT section

  ```
  # This is not valid. All execution regions containing code are marked OVERLAY.
  LR_NEEDING_PLT 0x8000
  {
      ER_1 +0 OVERLAY
      {
          *(+RO)
      }
      ER_2 +0
      {
          *(+RW)
      }
  }
  ```

- If a load region requires a PLT section, then the PLT section must be placed within the load region. By default, if a load region requires a PLT section, the linker places the PLT section in the first execution region containing code. You can override this choice with a scatter-loading selector.

  If there is more than one load region containing code, the PLT section for a load region with name *name* is `.plt_`*name*. If there is only one load region containing code, the PLT section is called `.plt`.

The following examples show valid and invalid scatter files that place a PLT section:

Valid scatter file example for placing a PLT section

```
#This is valid. The PLT section for LR1 is placed in LR1.
LR1 0x8000
{
    ER1 +0
    {
        *(+RO)
    }
    ER2 +0
    {
        *(.plt_LR1)
    }
}
LR2 0x10000
{
    ER1 +0
    {
        *(other_code)
    }
}
```

Invalid scatter file example for placing a PLT section

```
#This is not valid. The PLT section of LR1 has been placed in LR2.
LR1 0x8000
{
    ER1 +0
    {
        *(+RO)
    }
}
LR2 0x10000
{
    ER1 +0
    {
        *(.plt_LR1)
    }
}
```

Related concepts

2.5 Base Platform linking model on page 2-30

10.3 Placement of PLT sequences with the Base Platform model on page 10-213

8.3.4 Inheritance rules for load region address attributes on page 8-170

8.3.5 Inheritance rules for the RELOC address attribute on page 8-171

8.4.4 Inheritance rules for execution region address attributes on page 8-178

Related reference

8.3.3 Load region attributes on page 8-169

8.4.3 Execution region attributes on page 8-175

10.2 Scatter files for the Base Platform linking model

Scatter files containing relocatable and non-relocatable load regions for the Base Platform linking model.

Standard BPABI scatter file with relocatable load regions

If you do not specify a scatter file when linking for the Base Platform linking model, the linker uses a default scatter file defined for the standard *Base Platform Application Binary Interface* (BPABI) memory model. This scatter file defines the following relocatable load regions:

```
LR1 0x8000 RELOC
{
    ER_RO +0
    {
        *(+RO)
    }
}
LR2 0x0 RELOC
{
    ER_RW +0
    {
        *(+RW)
    }
    ER_ZI +0
    {
        *(+ZI)
    }
}
```

This example conforms to the BPABI, because it has the same two-region format as the BPABI specification.

Scatter file with some load regions that are not relocatable

This example shows two load regions LR1 and LR2 that are not relocatable.

```
LR1 0x8000
{
    ER_RO +0
    {
        *(+RO)
    }
    ER_RW +0
    {
        *(+RW)
    }
    ER_ZI +0
    {
        *(+ZI)
    }
}
LR2 0x10000
{
    ER_KNOWN_ADDRESS +0
    {
        *(fixedsection)
    }
}
LR3 0x20000 RELOC
{
    ER_RELOCATABLE +0
    {
        *(floatingsection)
    }
}
```

The linker does not have to generate dynamic relocations between LR1 and LR2 because they have fixed addresses. However, the `RELOC` load region LR3 might be widely separated from load regions LR1 and LR2 in the address space. Therefore, dynamic relocations are required between LR1 and LR3, and LR2 and LR3.

Use the options `--pltgot=direct --pltgot_opts=crosslr` to ensure a PLT is generated for each load region.

Related concepts

2.2 Bare-metal linking model on page 2-27

2.4 Base Platform Application Binary Interface (BPABI) linking model on page 2-29

10.1 Restrictions on the use of scatter files with the Base Platform model on page 10-209

Related reference

8.3.3 Load region attributes on page 8-169

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 linking in Base Platform mode.

To turn on PLT generation when in Base Platform mode (`--base_platform`) use `--pltgot=`*`option`* that generates PLT sequences. You can use the option `--pltgot_opts=crosslr` to add entries in the PLT for calls from and to `RELOC` load-regions. PLT generation for multiple Load Regions is only supported for `--pltgot=direct`.

The `--pltgot_opts=crosslr` option is useful when you have multiple load regions that might be moved relative to each other when the image is dynamically loaded. The linker generates a PLT for each load region so that calls do not have to be extended to reach a distant PLT.

Placement of linker generated PLT sections:

- When there is only one load region there is one PLT. The linker creates a section called `.plt` with an object `anon$$obj.o`.
- When there are multiple load regions, a PLT section is created for each load region that requires one. By default, the linker places the PLT section in the first execution region containing code. You can override this by specifying the exact PLT section name in the scatter file.

  For example, a load region with name *`LR_NAME`* the PLT section is called `.plt_`*`LR_NAME`* with an object of `anon$$obj.o`. To precisely name this PLT section in a scatter file, use the selector:

  `anon$$obj.o(.plt_`*`LR_NAME`*`)`

Be aware of the following:

- The linker gives an error message if the PLT for load region *`LR_NAME`* is moved out of load region *`LR_NAME`*.
- The linker gives an error message if load region *`LR_NAME`* contains a mixture of `RELOC` and non-`RELOC` execution regions. This is because it cannot guarantee that the `RELOC` execution regions are able to reach the PLT at run-time.
- `--pltgot=indirect` and `--pltgot=sbrel` are not supported for multiple load regions.

Related concepts

2.5 Base Platform linking model on page 2-30

Related reference

11.7 --base_platform on page 11-225
11.104 --pltgot=type on page 11-334
11.105 --pltgot_opts=mode on page 11-335

Chapter 11
Linker Command-line Options

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

It contains the following sections:

11.1 --any_contingency

Permits extra space in any execution regions containing `.ANY` sections for linker-generated content such as veneers and alignment padding.

Usage

Two percent of the extra space in such execution regions is reserved for veneers.

When a region is about to overflow because of potential padding, `armlink` lowers the priority of the `.ANY` selector.

This option is off by default. That is, `armlink` does not attempt to calculate padding and strictly follows the `.ANY` priorities.

Use this option with the `--scatter` option.

Related concepts

7.4.8 Behavior when .ANY sections overflow because of linker-generated content on page 7-136

Related reference

11.59 --info=topic[,topic,...] on page 11-281

11.3 --any_sort_order=order on page 11-221

11.120 --scatter=filename on page 11-350

8.5.2 Syntax of an input section description on page 8-181

11.2 --any_placement=algorithm on page 11-219

7.4 Placement of unassigned sections on page 7-128

11.2 --any_placement=algorithm

Controls the placement of sections that are placed using the `.ANY` module selector.

Syntax

`--any_placement=`*`algorithm`*

where *`algorithm`* is one of the following:

`best_fit`
Place the section in the execution region that currently has the least free space but is also sufficient to contain the section.

`first_fit`
Place the section in the first execution region that has sufficient space. The execution regions are examined in the order they are defined in the scatter file.

`next_fit`
Place the section using the following rules:

- Place in the current execution region if there is sufficient free space.
- Place in the next execution region only if there is insufficient space in the current region.
- Never place a section in a previous execution region.

`worst_fit`
Place the section in the execution region that currently has the most free space.

Use this option with the `--scatter` option.

Usage

The placement algorithms interact with scatter files and `--any_contingency` as follows:

Interaction with normal scatter-loading rules
Scatter-loading with or without `.ANY` assigns a section to the most specific selector. All algorithms continue to assign to the most specific selector in preference to `.ANY` priority or size considerations.

Interaction with `.ANY` priority

Priority is considered after assignment to the most specific selector in all algorithms.

`worst_fit` and `best_fit` consider priority before their individual placement criteria. For example, you might have `.ANY1` and `.ANY2` selectors, with the `.ANY1` region having the most free space. When using `worst_fit` the section is assigned to `.ANY2` because it has higher priority. Only if the priorities are equal does the algorithm come into play.

`first_fit` considers the most specific selector first, then priority. It does not introduce any more placement rules.

`next_fit` also does not introduce any more placement rules. If a region is marked full during `next_fit`, that region cannot be considered again regardless of priority.

Interaction with `--any_contingency`

The priority of a `.ANY` selector is reduced to `0` if the region might overflow because of linker-generated content. This is enabled and disabled independently of the sorting and placement algorithms.

`armlink` calculates a worst-case contingency for each section.

For `worst_fit`, `best_fit`, and `first_fit`, when a region is about to overflow because of the contingency, `armlink` lowers the priority of the related `.ANY` selector.

For `next_fit`, when a possible overflow is detected, `armlink` marks that section as `FULL` and does not consider it again. This stays consistent with the rule that when a section is full it can never be revisited.

Default

The default option is `worst_fit`.

Related concepts

7.4.5 Examples of using placement algorithms for .ANY sections on page 7-131

7.4.6 Example of next_fit algorithm showing behavior of full regions, selectors, and priority on page 7-133

7.4.8 Behavior when .ANY sections overflow because of linker-generated content on page 7-136

Related reference

11.3 --any_sort_order=order on page 11-221

11.59 --info=topic[,topic,...] on page 11-281

11.120 --scatter=filename on page 11-350

11.1 --any_contingency on page 11-218

7.4 Placement of unassigned sections on page 7-128

8.5.2 Syntax of an input section description on page 8-181

11.3 --any_sort_order=order

Controls the sort order of input sections that are placed using the `.ANY` module selector.

Syntax

`--any_sort_order=`*`order`*

where *`order`* is one of the following:

`descending_size`
: Sort input sections in descending size order.

`cmdline`
: The order that the section appears on the linker command-line. The command-line order is defined as `File.Object.Section` where:
  - `Section` is the section index, `sh_idx`, of the `Section` in the `Object`.
  - `Object` is the order that `Object` appears in the `File`.
  - `File` is the order the `File` appears on the command line.

  The order the `Object` appears in the `File` is only significant if the file is an `ar` archive.

By default, sections that have the same properties are resolved using the creation index. The `--tiebreaker` command-line option does not have any effect in the context of `--any_sort_order`.

Use this option with the `--scatter` option.

Usage

The sorting governs the order that sections are processed during `.ANY` assignment. Normal scatter-loading rules, for example `RO` before `RW`, are obeyed after the sections are assigned to regions.

Default

The default option is `--any_sort_order=descending_size`.

Related concepts

7.4.7 Examples of using sorting algorithms for .ANY sections on page 7-134

Related reference

11.59 --info=topic[,topic,...] on page 11-281

11.120 --scatter=filename on page 11-350

11.1 --any_contingency on page 11-218

7.4 Placement of unassigned sections on page 7-128

11.4 --api, --no_api

Enables and disables API section sorting. API sections are the sections that are called the most within a region.

Usage

In large region mode the API sections are extracted from the region and then inserted closest to the hotspots of the calling sections. This minimizes the number of veneers generated.

Default

The default is `--no_api`. The linker automatically switches to `--api` if at least one execution region contains more code than the smallest inter-section branch. The smallest inter-section branch depends on the code in the region and the target processor:

128MB
Execution region contains only A64 instructions.

32MB
Execution region contains only A32 instructions.

16MB
Execution region contains 32-bit T32 instructions.

4MB
Execution region contains only 16-bit T32 instructions.

Related concepts

3.6 Linker-generated veneers on page 3-53

Related reference

11.68 --largeregions, --no_largeregions on page 11-293

11.5 --autoat, --no_autoat

Controls the automatic assignment of `__at` sections to execution regions.

`__at` sections are sections that must be placed at a specific address.

Usage

If enabled, the linker automatically selects an execution region for each `__at` section. If a suitable execution region does not exist, the linker creates a load region and an execution region to contain the `__at` section.

If disabled, the standard scatter-loading section selection rules apply.

Default

The default is `--autoat`.

Restrictions

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

If you use `__at` sections with overlays, you cannot use `--autoat` to place those sections. You must specify the names of `__at` sections in a scatter file manually, and specify the `--no_autoat` option.

Related tasks

7.2.5 Placing __at sections at a specific address on page 7-120
7.2.7 Automatically placing __at sections on page 7-121
7.2.8 Manually placing __at sections on page 7-123

Related reference

8.2 Syntax of a scatter file on page 8-166

11.6 --bare_metal_pie

Specifies the bare-metal *Position Independent Executable* (PIE) linking model.

Note

Not supported for AArch64 state.

Note

The Bare-metal PIE feature is deprecated.

Default

The following default settings are automatically specified:

- `--fpic`.
- `--pie`.
- `--ref_pre_init`.

Related reference

11.51 --fpic on page 11-273
11.101 --pie on page 11-330
11.110 --ref_pre_init, --no_ref_pre_init on page 11-340

11.7 --base_platform

Specifies the Base Platform linking model. It is a superset of the *Base Platform Application Binary Interface* (BPABI) model, `--bpabi` option.

Note

Not supported for AArch64 state.

Usage

When you specify `--base_platform`, the linker also acts as if you specified `--bpabi` with the following exceptions:

- The full choice of memory models is available, including scatter-loading:
  - `--dll`.
  - `--force_so_throw`, `--no_force_so_throw`.
  - `--pltgot=`*`type`*.
  - `--rosplit`.

  Note

  If you do not specify a scatter file with `--scatter`, then the standard BPABI memory model scatter file is used.
- The default value of the `--pltgot` option is different to that for `--bpabi`:
  - For `--base_platform`, the default is `--pltgot=none`.
  - For `--bpabi` the default is `--pltgot=direct`.
- If you specify `--pltgot_opts=crosslr` then calls to and from a load region marked `RELOC` go by way of the *Procedure Linkage Table* (PLT).

To place unresolved weak references in the dynamic symbol table, use the `IMPORT` steering file command.

Note

If you are linking with `--base_platform`, and the parent load region has the `RELOC` attribute, then all execution regions within that load region must have a `+`*`offset`* base address.

Related concepts

10.2 Scatter files for the Base Platform linking model on page 10-211

2.4 Base Platform Application Binary Interface (BPABI) linking model on page 2-29

2.5 Base Platform linking model on page 2-30

8.3.5 Inheritance rules for the RELOC address attribute on page 8-171

Related reference

11.11 --bpabi on page 11-230

11.104 --pltgot=type on page 11-334

11.105 --pltgot_opts=mode on page 11-335

11.120 --scatter=filename on page 11-350

11.113 --remove, --no_remove on page 11-343

11.32 --dll on page 11-254

11.50 --force_so_throw, --no_force_so_throw on page 11-272

11.114 --ro_base=address on page 11-344

11.116 --rosplit on page 11-346

11.117 --rw_base=address on page 11-347

11.118 --rwpi on page 11-348

11.8 --bestdebug, --no_bestdebug

Selects between linking for smallest code and data size or for best debug illusion.

Usage

Input objects might contain *common data* (COMDAT) groups, but these might not be identical across all input objects because of differences such as objects compiled with different optimization levels.

Use `--bestdebug` to select COMDAT groups with the best debug view. Be aware that the code and data of the final image might not be the same when building with or without debug.

Default

The default is `--no_bestdebug`. The smallest COMDAT groups are selected when linking, at the expense of a possibly slightly poorer debug illusion.

Example

For two objects compiled with different optimization levels:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -c -O2 file1.c
armclang --target=arm-arm-none-eabi -march=armv8-a -c -O0 file2.c
armlink --bestdebug file1.o file2.o -o image.axf
```

Related concepts

4.1 Elimination of common section groups on page 4-66

4.2 Elimination of unused sections on page 4-67

Related reference

11.92 -o filename, --output=filename on page 11-321

11.9 --blx_arm_thumb, --no_blx_arm_thumb

Enables the linker to use the `BLX` instruction for A32 to T32 state changes.

Usage

If the linker cannot use `BLX` it must use an A32 to T32 interworking veneer to perform the state change.

This option is on by default. It has no effect if the target architecture does not support `BLX` or when linking for AArch64 state.

Related concepts

3.6.3 Veneer types on page 3-54

Related reference

11.10 --blx_thumb_arm, --no_blx_thumb_arm on page 11-229

11.10 --blx_thumb_arm, --no_blx_thumb_arm

Enables the linker to use the `BLX` instruction for T32 to A32 state changes.

Usage

If the linker cannot use `BLX` it must use a T32 to A32 interworking veneer to perform the state change.

This option is on by default. It has no effect if the target architecture does not support `BLX` or when linking for AArch64 state.

Related concepts

3.6.3 Veneer types on page 3-54

Related reference

11.9 --blx_arm_thumb, --no_blx_arm_thumb on page 11-228

11.11 --bpabi

Creates a *Base Platform Application Binary Interface* (BPABI) ELF file for passing to a platform-specific post-linker.

Note

Not supported for AArch64 state.

Usage

The BPABI model defines a standard-memory model that enables interoperability of BPABI-compliant files across toolchains. When you specify this option:

- *Procedure Linkage Table* (PLT) and *Global Offset Table* (GOT) generation is supported.
- The default value of the `--pltgot` option is `direct`.
- A *dynamically linked library* (DLL) placed on the command-line can define symbols.

Restrictions

The BPAPI model does not support scatter-loading. However, scatter-loading is supported in the Base Platform model.

Weak references in the dynamic symbol table are permitted only if the symbol is defined by a DLL placed on the command-line. You cannot place an unresolved weak reference in the dynamic symbol table with the `IMPORT` steering file command.

Related concepts

2.4 Base Platform Application Binary Interface (BPABI) linking model on page 2-29

2.5 Base Platform linking model on page 2-30

Related reference

11.7 --base_platform on page 11-225

11.113 --remove, --no_remove on page 11-343

11.32 --dll on page 11-254

11.104 --pltgot=type on page 11-334

Chapter 9 BPABI Shared Libraries and Executables on page 9-194

11.12 --branchnop, --no_branchnop

Enables or disables the replacement of any branch with a relocation that resolves to the next instruction with a `NOP`.

Usage

The default behavior is to replace any branch with a relocation that resolves to the next instruction with a `NOP`. However, there are cases where you might want to use `--no_branchnop` to disable this behavior. For example, when performing verification or pipeline flushes.

Default

The default is `--branchnop`.

Related concepts

4.6 About branches that optimize to a NOP on page 4-74

Related reference

11.62 --inline, --no_inline on page 11-286

11.141 --tailreorder, --no_tailreorder on page 11-373

11.13 --callgraph, --no_callgraph

Creates a file containing a static callgraph of functions.

The callgraph gives definition and reference information for all functions in the image.

Note

If you use the `--partial` option to create a partially linked object, then no callgraph file is created.

Usage

The callgraph file:

- Is saved in the same directory as the generated image.
- Has the name of the linked image with the extension, if any, replaced by the callgraph output extension, either `.htm` or `.txt`. Use the `--callgraph_file=`*`filename`* option to specify a different callgraph filename.
- Has a default output format of HTML. Use the `--callgraph_output=`*`fmt`* option to control the output format.

Note

If the linker is to calculate the function stack usage, any functions defined in the assembler files must have the appropriate:

- `.cfi_startproc` and `.cfi_endproc` directives.
- `.cfi_sections .debug_frame` directive.

The linker lists the following for each function `func`:

- Instruction set state for which the function is compiled (A32, T32, or A64).
- Set of functions that call `func`.
- Set of functions that are called by `func`.
- Number of times the address of `func` is used in the image.

In addition, the callgraph identifies functions that are:

- Called through interworking veneers.
- Defined outside the image.
- Permitted to remain undefined (weak references).
- Called through a *Procedure Linkage Table* (PLT).
- Not called but still exist in the image.

The static callgraph also gives information about stack usage. It lists the:

- Size of the stack frame used by each function.
- Maximum size of the stack used by the function over any call sequence, that is, over any acyclic chain of function calls.

If there is a cycle, or if the linker detects a function with no stack size information in the call chain, `+ Unknown` is added to the stack usage. A reason is added to indicate why stack usage is unknown.

The linker reports missing stack frame information if there is no debug frame information for the function.

For indirect functions, the linker cannot reliably determine which function made the indirect call. This might affect how the maximum stack usage is calculated for a call chain. The linker lists all function pointers used in the image.

Use frame directives in assembly language code to describe how your code uses the stack. These directives ensure that debug frame information is present for debuggers to perform stack unwinding or profiling.

Default

The default is `--no_callgraph`.

Related reference

11.14 --callgraph_file=filename on page 11-234
11.15 --callgraph_output=fmt on page 11-235
11.16 --callgraph_subset=symbol[,symbol,...] on page 11-236
11.17 --cgfile=type on page 11-237
11.18 --cgsymbol=type on page 11-238
11.19 --cgundefined=type on page 11-239
8.2 Syntax of a scatter file on page 8-166

11.14 --callgraph_file=filename

Controls the output filename of the callgraph.

Syntax

`--callgraph_file=`*`filename`*

where *`filename`* is the callgraph filename.

The default filename is the name of the linked image with the extension, if any, replaced by the callgraph output extension, either `.htm` or `.txt`.

Related reference

11.13 --callgraph, --no_callgraph on page 11-232
11.15 --callgraph_output=fmt on page 11-235
11.16 --callgraph_subset=symbol[,symbol,...] on page 11-236
11.17 --cgfile=type on page 11-237
11.18 --cgsymbol=type on page 11-238
11.19 --cgundefined=type on page 11-239
11.92 -o filename, --output=filename on page 11-321

11.15 --callgraph_output=fmt

Controls the output format of the callgraph.

Syntax

`--callgraph_output=`*`fmt`*

Where *`fmt`* can be one of the following:

`html`

Outputs the callgraph in HTML format.

`text`

Outputs the callgraph in plain text format.

Default

The default is `--callgraph_output=html`.

Related reference

11.13 --callgraph, --no_callgraph on page 11-232
11.14 --callgraph_file=filename on page 11-234
11.16 --callgraph_subset=symbol[,symbol,...] on page 11-236
11.17 --cgfile=type on page 11-237
11.18 --cgsymbol=type on page 11-238
11.19 --cgundefined=type on page 11-239

11.16 --callgraph_subset=symbol[,symbol,...]

Creates a file containing a static callgraph for one or more specified symbols.

Syntax

`--callgraph_subset=`*`symbol[,symbol,…]`*

where *`symbol`* is a comma-separated list of symbols.

Usage

The callgraph file:

- Is saved in the same directory as the generated image.
- Has the name of the linked image with the extension, if any, replaced by the callgraph output extension, either `.htm` or `.txt`. Use the `--callgraph_file=`*`filename`* option to specify a different callgraph filename.
- Has a default output format of HTML. Use the `--callgraph_output=`*`fmt`* option to control the output format.

Related reference

11.13 --callgraph, --no_callgraph on page 11-232
11.14 --callgraph_file=filename on page 11-234
11.15 --callgraph_output=fmt on page 11-235
11.17 --cgfile=type on page 11-237
11.18 --cgsymbol=type on page 11-238
11.19 --cgundefined=type on page 11-239

11.17 --cgfile=type

Controls the type of files to use for obtaining the symbols to be included in the callgraph.

Syntax

`--cgfile=`*`type`*

where *`type`* can be one of the following:

`all`

Includes symbols from all files.

`user`

Includes only symbols from user defined objects and libraries.

`system`

Includes only symbols from system libraries.

Default

The default is `--cgfile=all`.

Related reference

11.13 --callgraph, --no_callgraph on page 11-232
11.14 --callgraph_file=filename on page 11-234
11.15 --callgraph_output=fmt on page 11-235
11.16 --callgraph_subset=symbol[,symbol,...] on page 11-236
11.18 --cgsymbol=type on page 11-238
11.19 --cgundefined=type on page 11-239

11.18 --cgsymbol=type

Controls what symbols are included in the callgraph.

Syntax

`--cgsymbol=`*`type`*

Where *`type`* can be one of the following:

`all`

Includes both local and global symbols.

`locals`

Includes only local symbols.

`globals`

Includes only global symbols.

Default

The default is `--cgsymbol=all`.

Related reference

11.13 --callgraph, --no_callgraph on page 11-232
11.14 --callgraph_file=filename on page 11-234
11.15 --callgraph_output=fmt on page 11-235
11.16 --callgraph_subset=symbol[,symbol,...] on page 11-236
11.17 --cgfile=type on page 11-237
11.19 --cgundefined=type on page 11-239

11.19 --cgundefined=type

Controls what undefined references are included in the callgraph.

Syntax

`--cgundefined=`*`type`*

Where *`type`* can be one of the following:

`all`

Includes both function entries and calls to undefined weak references.

`entries`

Includes function entries for undefined weak references.

`calls`

Includes calls to undefined weak references.

`none`

Omits all undefined weak references from the output.

Default

The default is `--cgundefined=all`.

Related reference

11.13 --callgraph, --no_callgraph on page 11-232
11.14 --callgraph_file=filename on page 11-234
11.15 --callgraph_output=fmt on page 11-235
11.16 --callgraph_subset=symbol[,symbol,...] on page 11-236
11.17 --cgfile=type on page 11-237
11.18 --cgsymbol=type on page 11-238

11.20 --comment_section, --no_comment_section

Controls the inclusion of a comment section `.comment` in the final image.

Usage

Use `--no_comment_section` to remove the `.comment` section, to help reduce the image size.

Note

You can also use the `--filtercomment` option to merge comments.

Default

The default is `--comment_section`.

Related concepts

4.9 Linker merging of comment sections on page 4-77

Related reference

11.46 --filtercomment, --no_filtercomment on page 11-268

11.21 --cppinit, --no_cppinit

Enables the linker to use alternative C++ libraries with a different initialization symbol if required.

Syntax

`--cppinit=`*`symbol`*

Where *`symbol`* is the initialization symbol to use.

Usage

If you do not specify `--cppinit=`*`symbol`* then the default symbol `__cpp_initialize__aeabi_` is assumed.

`--no_cppinit` does not take a *`symbol`* argument.

Effect

The linker adds a non-weak reference to *`symbol`* if any static constructor or destructor sections are detected.

For `--cppinit=__cpp_initialize__aeabi_` in AArch32 state, the linker processes R_ARM_TARGET1 relocations as R_ARM_REL32, because this is required by the `__cpp_initialize__aeabi_` function. In all other cases R_ARM_TARGET1 relocations are processed as R_ARM_ABS32.

Note

There is no equivalent of R_ARM_TARGET1 in AARCH64 state.

`--no_cppinit` means do not add a reference.

Related reference

11.109 --ref_cpp_init, --no_ref_cpp_init on page 11-339

11.22 --cpu=list

Lists the architecture and processor names that are supported by the `--cpu=name` option.

Syntax

```
--cpu=list
```

Related reference

11.23 --cpu=name on page 11-243
11.52 --fpu=list on page 11-274
11.53 --fpu=name on page 11-275

11.23 --cpu=name

Enables code generation for the selected Arm processor or architecture.

If you do not include the `--cpu` option, `armlink` derives an architecture from the combination of the input objects.

If you include `--cpu=`*name*, `armlink`:

- Faults any input object that is not compatible with the cpu.
- For library selection, acts as if at least one input object is compiled with `--cpu=`*name*.

Note

You cannot specify targets with Armv8.4-A or later architectures on the `armlink` command-line. To link for such targets, you must not specify the `--cpu` option when invoking `armlink` directly.

Syntax

`--cpu=`*name*

Where *name* is the name of a processor or architecture:

Processor and architecture names are not case-sensitive.

Wildcard characters are not accepted.

The following table shows the supported architectures. For a complete list of the supported architecture and processor names, specify the `--cpu=list` option.

Table 11-1 Supported Arm architectures

Architecture name	Description
`6-M`	Armv6 architecture microcontroller profile.
`6S-M`	Armv6 architecture microcontroller profile with OS extensions.
`7-A`	Armv7 architecture application profile.
`7-A.security`	Armv7-A architecture profile with Security Extensions and includes the `SMC` instruction (formerly `SMI`).
`7-R`	Armv7 architecture real-time profile.
`7-M`	Armv7 architecture microcontroller profile.
`7E-M`	Armv7-M architecture profile with DSP extension.
`8-A.32`	Armv8-A architecture profile, AArch32 state.
`8-A.32.crypto`	Armv8-A architecture profile, AArch32 state with cryptographic instructions.
`8-A.64`	Armv8-A architecture profile, AArch64 state.
`8-A.64.crypto`	Armv8-A architecture profile, AArch64 state with cryptographic instructions.
`8.1-A.32`	Armv8.1, for Armv8-A architecture profile, AArch32 state.
`8.1-A.32.crypto`	Armv8.1, for Armv8-A architecture profile, AArch32 state with cryptographic instructions.
`8.1-A.64`	Armv8.1, for Armv8-A architecture profile, AArch64 state.
`8.1-A.64.crypto`	Armv8.1, for Armv8-A architecture profile, AArch64 state with cryptographic instructions.
`8.2-A.32`	Armv8.2, for Armv8-A architecture profile, AArch32 state.

Table 11-1 Supported Arm architectures (continued)

Architecture name	Description
`8.2-A.32.crypto`	Armv8.2, for Armv8-A architecture profile, AArch32 state with cryptographic instructions.
`8.2-A.32.crypto.dotprod`	Armv8.2, for Armv8-A architecture profile, AArch32 state with cryptographic instructions and the `VSDOT` and `VUDOT` instructions.
`8.2-A.32.dotprod`	Armv8.2, for Armv8-A architecture profile, AArch32 state with the `VSDOT` and `VUDOT` instructions.
`8.2-A.64`	Armv8.2, for Armv8-A architecture profile, AArch64 state.
`8.2-A.64.crypto`	Armv8.2, for Armv8-A architecture profile, AArch64 state with cryptographic instructions.
`8.2-A.64.crypto.dotprod`	Armv8.2, for Armv8-A architecture profile, AArch64 state with cryptographic instructions and the `SDOT` and `UDOT` instructions.
`8.2-A.64.dotprod`	Armv8.2, for Armv8-A architecture profile, AArch64 state with the `SDOT` and `UDOT` instructions.
`8.3-A.32`	Armv8.3, for Armv8-A architecture profile, AArch32 state.
`8.3-A.32.crypto`	Armv8.3, for Armv8-A architecture profile, AArch32 state with cryptographic instructions.
`8.3-A.32.crypto.dotprod`	Armv8.3, for Armv8-A architecture profile, AArch32 state with cryptographic instructions and the `VSDOT` and `VUDOT` instructions.
`8.3-A.32.dotprod`	Armv8.3, for Armv8-A architecture profile, AArch32 state with the `VSDOT` and `VUDOT` instructions.
`8.3-A.64`	Armv8.3, for Armv8-A architecture profile, AArch64 state.
`8.3-A.64.crypto`	Armv8.3, for Armv8-A architecture profile, AArch64 state with cryptographic instructions.
`8.3-A.64.crypto.dotprod`	Armv8.3, for Armv8-A architecture profile, AArch64 state with cryptographic instructions and the `SDOT` and `UDOT` instructions.
`8.3-A.64.dotprod`	Armv8.3, for Armv8-A architecture profile, AArch64 state with the `SDOT` and `UDOT` instructions.
`8-R`	Armv8-R architecture profile.
`8-M.Base`	Armv8-M baseline architecture profile. Derived from the Armv6-M architecture.
`8-M.Main`	Armv8-M mainline architecture profile. Derived from the Armv7-M architecture.
`8-M.Main.dsp`	Armv8-M mainline architecture profile with DSP extension.

Note

- The full list of supported architectures and processors depends on your license.

Usage

If you omit `--cpu`, the linker auto-detects the processor or architecture from the input object files.

Specify `--cpu=list` to list the supported processor and architecture names that you can use with `--cpu=name`.

The link phase fails if any of the component object files rely on features that are incompatible with the specified processor. The linker also uses this option to optimize the choice of system libraries and any veneers that have to be generated when building the final image.

Restrictions

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

Related reference

11.22 --cpu=list on page 11-242
11.52 --fpu=list on page 11-274
11.53 --fpu=name on page 11-275

11.24 --crosser_veneershare, --no_crosser_veneershare

Enables or disables veneer sharing across execution regions.

Usage

The default is `--crosser_veneershare`, and enables veneer sharing across execution regions.

`--no_crosser_veneershare` prohibits veneer sharing across execution regions.

Related reference

11.152 --veneershare, --no_veneershare on page 11-384

11.25 --datacompressor=opt

Enables you to specify one of the supplied algorithms for RW data compression.

Note

Not supported for AArch64 state.

Syntax

`--datacompressor=opt`

Where *opt* is one of the following:

on

Enables RW data compression to minimize ROM size.

off

Disables RW data compression.

list

Lists the data compressors available to the linker.

id

A data compression algorithm:

Table 11-2 Data compressor algorithms

id	Compression algorithm
0	run-length encoding
1	run-length encoding, with LZ77 on small-repeats
2	complex LZ77 compression

Specifying a compressor adds a decompressor to the code area. If the final image does not have compressed data, the decompressor is not added.

Usage

If you do not specify a data compression algorithm, the linker chooses the most appropriate one for you automatically. In general, it is not necessary to override this choice.

Default

The default is `--datacompressor=on`.

Related concepts

4.3.3 How compression is applied on page 4-69

4.3.4 Considerations when working with RW data compression on page 4-69

4.3.1 How the linker chooses a compressor on page 4-68

11.26 --debug, --no_debug

Controls the generation of debug information in the output file.

Usage

Debug information includes debug input sections and the symbol/string table.

Use `--no_debug` to exclude debug information from the output file. The resulting ELF image is smaller, but you cannot debug it at source level. The linker discards any debug input section it finds in the input objects and library members, and does not include the symbol and string table in the image. This only affects the image size as loaded into the debugger. It has no effect on the size of any resulting binary image that is downloaded to the target.

If you are using `--partial` the linker creates a partially-linked object without any debug data.

Note

Do not use `--no_debug` if a `fromelf--fieldoffsets` step is required. If your image is produced without debug information, `fromelf` cannot:

- Translate the image into other file formats.
- Produce a meaningful disassembly listing.

Default

The default is `--debug`.

Related information

--fieldoffsets fromelf option

11.27 --diag_error=tag[,tag,…]

Sets diagnostic messages that have a specific tag to Error severity.

Syntax

`--diag_error=tag[,tag,…]`
Where *tag* can be:

- A diagnostic message number to set to error severity. This is the four-digit number, *nnnn*, with the tool letter prefix, but without the letter suffix indicating the severity.
- `warning`, to treat all warnings as errors.

Related reference

11.28 --diag_remark=tag[,tag,...] on page 11-250
11.29 --diag_style=arm|ide|gnu on page 11-251
11.30 --diag_suppress=tag[,tag,...] on page 11-252
11.31 --diag_warning=tag[,tag,...] on page 11-253
11.130 --strict on page 11-362

11.28 --diag_remark=tag[,tag,…]

Sets diagnostic messages that have a specific tag to Remark severity.

Note

Remarks are not displayed by default. Use the `--remarks` option to display these messages.

Syntax

`--diag_remark=`*`tag[,tag,…]`*

Where *`tag`* is a comma-separated list of diagnostic message numbers. This is the four-digit number, *nnnn*, with the tool letter prefix, but without the letter suffix indicating the severity.

Related reference

11.27 --diag_error=tag[,tag,...] on page 11-249
11.29 --diag_style=arm|ide|gnu on page 11-251
11.30 --diag_suppress=tag[,tag,...] on page 11-252
11.31 --diag_warning=tag[,tag,...] on page 11-253
11.112 --remarks on page 11-342
11.130 --strict on page 11-362

11.29 --diag_style=arm|ide|gnu

Specifies the display style for diagnostic messages.

Syntax

`--diag_style=`*`string`*

Where *`string`* is one of:

`arm`
: Display messages using the legacy Arm compiler style.

`ide`
: Include the line number and character count for any line that is in error. These values are displayed in parentheses.

`gnu`
: Display messages in the format used by `gcc`.

Default

The default is `--diag_style=arm`.

Usage

`--diag_style=gnu` matches the format reported by the GNU Compiler, `gcc`.

`--diag_style=ide` matches the format reported by Microsoft Visual Studio.

Related reference

11.27 --diag_error=tag[,tag,...] on page 11-249
11.28 --diag_remark=tag[,tag,...] on page 11-250
11.30 --diag_suppress=tag[,tag,...] on page 11-252
11.31 --diag_warning=tag[,tag,...] on page 11-253
11.112 --remarks on page 11-342
11.130 --strict on page 11-362

11.30 --diag_suppress=tag[,tag,…]

Suppresses diagnostic messages that have a specific tag.

Syntax

`--diag_suppress=`*`tag[,tag,…]`*
Where *`tag`* can be:

- A diagnostic message number to be suppressed. This is the four-digit number, *nnnn*, with the tool letter prefix, but without the letter suffix indicating the severity.
- `error`, to suppress all errors that can be downgraded.
- `warning`, to suppress all warnings.

Example

To suppress the warning messages that have numbers `L6314W` and `L6305W`, use the following command:

```
armlink --diag_suppress=L6314,L6305 …
```

Related reference

11.27 --diag_error=tag[,tag,...] on page 11-249
11.28 --diag_remark=tag[,tag,...] on page 11-250
11.29 --diag_style=arm|ide|gnu on page 11-251
11.31 --diag_warning=tag[,tag,...] on page 11-253
11.130 --strict on page 11-362
11.112 --remarks on page 11-342

11.31 --diag_warning=tag[,tag,…]

Sets diagnostic messages that have a specific tag to Warning severity.

Syntax

`--diag_warning=`*`tag[,tag,…]`*
Where *`tag`* can be:

- A diagnostic message number to set to warning severity. This is the four-digit number, *nnnn*, with the tool letter prefix, but without the letter suffix indicating the severity.
- `error`, to set all errors that can be downgraded to warnings.

Related reference

11.27 --diag_error=tag[,tag,...] on page 11-249
11.28 --diag_remark=tag[,tag,...] on page 11-250
11.29 --diag_style=arm|ide|gnu on page 11-251
11.30 --diag_suppress=tag[,tag,...] on page 11-252
11.112 --remarks on page 11-342

11.32 --dll

Creates a *Base Platform Application Binary Interface* (BPABI) *dynamically linked library* (DLL).

Note

Not supported for AArch64 state.

Usage

The DLL is marked as a shared object in the ELF file header.

You must use `--bpabi` with `--dll` to produce a BPABI-compliant DLL.

You can also use `--dll` with `--base_platform`.

Note

By default, this option disables unused section elimination. Use the `--remove` option to re-enable unused sections when building a DLL.

Related reference

11.113 --remove, --no_remove on page 11-343
11.11 --bpabi on page 11-230
Chapter 9 BPABI Shared Libraries and Executables on page 9-194

11.33 --dynamic_linker=name

Specifies the dynamic linker to use to load and relocate the file at runtime.

Note

Not supported for AArch64 state.

Syntax

`--dynamic_linker=`*`name`*

`--dynamiclinker=`*`name`*

Where *`name`* is the name of the dynamic linker to store in the executable.

Usage

When you link with shared objects, the dynamic linker to use is stored in the executable. This option specifies a particular dynamic linker to use when the file is executed.

Related reference

11.47 --fini=symbol on page 11-269
11.61 --init=symbol on page 11-285
11.72 --library=name on page 11-298
Chapter 9 BPABI Shared Libraries and Executables on page 9-194

11.34 --eager_load_debug, --no_eager_load_debug

Manages how `armlink` loads debug section data.

Usage

The `--no_eager_load_debug` option causes the linker to remove debug section data from memory after object loading. This lowers the peak memory usage of the linker at the expense of some linker performance, because much of the debug data has to be loaded again when the final image is written.

Using `--no_eager_load_debug` option does not affect the debug data that is written into the ELF file.

The default is `--eager_load_debug`.

Note

If you use some command-line options, such as `--map`, the resulting image or object built without debug information might differ by a small number of bytes. This is because the `.comment` section contains the linker command line used, where the options have differed from the default. Therefore `--no_eager_load_debug` images are a little larger and contain Program Header and possibly a section header a small number of bytes later. Use `--no_comment_section` to eliminate this difference.

Related reference

11.20 --comment_section, --no_comment_section on page 11-240

11.35 --eh_frame_hdr

When an AArch64 image contains C++ exceptions, merges all `.eh_frame` sections into one `.eh_frame` section and then creates the `.eh_frame_hdr` section.

Usage

The `.eh_frame_hdr` section contains a binary search table of pointers to the `.eh_frame` records. During the merge `armlink` removes any orphaned records.

Only `.eh_frame` sections defined by the *Linux Standard Base* specification are supported.
The `.eh_frame_hdr` section is created according to the *Linux Standard Base* specification. If `armlink` finds an unexpected `.eh_frame` section, it stops merging, does not create the `.eh_frame_hdr` section, and generates corresponding warnings.

Default

The default is `--eh_frame_hdr`.

Restrictions

Valid only for AArch64 images.

Related information

Linux Foundation

11.36 --edit=file_list

Enables you to specify steering files containing commands to edit the symbol tables in the output binary.

Syntax

`--edit=file_list`

Where *`file_list`* can be more than one steering file separated by a comma. Do not include a space after the comma.

Usage

You can specify commands in a steering file to:

- Hide global symbols. Use this option to hide specific global symbols in object files. The hidden symbols are not publicly visible.
- Rename global symbols. Use this option to resolve symbol naming conflicts.

Examples

```
--edit=file1 --edit=file2 --edit=file3
```

```
--edit=file1,file2,file3
```

Related concepts

6.6.4 Hide and rename global symbols with a steering file on page 6-102

Related reference

6.6.2 Steering file command summary on page 6-100

Chapter 12 Linker Steering File Command Reference on page 12-394

11.37 --emit_debug_overlay_relocs

Outputs only relocations of debug sections with respect to overlaid program sections to aid an overlay-aware debugger.

Note

Not supported for AArch64 state.

Related reference

11.38 --emit_debug_overlay_section on page 11-260
11.40 --emit_relocs on page 11-262
11.39 --emit_non_debug_relocs on page 11-261

Related information

Manual overlay support
ABI for the Arm Architecture: Support for Debugging Overlaid Programs

11.38 --emit_debug_overlay_section

Emits a special debug overlay section during static linking.

Note

Not supported for AArch64 state.

Usage

In a relocatable file, a debug section refers to a location in a program section by way of a relocated location. A reference from a debug section to a location in a program section has the following format:

```
<debug_section_index, debug_section_offset>, <program_section_index,
program_section_offset>
```

During static linking the pair of *program* values is reduced to single value, the execution address. This is ambiguous in the presence of overlaid sections.

To resolve this ambiguity, use this option to output a `.ARM.debug_overlay` section of type `SHT_ARM_DEBUG_OVERLAY = SHT_LOUSER + 4` containing a table of entries as follows:

debug_section_offset, debug_section_index, program_section_index

Related reference

11.37 --emit_debug_overlay_relocs on page 11-259
11.40 --emit_relocs on page 11-262

Related information

Automatic overlay support
Manual overlay support
ABI for the Arm Architecture: Support for Debugging Overlaid Programs

11.39 --emit_non_debug_relocs

Retains only relocations from non-debug sections in an executable file.

Note

Not supported for AArch64 state.

Related reference

11.40 --emit_relocs on page 11-262

11.40 --emit_relocs

Retains all relocations in the executable file. This results in larger executable files.

Note

Not supported for AArch64 state.

Usage

This is equivalent to the GNU ld `--emit-relocs` option.

Related reference

11.37 --emit_debug_overlay_relocs on page 11-259
11.39 --emit_non_debug_relocs on page 11-261

Related information

ABI for the Arm Architecture: Support for Debugging Overlaid Programs

11.41 --entry=location

Specifies the unique initial entry point of the image. Although an image can have multiple entry points, only one can be the initial entry point.

Syntax

`--entry=location`
Where *location* is one of the following:
entry_address

A numerical value, for example: `--entry=0x0`

symbol

Specifies an image entry point as the address of *symbol*, for example: `--entry=reset_handler`

offset+object(section)

Specifies an image entry point as an *offset* inside a *section* within a particular *object*, for example:`--entry=8+startup.o(startupseg)`

There must be no spaces within the argument to `--entry`. The input section and object names are matched without case-sensitivity. You can use the following simplified notation:

- `object(section)`, if `offset` is zero.
- `object`, if there is only one input section. `armlink` generates an error message if there is more than one code input section in `object`.

Note

If the entry address of your image is in T32 state, then the least significant bit of the address must be set to 1. The linker does this automatically if you specify a symbol. For example, if the entry code starts at address `0x8000` in T32 state you must use `--entry=0x8001`.

Usage

The image can contain multiple entry points. Multiple entry points might be specified with the `ENTRY` directive in assembler source files. In such cases, a unique initial entry point must be specified for an image, otherwise the error L6305E is generated. The initial entry point specified with the `--entry` option is stored in the executable file header for use by the loader. There can be only one occurrence of this option on the command line. A debugger typically uses this entry address to initialize the *Program Counter* (PC) when an image is loaded. The initial entry point must meet the following conditions:

- The image entry point must lie within an execution region.
- The execution region must be non-overlay, and must be a root execution region (load address == execution address).

Related reference

11.128 --startup=symbol, --no_startup on page 11-360

Related information

ENTRY directive

11.42 --errors=filename

Redirects the diagnostics from the standard error stream to a specified file.

Syntax

`--errors=filename`

Usage

The specified file is created at the start of the link stage. If a file of the same name already exists, it is overwritten.

If `filename` is specified without path information, the file is created in the current directory.

Related reference

11.27 --diag_error=tag[,tag,...] on page 11-249
11.28 --diag_remark=tag[,tag,...] on page 11-250
11.29 --diag_style=arm|ide|gnu on page 11-251
11.30 --diag_suppress=tag[,tag,...] on page 11-252
11.31 --diag_warning=tag[,tag,...] on page 11-253
11.112 --remarks on page 11-342

11.43 --exceptions, --no_exceptions

Controls the generation of exception tables in the final image.

Usage

Using `--no_exceptions` generates an error message if any exceptions sections are present in the image after unused sections have been eliminated. Use this option to ensure that your code is exceptions free.

Default

The default is `--exceptions`.

11.44 --export_all, --no_export_all

Controls the export of all global, non-hidden symbols to the dynamic symbols table.

Usage

Use `--export_all` to dynamically export all global, non-hidden symbols from the executable or DLL to the dynamic symbol table. Use `--no_export_all` to prevent the exporting of symbols to the dynamic symbol table.

`--export_all` always exports non-hidden symbols into the dynamic symbol table. The dynamic symbol table is created if necessary.

You cannot use `--export_all` to produce a statically linked image because it always exports non-hidden symbols, forcing the creation of a dynamic segment.

For more precise control over the exporting of symbols, use one or more steering files.

Default

The default is `--export_all` for building shared libraries and *dynamically linked libraries* (DLLs).

The default is `--no_export_all` for building applications.

Related reference

11.45 --export_dynamic, --no_export_dynamic on page 11-267

11.45 --export_dynamic, --no_export_dynamic

Controls the export of dynamic symbols to the dynamic symbols table.

Note

Not supported for AArch64 state.

Usage

If an executable has dynamic symbols, then `--export_dynamic` exports all externally visible symbols.

`--export_dynamic` exports non-hidden symbols into the dynamic symbol table only if a dynamic symbol table already exists.

You can use `--export_dynamic` to produce a statically linked image if there are no imports or exports.

Default

`--no_export_dynamic` is the default.

Related reference

11.44 --export_all, --no_export_all on page 11-266

11.46 --filtercomment, --no_filtercomment

Controls whether or not the linker modifies the `.comment` section to assist merging.

Usage

The linker always removes identical comments. The `--filtercomment` permits the linker to preprocess the `.comment` section and remove some information that prevents merging.

Use `--no_filtercomment` to prevent the linker from modifying the `.comment` section.

Note

`armlink` does not preprocess comment sections from `armclang`.

Default

The default is `--filtercomment`.

Related concepts

4.9 Linker merging of comment sections on page 4-77

Related reference

11.20 --comment_section, --no_comment_section on page 11-240

11.47 --fini=symbol

Specifies the symbol name to use to define the entry point for finalization code.

Syntax

`--fini=symbol`

Where *symbol* is the symbol name to use for the entry point to the finalization code.

Usage

The dynamic linker executes this code when it unloads the executable file or shared object.

Related reference

11.33 --dynamic_linker=name on page 11-255
11.61 --init=symbol on page 11-285
11.72 --library=name on page 11-298

11.48 --first=section_id

Places the selected input section first in its execution region. This can, for example, place the section containing the vector table first in the image.

Syntax

`--first=section_id`

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

`symbol`

Selects the section that defines *`symbol`*. For example: `--first=reset`.

You must not specify a symbol that has more than one definition, because only one section can be placed first.

`object``(`*`section`*`)`

Selects *`section`* from *`object`*. There must be no space between *`object`* and the following open parenthesis. For example: `--first=init.o(init)`.

`object`

Selects the single input section in *`object`*. For example: `--first=init.o`.

If you use this short form and there is more than one input section in *`object`*, `armlink` generates an error message.

Usage

The `--first` option cannot be used with `--scatter`. Instead, use the `+FIRST` attribute in a scatter file.

Related concepts

3.3.2 Section placement with the FIRST and LAST attributes on page 3-49

3.3 Section placement with the linker on page 3-48

Related reference

11.69 --last=section_id on page 11-295

11.120 --scatter=filename on page 11-350

11.49 --force_explicit_attr

Causes the linker to retry the CPU mapping using build attributes constructed when an architecture is specified with `--cpu`.

Usage

The `--cpu` option checks the FPU attributes if the CPU chosen has a built-in FPU.

The error message `L6463U`: `Input Objects contain <archtype> instructions but could not find valid target for <archtype> architecture based on object attributes. Suggest using --cpu option to select a specific cpu.` is given in the following situations:

- The ELF file contains instructions from architecture *`archtype`* yet the build attributes claim that *`archtype`* is not supported.
- The build attributes are inconsistent enough that the linker cannot map them to an existing CPU.

If setting the `--cpu` option still fails, use `--force_explicit_attr` to cause the linker to retry the CPU mapping using build attributes constructed from `--cpu=`*`archtype`*. This might help if the error is being given solely because of inconsistent build attributes.

Related reference

11.23 --cpu=name on page 11-243
11.53 --fpu=name on page 11-275

11.50 --force_so_throw, --no_force_so_throw

Controls the assumption made by the linker that an input shared object might throw an exception.

Note

Not supported for AArch64 state.

Usage

By default, exception tables are discarded if no code throws an exception.

Use `--force_so_throw` to specify that all shared objects might throw an exception and so force the linker to keep the exception tables, regardless of whether the image can throw an exception or not.

Default

The default is `--no_force_so_throw`.

11.51 --fpic

Enables you to link *Position-Independent Code* (PIC), that is, code that has been compiled using the `-fbare-metal-pie` or `-fpic` compiler command-line options.

The `--fpic` option is implicitly specified when the `--bare_metal_pie` option is used.

Note

The Bare-metal PIE feature is deprecated.

Related reference

11.6 --bare_metal_pie on page 11-224

11.52 --fpu=list

Lists the *Floating Point Unit* (FPU) architectures that are supported by the `--fpu=name` option.

Deprecated options are not listed.

Syntax

```
--fpu=list
```

Related reference

11.22 --cpu=list on page 11-242
11.23 --cpu=name on page 11-243
11.53 --fpu=name on page 11-275

11.53 --fpu=name

Specifies the target FPU architecture.

Syntax

`--fpu=`*name*

Where *name* is the name of the target FPU architecture. Specify `--fpu=list` to list the supported FPU architecture names that you can use with `--fpu=name`.

The default floating-point architecture depends on the target architecture.

Note

Software floating-point linkage is not supported for AArch64 state.

Usage

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

The linker uses this option to optimize the choice of system libraries. The default is to select an FPU that is compatible with all of the component object files.

The linker fails if any of the component object files rely on features that are incompatible with the selected FPU architecture.

Restrictions

Arm NEON™ support is disabled for `SoftVFP`.

Default

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

If the processor you specify with `--cpu` has a VFP coprocessor, the default target FPU architecture is the VFP architecture for that processor.

Related reference

11.22 --cpu=list on page 11-242
11.23 --cpu=name on page 11-243
11.52 --fpu=list on page 11-274

11.54 --got=type

Generates *Global Offset Tables* (GOTs) to resolve GOT relocations in bare metal images. `armlink` statically resolves the GOT relocations.

Syntax

`--got=type`

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

`none`
Disables GOT generation.

`local`
Creates a local offset table for each execution region.

Note

Not supported for AArch32 state.

`global`
Creates a single offset table for the whole image.

Default

The default for AArch32 state is `none`.

The default for AArch64 state is `local`.

11.55 --gnu_linker_defined_syms

Enables support for the GNU equivalent of input section symbols.

Note

The `--gnu_linker_defined_syms` linker option is deprecated.

Usage

If you want GNU-style behavior when treating the Arm symbols `SectionName$$Base` and `SectionName$$Limit`, then specify `--gnu_linker_defined_syms`.

Table 11-3 GNU equivalent of input sections

GNU symbol	Arm symbol	Description
`__start_`*`SectionName`*	*`SectionName`*`$$Base`	Address of the start of the consolidated section called *`SectionName`*.
`__stop_`*`SectionName`*	*`SectionName`*`$$Limit`	Address of the byte beyond the end of the consolidated section called *`SectionName`*

Note

- A reference to *`SectionName`* by a GNU input section symbol is sufficient for `armlink` to prevent the section from being removed as unused.
- A reference by an Arm input section symbol is not sufficient to prevent the section from being removed as unused.

11.56 --help

Displays a summary of the main command-line options.

Default

This is the default if you specify `armlink` without any options or source files.

Related reference

11.154 --version_number on page 11-386

11.156 --vsn on page 11-388

11.122 --show_cmdline on page 11-353

11.57 --import_cmse_lib_in=filename

Reads an existing import library and creates gateway veneers with the same address as given in the import library. This option is useful when producing a new version of a Secure image where the addresses in the output import library must not change. It is optional for a Secure image.

Syntax

`--import_cmse_lib_in=`*`filename`*

Where *`filename`* is the name of the import library file.

Usage

The input import library is an object file that contains only a symbol table. Each symbol specifies an absolute address of a secure gateway veneer for an entry function of the same name as the symbol.

`armlink` places secure gateway veneers generated from an existing import library using the `__at` feature. New secure gateway veneers must be placed using a scatter file.

Related concepts

3.6.6 Generation of secure gateway veneers on page 3-56

Related reference

11.58 --import_cmse_lib_out=filename on page 11-280

Related information

Building Secure and Non-secure Images Using Armv8-M Security Extensions

11.58 --import_cmse_lib_out=filename

Outputs the secure code import library to the location specified. This option is required for a Secure image.

Syntax

`--import_cmse_lib_out=`*`filename`*

Where *`filename`* is the name of the import library file.

The output import library is an object file that contains only a symbol table. Each symbol specifies an absolute address of a secure gateway for an entry function of the same name as the symbol. Secure gateways include both secure gateway veneers generated by `armlink` and any other secure gateways for entry functions found in the image.

Related concepts

3.6.6 Generation of secure gateway veneers on page 3-56

Related reference

11.57 --import_cmse_lib_in=filename on page 11-279

Related information

Building Secure and Non-secure Images Using Armv8-M Security Extensions

11.59 --info=topic[,topic,…]

Prints information about specific topics. You can write the output to a text file using `--list=file`.

Syntax

`--info=topic[,topic,…]`

Where *topic* is a comma-separated list from the following topic keywords:

any
For unassigned sections that are placed using the `.ANY` module selector, lists:
- The sort order.
- The placement algorithm.
- The sections that are assigned to each execution region in the order that the placement algorithm assigns them.
- Information about the contingency space and policy that is used for each region.

This keyword also displays additional information when you use the execution region attribute `ANY_SIZE` in a scatter file.

architecture
Summarizes the image architecture by listing the processor, FPU, and byte order.

common
Lists all common sections that are eliminated from the image. Using this option implies `--info=common,totals.`

compression
Gives extra information about the RW compression process.

debug
Lists all rejected input debug sections that are eliminated from the image as a result of using `--remove`. Using this option implies `--info=debug,totals`.

exceptions
Gives information on exception table generation and optimization.

inline
If you also specify `--inline`, lists all functions that the linker inlines, and the total number inlined.

inputs
Lists the input symbols, objects, and libraries.

libraries
Lists the full path name of every library the link stage automatically selects.

You can use this option with `--info_lib_prefix` to display information about a specific library.

merge
Lists the **const** strings that the linker merges. Each item lists the merged result, the strings being merged, and the associated object files.

pltgot
Lists the PLT entries that are built for the executable or DLL.

sizes
Lists the code and data (RO Data, RW Data, ZI Data, and Debug Data) sizes for each input object and library member in the image. Using this option implies `--info=sizes,totals`.

stack
Lists the stack usage of all functions.

summarysizes
Summarizes the code and data sizes of the image.

summarystack
Summarizes the stack usage of all global symbols.

tailreorder
Lists all the tail calling sections that are moved above their targets, as a result of using `--tailreorder`.

totals
Lists the totals of the code and data (RO Data, RW Data, ZI Data, and Debug Data) sizes for input objects and libraries.

unused
Lists all unused sections that are eliminated from the user code as a result of using `--remove`. It does not list any unused sections that are loaded from the Arm C libraries.

unusedsymbols
Lists all symbols that unused section elimination removes.

veneers
Lists the linker-generated veneers.

veneercallers
Lists the linker-generated veneers with additional information about the callers to each veneer. Use with `--verbose` to list each call individually.

veneerpools
Displays information on how the linker has placed veneer pools.

visibility
Lists the symbol visibility information. You can use this option with either `--info=inputs` or `--verbose` to enhance the output.

weakrefs
Lists all symbols that are the target of weak references, and whether they were defined.

Usage

The output from `--info=sizes,totals` always includes the padding values in the totals for input objects and libraries.

If you are using RW data compression (the default), or if you have specified a compressor using the `--datacompressor=id` option, the output from `--info=sizes,totals` includes an entry under `Grand Totals` to reflect the true size of the image.

Note

Spaces are not permitted between topic keywords in the list. For example, you can enter `--info=sizes,totals` but not `--info=sizes, totals`.

Related concepts

4.2 Elimination of unused sections on page 4-67
4.3.4 Considerations when working with RW data compression on page 4-69
4.3 Optimization with RW data compression on page 4-68
4.3.1 How the linker chooses a compressor on page 4-68
4.3.3 How compression is applied on page 4-69

Related reference

11.1 --any_contingency on page 11-218
11.3 --any_sort_order=order on page 11-221
11.60 --info_lib_prefix=opt on page 11-284
11.89 --merge, --no_merge on page 11-318
11.150 --veneer_inject_type=type on page 11-382
5.1 Options for getting information about linker-generated files on page 5-81
7.4 Placement of unassigned sections on page 7-128
11.25 --datacompressor=opt on page 11-247
11.62 --inline, --no_inline on page 11-286
11.113 --remove, --no_remove on page 11-343

11.67 --keep_intermediate on page 11-292
11.141 --tailreorder, --no_tailreorder on page 11-373
8.4.3 Execution region attributes on page 8-175

11.60 --info_lib_prefix=opt

Specifies a filter for the `--info=libraries` option. The linker only displays the libraries that have the same prefix as the filter.

Syntax

`--info=libraries --info_lib_prefix=opt`

Where *opt* is the prefix of the required library.

Examples

- Displaying a list of libraries without the filter:

```
armlink --info=libraries test.o
```

  Produces a list of libraries, for example:

```
install_directory\lib\armlib\c_4.l
install_directory\lib\armlib\fz_4s.l
install_directory\lib\armlib\h_4.l
install_directory\lib\armlib\m_4s.l
install_directory\lib\armlib\vfpsupport.l
```

- Displaying a list of libraries with the filter:

```
armlink --info=libraries --info_lib_prefix=c test.o
```

  Produces a list of libraries with the specified prefix, for example:

```
install_directory\lib\armlib\c_4.l
```

Related reference

11.59 --info=topic[,topic,...] on page 11-281

11.61 --init=symbol

Specifies a symbol name to use for the initialization code. A dynamic linker executes this code when it loads the executable file or shared object.

Syntax

`--init=symbol`

Where *symbol* is the symbol name you want to use to define the location of the initialization code.

Related reference

11.33 --dynamic_linker=name on page 11-255
11.47 --fini=symbol on page 11-269
11.72 --library=name on page 11-298

11.62 --inline, --no_inline

Enables or disables branch inlining to optimize small function calls in your image.

Note

Not supported for AArch64 state.

Default

The default is `--no_inline`.

Note

This branch optimization is off by default because enabling it changes the image such that debug information might be incorrect. If enabled, the linker makes no attempt to correct the debug information.

`--no_inline` turns off inlining for user-supplied objects only. The linker still inlines functions from the Arm standard libraries by default.

Related concepts

4.4 Function inlining with the linker on page 4-71

Related reference

11.12 --branchnop, --no_branchnop on page 11-231

11.63 --inline_type=type on page 11-287

11.141 --tailreorder, --no_tailreorder on page 11-373

11.63 --inline_type=type

Inlines functions from all objects, Arm standard libraries only, or turns off inlining completely.

Syntax

`--inline_type=`*`type`*

Where *`type`* is one of:

`all`
The linker is permitted to inline functions from all input objects.

`library`
The linker is permitted to inline functions from the Arm standard libraries.

`none`
The linker is not permitted to inline functions.

This option takes precedence over `--inline` if both options are present on the command line. The mapping between the options is:

- `--inline` maps to `--inline_type=all`
- `--no_inline` maps to `--inline_type=library`

Note

To disable linker inlining completely you must use `--inline_type=none`.

Related reference

11.62 --inline, --no_inline on page 11-286
11.141 --tailreorder, --no_tailreorder on page 11-373

11.64 --inlineveneer, --no_inlineveneer

Enables or disables the generation of inline veneers to give greater control over how the linker places sections.

Default

The default is `--inlineveneer`.

Related concepts

3.6.3 Veneer types on page 3-54

3.6 Linker-generated veneers on page 3-53

3.6.2 Veneer sharing on page 3-53

3.6.4 Generation of position independent to absolute veneers on page 3-55

3.6.5 Reuse of veneers when scatter-loading on page 3-55

Related reference

11.102 --piveneer, --no_piveneer on page 11-331

11.152 --veneershare, --no_veneershare on page 11-384

11.65 input-file-list

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

Usage

The linker sorts through the input file list in order. If the linker is unable to resolve input file problems then a diagnostic message is produced.

The symdefs files can be included in the list to provide global symbol addresses for previously generated image files.

You can use libraries in the input file list in the following ways:

- Specify a library to be added to the list of libraries that the linker uses to extract members if they resolve any non weak unresolved references. For example, specify `mystring.lib` in the input file list.

  Note

  Members from the libraries in this list are added to the image only when they resolve an unresolved non weak reference.

- Specify particular members to be extracted from a library and added to the image as individual objects. Members are selected from a comma separated list of patterns that can include wild characters. Spaces are permitted but if you use them you must enclose the whole input file list in quotes.

  The following shows an example of an input file list both with and without spaces:

  ```
  mystring.lib(strcmp.o,std*.o)
  ```

  ```
  “mystring.lib(strcmp.o, std*.o)”
  ```

The linker automatically searches the appropriate C and C++ libraries to select the best standard functions for your image. You can use `--no_scanlib` to prevent automatic searching of the standard system libraries.

The linker processes the input file list in the following order:

1. Objects are added to the image unconditionally.
2. Members selected from libraries using patterns are added to the image unconditionally, as if they are objects. For example, to add all `a*.o` objects and `stdio.o` from `mystring.lib` use the following:

   ```
   "mystring.lib(stdio.o, a*.o)"
   ```

3. Library files listed on the command-line are searched for any unresolved non-weak references. The standard C or C++ libraries are added to the list of libraries that the linker later uses to resolve any remaining references.

Related concepts

6.5 Access symbols in another image on page 6-97

3.9 How the linker performs library searching, selection, and scanning on page 3-60

Related reference

11.119 --scanlib, --no_scanlib on page 11-349

11.129 --stdlib on page 11-361

11.66 --keep=section_id

Specifies input sections that must not be removed by unused section elimination.

Syntax

`--keep=section_id`

Where *section_id* is one of the following:

symbol

Specifies that an input section defining *symbol* is to be retained during unused section elimination. If multiple definitions of *symbol* exist, `armlink` generates an error message.

For example, you might use `--keep=int_handler`.

To keep all sections that define a symbol ending in `_handler`, use `--keep=*_handler`.

object(*section*)

Specifies that *section* from *object* is to be retained during unused section elimination. If a single instance of *section* is generated, you can omit *section*, for example, `file.o()`. Otherwise, you must specify *section*.

For example, to keep the `vect` section from the `vectors.o` object use: `--keep=vectors.o(vect)`

To keep all sections from the `vectors.o` object where the first three characters of the name of the sections are `vec`, use: `--keep=vectors.o(vec*)`

object

Specifies that the single input section from *object* is to be retained during unused section elimination. If you use this short form and there is more than one input section in *object*, the linker generates an error message.

For example, you might use `--keep=dspdata.o`.

To keep the single input section from each of the objects that has a name starting with `dsp`, use `--keep=dsp*.o`.

Usage

All forms of the *section_id* argument can contain the `*` and `?` wild characters. Matching is case-insensitive, even on hosts with case-sensitive file naming. For example:

- `--keep foo.o(Premier*)` causes the entire match for `Premier*` to be case-insensitive.
- `--keep foo.o(Premier)` causes a case-insensitive match for the string Premier.

Note

The only case where a case-sensitive match is made is for `--keep=`*symbol* when *symbol* does not contain any wildcard characters.

Use `*.o` to match all object files. Use `*` to match all object files and libraries.

You can specify multiple `--keep` options on the command line.

Matching a symbol that has the same name as an object

If you name a symbol with the same name as an object, then `--keep=`*symbol_id* searches for a symbol that matches *symbol_id*:

- If a symbol is found, it matches the symbol.
- If no symbol is found, it matches the object.

You can force `--keep` to match an object with `--keep=symbol_id()`. Therefore, to keep both the symbol and the object, specify `--keep foo.o --keep foo.o()`.

Related concepts

3.9 How the linker performs library searching, selection, and scanning on page 3-60

3.1 The structure of an Arm® ELF image on page 3-33

11.67 --keep_intermediate

Specifies whether the linker preserves the ELF intermediate object file produced by the link time optimizer.

Syntax

`--keep_intermediate=`*`option`*

Where *`option`* is:

`lto`

Preserve an intermediate ELF object file produced by the link time optimizer.

Default

By default, `armlink` does not preserve the intermediate object file produced by the link time optimizer.

Related reference

11.79 --lto, --no_lto on page 11-306

Related information

Optimizing across modules with link time optimization

11.68 --largeregions, --no_largeregions

Controls the sorting order of sections in large execution regions to minimize the distance between sections that call each other.

Usage

If the execution region contains more code than the range of a branch instruction then the linker switches to large region mode. In this mode the linker sorts according to the approximated average call depth of each section in ascending order. The linker might also distribute veneers amongst the code sections to minimize the number of veneers.

——— **Note** ———

Large region mode can result in large changes to the layout of an image even when small changes are made to the input.

———————————

To disable large region mode and revert to lexical order, use `--no_largeregions`. Section placement is then predictable and image comparisons are more predictable. The linker automatically switches on `--veneerinject` if it is needed for a branch to reach the veneer.

Large region support enables:

- Average call depth sorting, `--sort=AvgCallDepth`.
- API sorting, `--api`.
- Veneer injection, `--veneerinject`.

The following command lines are equivalent:

```
armlink --largeregions --no_api --no_veneerinject --sort=Lexical
armlink --no_largeregions
```

Default

The default is `--no_largeregions`. The linker automatically switches to `--largeregions` if at least one execution region contains more code than the smallest inter-section branch. The smallest inter-section branch depends on the code in the region and the target processor:

128MB

Execution region contains only A64 instructions.

32MB

Execution region contains only A32 instructions.

16MB

Execution region contains T32 instructions, 32-bit T32 instructions are supported.

4MB

Execution region contains T32 instructions, no 32-bit T32 instructions are supported.

Related concepts

3.6 Linker-generated veneers on page 3-53

3.6.2 Veneer sharing on page 3-53

3.6.3 Veneer types on page 3-54

3.6.4 Generation of position independent to absolute veneers on page 3-55

Related reference

11.4 --api, --no_api on page 11-222

11.126 --sort=algorithm on page 11-357

11.150 --veneer_inject_type=type on page 11-382
11.149 --veneerinject, --no_veneerinject on page 11-381

11.69 --last=section_id

Places the selected input section last in its execution region.

Syntax

`--last=section_id`

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

`symbol`

Selects the section that defines *`symbol`*. You must not specify a symbol that has more than one definition because only a single section can be placed last. For example: `--last=checksum`.

`object``(`*`section`*`)`

Selects the *`section`* from *`object`*. There must be no space between *`object`* and the following open parenthesis. For example: `--last=checksum.o(check)`.

`object`

Selects the single input section from *`object`*. For example: `--last=checksum.o`.

If you use this short form and there is more than one input section in *`object`*, `armlink` generates an error message.

Usage

The `--last` option cannot be used with `--scatter`. Instead, use the `+LAST` attribute in a scatter file.

Example

This option can force an input section that contains a checksum to be placed last in the RW section.

Related concepts

3.3.2 Section placement with the FIRST and LAST attributes on page 3-49

3.3 Section placement with the linker on page 3-48

Related reference

11.48 --first=section_id on page 11-270

11.120 --scatter=filename on page 11-350

11.70 --legacyalign, --no_legacyalign

Controls how padding is inserted into the image.

Note

The `--legacyalign` and `--no_legacyalign` linker options are deprecated.

Usage

Using `--legacyalign`, the linker assumes execution regions and load regions to be four-byte aligned. This option enables the linker to minimize the amount of padding that it inserts into the image.

The `--no_legacyalign` option instructs the linker to insert padding to force natural alignment of execution regions. Natural alignment is the highest known alignment for that region.

Use `--no_legacyalign` to ensure strict conformance with the ELF specification.

You can also use expression evaluation in a scatter file to avoid padding.

Default

The default is `--no_legacyalign`,

Related concepts

3.3 Section placement with the linker on page 3-48

7.12 Example of using expression evaluation in a scatter file to avoid padding on page 7-151

Related reference

8.3.3 Load region attributes on page 8-169

8.4.3 Execution region attributes on page 8-175

11.71 --libpath=pathlist

Specifies a list of paths that the linker uses to search for the Arm standard C and C++ libraries.

Syntax

`--libpath=`*`pathlist`*

Where *`pathlist`* is a comma-separated list of paths that the linker only uses to search for required Arm libraries. Do not include spaces between the comma and the path name when specifying multiple path names, for example, *`path1,path2,path3,…,pathn`*.

Note

This option does not affect searches for user libraries. Use `--userlibpath` instead for user libraries.

Related concepts

3.9 How the linker performs library searching, selection, and scanning on page 3-60

Related reference

11.148 --userlibpath=pathlist on page 11-380

Related information

Toolchain environment variables

11.72 --library=name

Enables the linker to search a static library without you having specifying the full library filename on the command-line.

Note

Not supported in the Keil® Microcontroller Development Kit (Keil MDK).

Syntax

`--library=name`

Links with the static library, `libname.a`.

Usage

The order that references are resolved to libraries is the order that you specify the libraries on the command line.

Example

The following example shows how to search for `libfoo.a` before `libbar.a`:

```
--library=foo --library=bar
```

Related reference

11.51 --fpic on page 11-273

11.73 --library_security=protection

Selects one of the security hardened libraries with varying levels of protection, which include branch protection and memory tagging.

Note

This topic includes descriptions of [ALPHA] features. See *Support level definitions* on page 1-21.

Default

The default is `--library_security=auto`.

Syntax

`--library_security=`*`protection`*

Parameters

`protection` specifies the level of protection in the library.

v8.3a

Selects the `v8.3a` library, which provides branch protection using Branch Target Identification and Pointer Authentication on function returns.

v8.5a [ALPHA]

Selects the `v8.5a` library, which provides memory tagging protection of the stack used by the library code. This library also includes all the protection in the `v8.3a` library. Use of the `v8.5a` library is an [ALPHA] feature.

none

Selects the standard C library that does not provide protection using Branch Target Identification and Pointer Authentication, and does not provide memory tagging stack protection.

auto

The linker automatically selects either the standard C library, or the `v8.3a`, or the `v8.5a` library. If at least one input object file has been compiled with `-mmemtag-stack` and at least one input object file has return address signing with pointer authentication, then the linker selects the `v8.5a` library. Otherwise, if at least one input object file has been compiled for Armv8.3-A or later, and has return address signing with pointer authentication, then the linker selects the `v8.3a` library. Otherwise, the behavior is the same as `--library_security=none`.

Note

- The presence of BTI instructions in the compiled objects does not affect automatic library selection.
- The presence of memory tagging instructions in the compiled objects does not affect automatic library selection.

Usage

Use `--library_security` to override the automatic selection of protected libraries for branch protection and memory tagging stack protection (stack tagging).

Branch protection protects your code from Return Oriented Programming (ROP) and Jump Oriented Programming (JOP) attacks. Branch protection using pointer authentication and branch target identification are only available in AArch64 state.

Memory tagging stack protection protects accesses to variables on the stack whose addresses are taken. Memory tagging protection is available for the AArch64 state for architectures with the memory tagging extension.

Note

- Selecting the `v8.5a` library does not automatically imply memory tagging protection of the heap. To enable memory tagging protection of the heap, you must define the symbol `__use_memtag_heap`. You can define this symbol irrespective of the level of *`protection`* you use for `--library_security=`*`protection`*. For more information, see *Choosing a heap implementation for memory allocation functions*.
- Code compiled with stack tagging can be safely linked together with code compiled without stack tagging. However, if any object file is compiled with `-mmemtag-stack`, and if `setjmp`, `longjmp`, or C ++ exceptions are present anywhere in the image, then you must use the `v8.5a` library to avoid stack tagging related memory fault at runtime.

Examples

This uses the `v8.3a` library with branch protection using Branch Target Identification and Pointer Authentication:

```
armlink --cpu=8.3-A.64 --library_security=v8.3a foo.o
```

This uses the standard C library without any branch protection using Branch Target Identification and Pointer Authentication:

```
armlink --cpu=8.3-A.64 --library_security=none foo.o
```

This uses the `v8.5a` library with memory tagging stack protection, and branch protection using Branch Target Identification and Pointer Authentication:

```
armlink --library_security=v8.5a foo.o
```

Related reference

11.74 --library_type=lib on page 11-301

Related information

-mbranch-protection

11.74 --library_type=lib

Selects the library to be used at link time.

Syntax

`--library_type=lib`

Where `lib` can be one of:

`standardlib`
Specifies that the full Arm Compiler runtime libraries are selected at link time. This is the default.

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

——— **Note** ———

`microlib` is not supported for AArch64 state.

———————————

Usage

Use this option when use of the libraries require more specialized optimizations.

Default

If you do not specify `--library_type` at link time and no object file specifies a preference, then the linker assumes `--library_type=standardlib`.

Related information

Building an application with microlib

11.75 --list=filename

Redirects diagnostic output to a file.

Syntax

`--list=`*`filename`*

Where *`filename`* is the file to use to save the diagnostic output. *`filename`* can include a path

Usage

Redirects the diagnostics output by the `--info`, `--map`, `--symbols`, `--verbose`, `--xref`, `--xreffrom`, and `--xrefto` options to *`file`*.

The specified file is created when diagnostics are output. If a file of the same name already exists, it is overwritten. However, if diagnostics are not output, a file is not created. In this case, the contents of any existing file with the same name remain unchanged.

If *`filename`* is specified without a path, it is created in the output directory, that is, the directory where the output image is being written.

Related reference

11.85 --map, --no_map on page 11-314
11.153 --verbose on page 11-385
11.158 --xref, --no_xref on page 11-390
11.159 --xrefdbg, --no_xrefdbg on page 11-391
11.160 --xref{from|to}=object(section) on page 11-392
11.59 --info=topic[,topic,...] on page 11-281
11.137 --symbols, --no_symbols on page 11-369

11.76 --list_mapping_symbols, --no_list_mapping_symbols

Enables or disables the addition of mapping symbols in the output produced by `--symbols`.

The mapping symbols `$a`, `$t`, `$t.x`, `$d`, and `$x` flag transitions between A32 code, T32 code, ThumbEE code (Armv7-A), data, and A64 code.

Default

The default is `--no_list_mapping_symbols`.

Related concepts

6.1 About mapping symbols on page 6-88

Related reference

11.137 --symbols, --no_symbols on page 11-369

Related information

ELF for the Arm Architecture

11.77 --load_addr_map_info, --no_load_addr_map_info

Includes the load addresses for execution regions and the input sections within them in the map file.

Usage

If an input section is compressed, then the load address has no meaning and `COMPRESSED` is displayed instead.

For sections that do not have a load address, such as ZI data, the load address is blank

Default

The default is `--no_load_addr_map_info`.

Restrictions

You must use `--map` with this option.

Example

The following example shows the format of the map file output:

```
    Base Addr    Load Addr    Size         Type   Attr      Idx    E Section Name
Object
    0x00008000   0x00008000   0x00000008   Code   RO           25  * !!!main
__main.o(c_4.l)
    0x00010000   COMPRESSED   0x00001000   Data   RW            2    dataA
data.o
    0x00003000         -      0x00000004   Zero   RW            2    .bss
test.o
```

Related reference

11.85 --map, --no_map on page 11-314

11.78 --locals, --no_locals

Adds local symbols or removes local symbols depending on whether an image or partial object is being output.

Usage

The `--locals` option adds local symbols in the output symbol table.

The effect of the `--no_locals` option is different for images and object files.

When producing an executable image `--no_locals` removes local symbols from the output symbol table.

For object files built with the `--partial` option, the `--no_locals` option:

- Keeps mapping symbols and build attributes in the symbol table.
- Removes those local symbols that can be removed without loss of functionality.

  Symbols that cannot be removed, such as the targets for relocations, are kept. For these symbols, the names are removed. These are marked as `[Anonymous Symbol]` in the `fromelf --text` output.

`--no_locals` is a useful optimization if you want to reduce the size of the output symbol table in the final image.

Default

The default is `--locals`.

Related reference

11.108 --privacy on page 11-338

Related information

--privacy fromelf option

--strip=option[,option,...] fromelf option

11.79 --lto, --no_lto

Enables link time optimization.

Caution

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

Note

When you specify the `-flto` option, `armclang` produces ELF files that contain bitcode in a `.llvmbc` section.

With the `--no_lto` option, `armlink` gives an error message if it encounters any `.llvmbc` sections.

Default

The default is `--no_lto`.

Dependencies

Link time optimization requires the dependent library `libLTO`.

Table 11-4 Link time optimization dependencies

Dependency	Windows filename	Linux filename
`libLTO`	`LTO.dll`	`libLTO.so`

By default, the dependent library `libLTO` is present in the same directory as `armlink`.

The search order for these dependencies is as follows.

`LTO.dll`:

1. The same directory as the `armlink` executable.
2. The directories in the current directory search path.

`libLTO.so`:

1. The same directory as the `armlink` executable.
2. The directories in the `LD_LIBRARY_PATH` environment variable.
3. The cache file `/etc/ld.so.cache`.
4. The directories `/lib` and `/usr/lib`.

   These directories might have the suffix `64` on some 64-bit Linux systems. For example, on 64-bit Red Hat Enterprise Linux the directories are `/lib64` and `/usr/lib64`.

Note

The `armclang` executables and the `libLTO` library must come from the same Arm Compiler 6 installation. Any use of `libLTO` other than that supplied with Arm Compiler 6 is unsupported.

Note

Link Time Optimization does not honor the `armclang -mexecute-only` option. If you use the `armclang -flto` or `-Omax` options, then the compiler cannot generate execute-only code and produces a warning.

Related reference

11.59 --info=topic[,topic,...] on page 11-281

11.67 --keep_intermediate on page 11-292

11.80 --lto_keep_all_symbols, --no_lto_keep_all_symbols on page 11-308

11.81 --lto_intermediate_filename on page 11-309

11.83 --lto_relocation_model on page 11-312

11.82 --lto_level on page 11-310

11.96 -Omax on page 11-325

Related information

-flto

Optimizing across modules with link time optimization

11.80 --lto_keep_all_symbols, --no_lto_keep_all_symbols

Specifies whether link time optimization removes unreferenced global symbols.

Using `--lto_keep_all_symbols` affects all symbols and largely reduces the usefulness of link time optimization. If you need to keep only a specific unreferenced symbol, then use the `--keep` option instead.

Default

The default is `--no_lto_keep_all_symbols`.

Related reference

11.66 --keep=section_id on page 11-290

11.79 --lto, --no_lto on page 11-306

Related information

Optimizing across modules with link time optimization

11.81 --lto_intermediate_filename

Specifies the name of the ELF object file produced by the link time optimizer.

Syntax

`--lto_intermediate_filename=`*`filename`*

Where *`filename`* is the filename the link time optimizer uses for the ELF object file it produces.

Usage

The purpose of the `--lto_intermediate_filename` option is so that the intermediate file produced by the link time optimizer can be named in other inputs to the linker, such as scatter loading files.

Note

The `--lto_intermediate_filename` option does not cause the linker to keep the intermediate object file. Use the `--keep-intermediate=lto` option to keep the intermediate file.

Default

The default is a temporary filename.

Related reference

11.67 --keep_intermediate on page 11-292
11.79 --lto, --no_lto on page 11-306

Related information

Optimizing across modules with link time optimization

11.82 --lto_level

Sets the optimization level for the link time optimization feature.

Syntax

`--lto_level=O`*`level`*

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

`0`

Minimum optimization for the performance of the compiled binary. Turns off most optimizations. When debugging is enabled, this option generates code that directly corresponds to the source code. Therefore, this optimization might result in a significantly larger image.

`1`

Restricted optimization. When debugging is enabled, this option selects a good compromise between image size, performance, and quality of debug view.

Arm recommends `-O1` rather than `-O0` for the best trade-off between debug view, code size, and performance.

`2`

High optimization. When debugging is enabled, the debug view might be less satisfactory because the mapping of object code to source code is not always clear. The linker might perform optimizations that the debug information cannot describe.

This optimization is the default optimization level.

`3`

Very high optimization. When debugging is enabled, this option typically gives a poor debug view. Arm recommends debugging at lower optimization levels.

`fast`

Enables all the optimizations from level `3` including those optimizations that are performed with the `-ffp-mode=fast armclang` option. This level also performs other aggressive optimizations that might violate strict compliance with language standards.

`max`

Maximum optimization. Specifically targets performance optimization. Enables all the optimizations from level `fast`, together with other aggressive optimizations.

——— **Caution** ———

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

———————————

——— **Note** ———

- Code-size, build-time, and the debug view can each be adversely affected when using this option.
- Arm cannot guarantee that the best performance optimization is achieved in all code cases.

———————————

`s`

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

`z`

Performs optimizations to minimize image size.

Default

If you do not specify `O`*`level`*, the linker assumes `O2`.

Related reference

11.79 --lto, --no_lto on page 11-306

11.96 -Omax on page 11-325

Related information

-O

Optimizing across modules with link time optimization

11.83 --lto_relocation_model

Specifies whether the link time optimizer produces absolute or position independent code.

Syntax

`--lto_relocation_model=`*`model`*

Where model is one of the following:

`static`
The link time optimizer produces absolute code.

`pic`
The link time optimizer produces code that uses GOT relative position independent code.

The `--lto_relocation_model=pic` option requires the `armlink --bare_metal_pie` option.

Note

The Bare-metal PIE feature is deprecated.

Default

The default is `--lto_relocation_model=static`.

Related reference
11.6 --bare_metal_pie on page 11-224
11.79 --lto, --no_lto on page 11-306
Related information
Optimizing across modules with link time optimization

11.84 --mangled, --unmangled

Instructs the linker to display mangled or unmangled C++ symbol names in diagnostic messages, and in listings produced by the `--xref`, `--xreffrom`, `--xrefto`, and `--symbols` options.

Usage

If `--unmangled` is selected, C++ symbol names are displayed as they appear in your source code.

If `--mangled` is selected, C++ symbol names are displayed as they appear in the object symbol tables.

Default

The default is `--unmangled`.

Related reference

11.137 --symbols, --no_symbols on page 11-369
11.158 --xref, --no_xref on page 11-390
11.159 --xrefdbg, --no_xrefdbg on page 11-391
11.160 --xref{from|to}=object(section) on page 11-392

11.85 --map, --no_map

Enables or disables the printing of a memory map.

Usage

The map contains the address and the size of each load region, execution region, and input section in the image, including linker-generated input sections. This can be output to a text file using `--list=filename`.

Default

The default is `--no_map`.

Related reference

11.77 --load_addr_map_info, --no_load_addr_map_info on page 11-304
11.75 --list=filename on page 11-302
11.121 --section_index_display=type on page 11-352

11.86 --max_er_extension=size

Specifies a constant value to add to the size of an execution region when no maximum size is specified for that region. The value is used only when placing `__at` sections.

Syntax

`--max_er_extension=`*`size`*

Where *`size`* is the constant value in bytes to use when calculating the size of the execution region.

Default

The default size is `10240` bytes.

Related tasks

7.2.7 Automatically placing __at sections on page 7-121

11.87 --max_veneer_passes=value

Specifies a limit to the number of veneer generation passes the linker attempts to make when certain conditions are met.

Syntax

`--max_veneer_passes=`*`value`*

Where *`value`* is the maximum number of veneer passes the linker is to attempt. The minimum value you can specify is one.

Usage

The linker applies this limit when both the following conditions are met:

- A section that is sufficiently large has a relocation that requires a veneer.
- The linker cannot place the veneer close enough to the call site.

The linker attempts to diagnose the failure if the maximum number of veneer generation passes you specify is exceeded, and displays a warning message. You can downgrade this warning message using `--diag_remark`.

Default

The default number of passes is 10.

Related reference

11.28 --diag_remark=tag[,tag,...] on page 11-250
11.31 --diag_warning=tag[,tag,...] on page 11-253

11.88 --max_visibility=type

Controls the visibility of all symbol definitions.

Syntax

`--max_visibility=`*`type`*

Where *`type`* can be one of:

`default`

Default visibility.

`protected`

Protected visibility.

Usage

Use `--max_visibility=protected` to limit the visibility of all symbol definitions. Global symbol definitions that normally have default visibility, are given protected visibility when this option is specified.

Default

The default is `--max_visibility=default`.

Related reference

11.95 --override_visibility on page 11-324

11.89 --merge, --no_merge

Enables or disables the merging of **const** strings that are placed in shareable sections by the compiler.

Usage

Using `--merge` can reduce the size of the image if there are similarities between **const** strings.

Use `--info=merge` to see a listing of the merged **const** strings.

By default, merging happens between different load and execution regions. Therefore, code from one execution or load region might use a string stored in different region. If you do not want this behavior, then do one of the following:

- Use the `PROTECTED` load region attribute if you are using scatter-loading.
- Globally disable merging with `--no_merge`.

Default

The default is `--merge`.

Related reference

11.59 --info=topic[,topic,...] on page 11-281
8.3.3 Load region attributes on page 8-169

11.90 --merge_litpools, --no_merge_litpools

Attempts to merge identical constants in objects targeted at AArch32 state. The objects must be produced with Arm Compiler 6.

Default

`--merge_litpools` is the default.

Related tasks

4.10 Merging identical constants on page 4-78

11.91 --muldefweak, --no_muldefweak

Enables or disables multiple weak definitions of a symbol.

Usage

If enabled, the linker chooses the first definition that it encounters and discards all the other duplicate definitions. If disabled, the linker generates an error message for all multiply defined weak symbols.

Default

The default is `--muldefweak`.

11.92 -o filename, --output=filename

Specifies the name of the output file. The file can be either a partially-linked object or an executable image, depending on the command-line options used.

Syntax

`--output=`*`filename`*

`-o` *`filename`*

Where *`filename`* is the name of the output file, and can include a path.

Usage

If `--output=`*`filename`* is not specified, the linker uses the following default filenames:

`__image.axf`

If the output is an executable image.

`__object.o`

If the output is a partially-linked object.

If *`filename`* is specified without path information, it is created in the current working directory. If path information is specified, then that directory becomes the default output directory.

Related reference

11.14 --callgraph_file=filename on page 11-234
11.100 --partial on page 11-329

11.93 --output_float_abi=option

Specifies the floating-point procedure call standard to advertise in the ELF header of the executable.

Note

Not supported for AArch64 state.

Syntax

`--output_float_abi=option`

where *option* is one of the following:

auto
Checks the object files to determine whether the hard float or soft float bit in the ELF header flag is set.

hard
The executable file is built to conform to the hardware floating-point procedure-call standard.

soft
Conforms to the software floating-point procedure-call standard.

Usage

When the option is set to `auto`:

- For multiple object files:
  - If all the object files specify the same value for the flag, then the executable conforms to the relevant standard.
  - If some files have the hard float and soft float bits in the ELF header flag set to different values from other files, this option is ignored and the hard float and soft float bits in the executable are unspecified.
- If a file has the build attribute `Tag_ABI_VFP_args` set to `2`, then the hard float and soft float bits in the ELF header flag in the executable are set to zero.
- If a file has the build attribute `Tag_ABI_VFP_args` set to `3`, then `armlink` ignores this option.

You can use `fromelf --text` on the image to see whether hard or soft float is set in the ELF header flag.

Default

The default option is `auto`.

Related information

--decode_build_attributes
--text
ELF for the Arm Architecture
Run-time ABI for the Arm Architecture

11.94 --overlay_veneers

When using the automatic overlay mechanism, causes `armlink` to redirect calls between overlays to a veneer. The veneer allows an overlay manager to unload and load the correct overlays.

Note

You must use this option if your scatter file includes execution regions with `AUTO_OVERLAY` attribute assigned to them.

Usage

`armlink` creates a veneer for a function call when any of the following are true:

- The calling function is in non-overlaid code and the called function is in an overlay.
- The calling function is in an overlay and the called function is in a different overlay.
- The calling function is in an overlay and the called function is in non-overlaid code.

In the last of these cases, an overlay does not have to be loaded immediately, but the overlay manager typically has to adjust the return address. It does this adjustment so that it can arrange to check on function return that the overlay of the caller is reloaded before returning to it.

Veneers are not created when calls between two functions are in the same overlay. If the calling function is running, then the called function is guaranteed to be loaded already, because each overlay is atomic. This situation is also guaranteed when the called function returns.

A relocation might refer to a function in an overlay and not modify a branch instruction. For example, the relocations R_ARM_ABS32 or R_ARM_REL32 do not modify a branch instruction. In this situation, `armlink` redirects the relocation to point at a veneer for the function regardless of where the relocation is. This redirection is done in case the address of the function is passed into another overlay as an argument.

Related reference

8.4.3 Execution region attributes on page 8-175

Related information

Automatic overlay support

11.95 --override_visibility

Enables `EXPORT` and `IMPORT` directives in a steering file to override the visibility of a symbol.

Usage

By default:

- Only symbol definitions with `STV_DEFAULT` or `STV_PROTECTED` visibility can be exported.
- Only symbol references with `STV_DEFAULT` visibility can be imported.

When you specify `--override_visibility`, any global symbol definition can be exported and any global symbol reference can be imported.

Related reference

11.145 --undefined_and_export=symbol on page 11-377

12.1 EXPORT steering file command on page 12-395

12.3 IMPORT steering file command on page 12-397

11.96 -Omax

Enables maximum link time optimization.

`-Omax` automatically enables the `--lto` and `--lto_level=Omax` options.

If you have object files that have been compiled with the `armclang -Omax` option, then you can link them using the `armlink -Omax` option to produce an image with maximum link time optimization.

Related reference

11.82 --lto_level on page 11-310
11.79 --lto, --no_lto on page 11-306

Related information

-O
Optimizing across modules with link time optimization

11.97 --pad=num

Enables you to set a value for padding bytes. The linker assigns this value to all padding bytes inserted in load or execution regions.

Syntax

`--pad=`*num*

Where *num* is an integer, which can be given in hexadecimal format.

For example, setting *num* to `0xFF` might help to speed up ROM programming time. If *num* is greater than `0xFF`, then the padding byte is cast to a `char`, that is `(char)`*num*.

Usage

Padding is only inserted:

- Within load regions. No padding is present between load regions.
- Between fixed execution regions (in addition to forcing alignment). Padding is not inserted up to the maximum length of a load region unless it has a fixed execution region at the top.
- Between sections to ensure that they conform to alignment constraints.

Related concepts

3.1.2 Input sections, output sections, regions, and program segments on page 3-34

3.1.3 Load view and execution view of an image on page 3-35

11.98 --paged

Enables Demand Paging mode to help produce ELF files that can be demand paged efficiently.

Usage

A default page size of `0x8000` bytes is used. You can change this with the `--pagesize` command-line option.

Default

Related concepts

3.4 Linker support for creating demand-paged files on page 3-51

Related tasks

7.9 Aligning regions to page boundaries on page 7-147

Related reference

11.99 --pagesize=pagesize on page 11-328

11.99 --pagesize=pagesize

Allows you to change the page size used when demand paging.

Syntax

`--pagesize=`*`pagesize`*

Where *`pagesize`* is the page size in bytes.

Default

The default value is `0x8000`.

Related concepts

3.4 Linker support for creating demand-paged files on page 3-51

Related tasks

7.9 Aligning regions to page boundaries on page 7-147

Related reference

11.98 --paged on page 11-327

11.100 --partial

Creates a partially-linked object that can be used in a subsequent link step.

Restrictions

You cannot use `--partial` with `--scatter`.

Related concepts

2.3 Partial linking model on page 2-28

11.101 --pie

Species the *Position Independent Executable* (PIE) linking model.

Note

The Bare-metal PIE feature is deprecated.

Note

You must use this option with the `--fpic` and `--ref_pre_init` options.

Related reference

11.51 --fpic on page 11-273
11.6 --bare_metal_pie on page 11-224
11.110 --ref_pre_init, --no_ref_pre_init on page 11-340

11.102 --piveneer, --no_piveneer

Enables or disables the generation of a veneer for a call from *position independent* (PI) code to absolute code.

Usage

When using `--no_piveneer`, an error message is produced if the linker detects a call from PI code to absolute code.

Note

Not supported for AArch64 state.

Default

The default is `--piveneer`.

Related concepts

3.6.4 Generation of position independent to absolute veneers on page 3-55

3.6 Linker-generated veneers on page 3-53

3.6.2 Veneer sharing on page 3-53

3.6.3 Veneer types on page 3-54

3.6.5 Reuse of veneers when scatter-loading on page 3-55

Related reference

11.64 --inlineveneer, --no_inlineveneer on page 11-288

11.152 --veneershare, --no_veneershare on page 11-384

11.103 --pixolib

Generates a Position Independent eXecute Only (PIXO) library.

Default

`--pixolib` is disabled by default.

Syntax

`--pixolib`

Parameters

None.

Usage

Use `--pixolib` to create a PIXO library, which is a relocatable library containing eXecutable Only code.

When creating the PIXO library, if you use `armclang` to invoke the linker, then `armclang` automatically passes the linker option `--pixolib` to `armlink`. If you invoke the linker separately, then you must use the `armlink --pixolib` command-line option. When creating a PIXO library, you must also provide a scatter file to the linker.

Each PIXO library must contain all the required standard library functions. Arm Compiler 6 provides PIXO variants of the standard libraries based on Microlib. You must specify the required libraries on the command-line when creating your PIXO library. These libraries are location in the compiler installation directory under `/lib/pixolib/`.

The PIXO variants of the standard libraries have the naming format `<base>.<endian>`:

- `<base>`

  mc_wg
  : C library.

  m_wgv
  : Math library for targets with hardware double precision floating-point support that is compatible with `vfpv5-d16`.

  m_wgm
  : Math library for targets with hardware single precision floating-point support that is compatible with `fpv4-sp-d16`.

  m_wgs
  : Math library for targets without hardware support for floating-point.

  mf_wg
  : Software floating-point library. This library is required when:
    - Using `printf()` to print floating-point values.
    - Using a math library that does not have all the required floating-point support in hardware. For example if your code has double precision floating-point operations but your target has `fpv4-sp-d16`, then the software floating-point library is used for the double-precision operations.

- `<endian>`

  l
  : Little endian

b

Big endian

Restrictions

Note

Generation of PIXO libraries is only supported for Armv7-M targets.

When linking your application code with your PIXO library:

- The linker must not remove any unused sections from the PIXO library. You can ensure this with the armlink `--keep` command-line option.
- The `RW` sections with `SHT_NOBITS` and `SHT_PROGBITS` must be kept in the same order and relative offset for each PIXO library in the final image, as they were in the original PIXO libraries before linking the final image.

Examples

This example shows the command-line invocations for compiling and linking in separate steps, to create a PIXO library from the source file `foo.c`.

```
armclang --target=arm-arm-none-eabi -march=armv7-m -mpixolib -c -o foo.o foo.c
armlink --pixolib --scatter=pixo.scf -o foo-pixo-library.o foo.o mc_wg.l
```

This example shows the command-line invocations for compiling and linking in a single step, to create a PIXO library from the source file `foo.c`.

```
armclang --target=arm-arm-none-eabi -march=armv7-m -mpixolib -Wl,--scatter=pixo.scf -o foo-
pixo-library.o foo.c mc_wg.l
```

Related reference

11.66 --keep=section_id on page 11-290

11.128 --startup=symbol, --no_startup on page 11-360

Related information

-mpixolib

The Arm C Micro-Library

11.104 --pltgot=type

Specifies the type of *Procedure Linkage Table* (PLT) and *Global Offset Table* (GOT) to use, corresponding to the different addressing modes of the *Base Platform Application Binary Interface* (BPABI).

Note

This option is supported only when using `--base_platform` or `--bpabi`.

Note

Not supported for AArch64 state.

Syntax

`--pltgot=type`

Where *type* is one of the following:

`none`

References to imported symbols are added as dynamic relocations for processing by a platform specific post-linker.

`direct`

References to imported symbols are resolved to read-only pointers to the imported symbols. These are direct pointer references.

Use this type to turn on PLT generation when using `--base_platform`.

`indirect`

The linker creates a GOT and possibly a PLT entry for the imported symbol. The reference refers to PLT or GOT entry.

This type is not supported if you have multiple load regions.

`sbrel`

Same referencing as `indirect`, except that GOT entries are stored as offsets from the static base address for the segment held in R9 at runtime.

This type is not supported if you have multiple load regions.

Default

When the `--bpabi` or `--dll` options are used, the default is `--pltgot=direct`.

When the `--base_platform` option is used, the default is `--pltgot=none`.

Related concepts

2.5 Base Platform linking model on page 2-30
2.4 Base Platform Application Binary Interface (BPABI) linking model on page 2-29

Related reference

11.7 --base_platform on page 11-225
11.11 --bpabi on page 11-230
11.105 --pltgot_opts=mode on page 11-335
11.32 --dll on page 11-254

11.105 --pltgot_opts=mode

Controls the generation of *Procedure Linkage Table* (PLT) entries for weak references and function calls to relocatable targets within the same file.

Note

Not supported for AArch64 state.

Syntax

`--pltgot_opts=mode[,mode,...]`

Where *mode* is one of the following:

crosslr
Calls to and from a load region marked `RELOC` go by way of the PLT.

nocrosslr
Calls to and from a load region marked `RELOC` do not generate PLT entries.

noweakrefs
Generates a `NOP` for a function call, or zero for data. No PLT entry is generated. Weak references to imported symbols remain unresolved.

weakrefs
Weak references produce a PLT entry. These references must be resolved at a later link stage.

Default

The default is `--pltgot_opts=nocrosslr,noweakrefs`.

Related reference

11.7 --base_platform on page 11-225
11.104 --pltgot=type on page 11-334

11.106 --predefine="string"

Enables commands to be passed to the preprocessor when preprocessing a scatter file.

You specify a preprocessor on the first line of the scatter file.

Syntax

`--predefine="string"`

You can use more than one `--predefine` option on the command-line.

You can also use the synonym `--pd="string"`.

Restrictions

Use this option with `--scatter`.

Example scatter file before preprocessing

The following example shows the scatter file contents before preprocessing.

```
#! armclang -E
lr1 BASE
{
    er1 BASE
    {
        *(+RO)
    }
    er2 BASE2
    {
        *(+RW+ZI)
    }
}
```

Use `armlink` with the command-line options:

```
--predefine="-DBASE=0x8000" --predefine="-DBASE2=0x1000000" --scatter=filename
```

This passes the command-line options: `-DBASE=0x8000 -DBASE2=0x1000000` to the compiler to preprocess the scatter file.

Example scatter file after preprocessing

The following example shows how the scatter file looks after preprocessing:

```
lr1 0x8000
{
    er1 0x8000
    {
        *(+RO)
    }
    er2 0x1000000
    {
        *(+RW+ZI)
    }
}
```

Related reference

7.11 Preprocessing a scatter file on page 7-149
11.120 --scatter=filename on page 11-350

11.107 --preinit, --no_preinit

Enables the linker to use a different image pre-initialization routine if required.

Syntax

`--preinit=symbol`

If `--preinit=symbol` is not specified then the default symbol `__arm_preinit_` is assumed.

`--no_preinit` does not take a symbol argument.

Effect

The linker adds a non-weak reference to symbol if a `.preinit_array` section is detected.

For `--preinit=__arm_preinit_` or `--cppinit=__cpp_initialize_aeabi_`, the linker processes R_ARM_TARGET1 relocations as R_ARM_REL32, because this is required by the `__arm_preinit` and `__cpp_initialize_aeabi_` functions. In all other cases R_ARM_TARGET1 relocations are processes as R_ARM_ABS32.

Related reference

11.51 --fpic on page 11-273
11.110 --ref_pre_init, --no_ref_pre_init on page 11-340
11.6 --bare_metal_pie on page 11-224

11.108 --privacy

Modifies parts of an image to help protect your code.

Usage

The effect of this option is different for images and object files.

When producing an executable image it removes local symbols from the output symbol table.

For object files built with the `--partial` option, this option:

- Changes section names to a default value, for example, changes code section names to `.text`.
- Keeps mapping symbols and build attributes in the symbol table.
- Removes those local symbols that can be removed without loss of functionality.

  Symbols that cannot be removed, such as the targets for relocations, are kept. For these symbols, the names are removed. These are marked as `[Anonymous Symbol]` in the `fromelf --text` output.

Note

To help protect your code in images and objects that are delivered to third parties, use the `fromelf --privacy` command.

Related reference

11.78 --locals, --no_locals on page 11-305

11.100 --partial on page 11-329

Related information

--privacy fromelf option

--strip=option[,option,...] fromelf option

Options to protect code in object files with fromelf

11.109 --ref_cpp_init, --no_ref_cpp_init

Enables or disables the adding of a reference to the C++ static object initialization routine in the Arm libraries.

Usage

The default reference added is `__cpp_initialize__aeabi_`. To change this you can use `--cppinit`.

Use `--no_ref_cpp_init` if you are not going to use the Arm libraries.

Default

The default is `--ref_cpp_init`.

Related reference

11.21 --cppinit, --no_cppinit on page 11-241

Related information

C++ initialization, construction and destruction

11.110 --ref_pre_init, --no_ref_pre_init

Allows the linker to add or not add references to the image pre-initialization routine in the Arm libraries. The default reference added is `__arm_preinit_`. To change this you can use `--preinit`.

Default

The default is `--no_ref_pre_init`.

Related reference

11.51 --fpic on page 11-273

11.107 --preinit, --no_preinit on page 11-337

11.6 --bare_metal_pie on page 11-224

11.111 --reloc

Creates a single relocatable load region with contiguous execution regions.

Note

This option is deprecated. Use the *BPABI* on page 9-194 or the *Base Platform linking model* on page 10-208.

Note

Not supported for AArch64 state.

Usage

Only use this option for legacy systems with the type of relocatable ELF images that conform to the *ELF for the Arm® Architecture* specification. The generated image might not be compliant with the ELF for the Arm Architecture specification.

When relocated `MOVT` and `MOVW` instructions are encountered in an image being linked with `--reloc`, `armlink` produces the following additional dynamic tags:

DT_RELA

The address of a relocation table.

DT_RELASZ

The total size, in bytes, of the DT_RELA relocation table.

DT_RELAENT

The size, in bytes, of the DT_RELA relocation entry.

Restrictions

You cannot use `--reloc` with `--scatter`.

You cannot use this option with `--xo_base`.

Related concepts

7.13.2 Type 1 image, one load region and contiguous execution regions on page 7-152

3.2.4 Type 3 image structure, multiple load regions and non-contiguous execution regions on page 3-45

Related information

Base Platform ABI for the Arm Architecture

ELF for the Arm Architecture

11.112 --remarks

Enables the display of remark messages, including any messages redesignated to remark severity using `--diag_remark`.

Note

The linker does not issue remarks by default.

Related reference

11.28 --diag_remark=tag[,tag,...] on page 11-250

11.42 --errors=filename on page 11-264

11.113 --remove, --no_remove

Enables or disables the removal of unused input sections from the image.

Usage

An input section is considered used if it contains an entry point, or if it is referred to from a used section.

By default, unused section elimination is disabled when building *dynamically linked libraries* (DLLs) or shared objects. Use `--remove` to enable unused section elimination.

Use `--no_remove` when debugging to retain all input sections in the final image even if they are unused.

Use `--remove` with the `--keep` option to retain specific sections in a normal build.

Default

The default is `--remove`.

The default is `--no_remove` only if you specify `--base_platform` or `--bpabi` with `--dll`.

Related concepts

4.2 Elimination of unused sections on page 4-67

3.9 How the linker performs library searching, selection, and scanning on page 3-60

4.1 Elimination of common section groups on page 4-66

Related reference

11.7 --base_platform on page 11-225

11.11 --bpabi on page 11-230

11.32 --dll on page 11-254

11.66 --keep=section_id on page 11-290

11.114 --ro_base=address

Sets both the load and execution addresses of the region containing the RO output section at a specified address.

Syntax

`--ro_base=address`

Where *address* must be word-aligned.

Usage

If *execute-only* (XO) sections are present, and you specify `--ro_base` without `--xo_base`, then an ER_XO execution region is created at the address specified by `--ro_base`. The ER_RO execution region immediately follows the ER_XO region.

Default

If this option is not specified, and no scatter file is specified, the default is `--ro_base=0x8000`. If XO sections are present, then this is the default value used to place the ER_XO region.

Restrictions

You cannot use `--ro_base` with:

- `--scatter`.

Related reference

11.115 --ropi on page 11-345
11.116 --rosplit on page 11-346
11.117 --rw_base=address on page 11-347
11.157 --xo_base=address on page 11-389
11.161 --zi_base=address on page 11-393
11.120 --scatter=filename on page 11-350

11.115 --ropi

Makes the load and execution region containing the RO output section position-independent.

Note

Not supported for AArch64 state.

Usage

If this option is not used, the region is marked as absolute. Usually each read-only input section must be *Read-Only Position-Independent* (ROPI). If this option is selected, the linker:

- Checks that relocations between sections are valid.
- Ensures that any code generated by the linker itself, such as interworking veneers, is ROPI.

Note

The linker gives a downgradable error if `--ropi` is used without `--rwpi` or `--rw_base`.

Restrictions

You cannot use `--ropi`:

- With `--fpic`, `--scatter`, or `--xo_base`.
- When an object file contains execute-only sections.

Related reference

11.114 --ro_base=address on page 11-344
11.116 --rosplit on page 11-346
11.117 --rw_base=address on page 11-347
11.157 --xo_base=address on page 11-389
11.161 --zi_base=address on page 11-393
11.120 --scatter=filename on page 11-350

11.116 --rosplit

Splits the default RO load region into two RO output sections.

The RO load region is split into the RO output sections:

- `RO-CODE`.
- `RO-DATA`.

Restrictions

You cannot use `--rosplit` with:

- `--scatter`.

Related reference

11.114 --ro_base=address on page 11-344
11.115 --ropi on page 11-345
11.117 --rw_base=address on page 11-347
11.157 --xo_base=address on page 11-389
11.161 --zi_base=address on page 11-393
11.120 --scatter=filename on page 11-350

11.117 --rw_base=address

Sets the execution addresses of the region containing the RW output section at a specified address.

Syntax

`--rw_base=`*`address`*
Where *`address`* must be word-aligned.

Note

This option does not affect the placement of execute-only sections.

Restrictions

You cannot use `--rw_base` with:

- `--scatter`.

Related reference

11.114 --ro_base=address on page 11-344
11.115 --ropi on page 11-345
11.116 --rosplit on page 11-346
11.157 --xo_base=address on page 11-389
11.161 --zi_base=address on page 11-393
11.120 --scatter=filename on page 11-350

11.118 --rwpi

Makes the load and execution region containing the RW and ZI output section position-independent.

Note

Not supported for AArch64 state.

Usage

If this option is not used the region is marked as absolute. This option requires a value for `--rw_base`. If `--rw_base` is not specified, `--rw_base=0` is assumed. Usually each writable input section must be *Read-Write Position-Independent* (RWPI).

If this option is selected, the linker:

- Checks that the `PI` attribute is set on input sections to any read-write execution regions.
- Checks that relocations between sections are valid.
- Generates entries relative to the static base in the table `Region$$Table`.

  This is used when regions are copied, decompressed, or initialized.

Restrictions

You cannot use `--rwpi`:

- With `--fpic` `--scatter`, or `--xo_base`.
- When an object file contains execute-only sections.

Related reference

11.127 --split on page 11-359

11.120 --scatter=filename on page 11-350

11.119 --scanlib, --no_scanlib

Enables or disables scanning of the Arm libraries to resolve references.

Use `--no_scanlib` if you want to link your own libraries.

Default

The default is `--scanlib`.

Related reference

11.129 --stdlib on page 11-361

11.120 --scatter=filename

Creates an image memory map using the scatter-loading description that is contained in the specified file.

The description provides grouping and placement details of the various regions and sections in the image.

Syntax

`--scatter=`*`filename`*

Where *`filename`* is the name of a scatter file.

Usage

To modify the placement of any unassigned input sections when `.ANY` selectors are present, use the following command-line options with `--scatter`:

- `--any_contingency`.
- `--any_placement`.
- `--any_sort_order`.

You cannot use the `--scatter` option with:

- `--bpabi`.
- `--first`.
- `--last`.
- `--partial`.
- `--reloc`.
- `--ro_base`.
- `--ropi`.
- `--rosplit`.
- `--rw_base`.
- `--rwpi`.
- `--split`.
- `--xo_base`.
- `--zi_base`.

You can use `--dll` when specified with `--base_platform`.

Related concepts

7.4.5 Examples of using placement algorithms for .ANY sections on page 7-131

7.4.8 Behavior when .ANY sections overflow because of linker-generated content on page 7-136

Related reference

11.1 --any_contingency on page 11-218

11.3 --any_sort_order=order on page 11-221

11.7 --base_platform on page 11-225

7.11 Preprocessing a scatter file on page 7-149

11.48 --first=section_id on page 11-270

11.69 --last=section_id on page 11-295

11.114 --ro_base=address on page 11-344

11.115 --ropi on page 11-345

11.116 --rosplit on page 11-346

11.117 --rw_base=address on page 11-347

11.118 --rwpi on page 11-348

11.127 --split on page 11-359

11.157 --xo_base=address on page 11-389

11.161 --zi_base=address on page 11-393

11.11 --bpabi on page 11-230
11.32 --dll on page 11-254
11.100 --partial on page 11-329
11.111 --reloc on page 11-341
Chapter 7 Scatter-loading Features on page 7-104

11.121 --section_index_display=type

Changes the display of the index column when printing memory map output.

Syntax

`--section_index_display=`*`type`*

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

`cmdline`

Alters the display of the map file to show the order that a section appears on the command-line. The command-line order is defined as `File.Object.Section` where:

- `Section` is the section index, `sh_idx`, of the `Section` in the `Object`.
- `Object` is the order that `Object` appears in the `File`.
- `File` is the order the `File` appears on the command line.

The order the `Object` appears in the `File` is only significant if the file is an `ar` archive.

`internal`

The index value represents the order in which the linker creates the section.

`input`

The index value represents the section index of the section in the original input file. This is useful when you want to find the exact section in an input object.

Usage

Use this option with `--map`.

Default

The default is `--section_index_display=internal`.

Related reference

11.85 --map, --no_map on page 11-314
11.142 --tiebreaker=option on page 11-374

11.122 --show_cmdline

Outputs the command line used by the linker.

Usage

Shows the command line after processing by the linker, and can be useful to check:

- The command line a build system is using.
- How the linker is interpreting the supplied command line, for example, the ordering of command-line options.

The commands are shown normalized, and the contents of any via files are expanded.

The output is sent to the standard error stream (`stderr`).

Related reference

11.56 --help on page 11-278
11.155 --via=filename on page 11-387

11.123 --show_full_path

Displays the full path name of an object in any diagnostic messages.

Usage

If the file representing object `obj` has full path name `path/to/obj` then the linker displays `path/to/obj` instead of `obj` in any diagnostic message.

Related reference

11.124 --show_parent_lib on page 11-355
11.125 --show_sec_idx on page 11-356

11.124 --show_parent_lib

Displays the library name containing an object in any diagnostic messages.

Usage

If an object `obj` comes from library `lib`, then this option displays `lib(obj)` instead of `obj` in any diagnostic messages.

Related reference

11.123 --show_full_path on page 11-354
11.125 --show_sec_idx on page 11-356

11.125 --show_sec_idx

Displays the section index, `sh_idx`, of section in the originating object.

Example

If section `sec` has section index 3 then it is displayed as `sec:3` in all diagnostic messages.

Related reference

11.123 --show_full_path on page 11-354
11.124 --show_parent_lib on page 11-355

11.126 --sort=algorithm

Specifies the sorting algorithm used by the linker to determine the order of sections in an output image.

Syntax

`--sort=`*`algorithm`*

where *`algorithm`* is one of the following:

`Alignment`

Sorts input sections by ascending order of alignment value.

`AlignmentLexical`

Sorts input sections by ascending order of alignment value, then sorts lexically.

`AvgCallDepth`

Sorts all T32 code before A32 code and then sorts according to the approximated average call depth of each section in ascending order.

Use this algorithm to minimize the number of long branch veneers.

——— **Note** ———

The approximation of the average call depth depends on the order of input sections. Therefore, this sorting algorithm is more dependent on the order of input sections than using, say, `RunningDepth`.

———

`BreadthFirstCallTree`

This is similar to the `CallTree` algorithm except that it uses a breadth-first traversal when flattening the Call Tree into a list.

`CallTree`

The linker flattens the call tree into a list containing the read-only code sections from all execution regions that have `CallTree` sorting enabled.

Sections in this list are copied back into their execution regions, followed by all the non read-only code sections, sorted lexically. Doing this ensures that sections calling each other are placed close together.

——— **Note** ———

This sorting algorithm is less dependent on the order of input sections than using either `RunningDepth` or `AvgCallDepth`.

———

`Lexical`

Sorts according to the name of the section and then by input order if the names are the same.

`LexicalAlignment`

Sorts input sections lexically, then according to the name of the section, and then by input order if the names are the same.

`LexicalState`

Sorts T32 code before A32 code, then sorts lexically.

`List`

Provides a list of the available sorting algorithms. The linker terminates after displaying the list.

ObjectCode

Sorts code sections by tiebreaker. All other sections are sorted lexically. This is most useful when used with `--tiebreaker=cmdline` because it attempts to group all the sections from the same object together in the memory map.

RunningDepth

Sorts all T32 code before A32 code and then sorts according to the running depth of the section in ascending order. The running depth of a section S is the average call depth of all the sections that call S, weighted by the number of times that they call S.

Use this algorithm to minimize the number of long branch veneers.

Usage

The sorting algorithms conform to the standard rules, placing input sections in ascending order by attributes.

You can also specify sort algorithms in a scatter file for individual execution regions. Use the `SORTTYPE` keyword to do this.

Note

The `SORTTYPE` execution region attribute overrides any sorting algorithm that you specify with this option.

Default

The default algorithm is `--sort=Lexical`. In large region mode, the default algorithm is `--sort=AvgCallDepth`.

Related concepts

3.3 Section placement with the linker on page 3-48

8.4 Execution region descriptions on page 8-173

Related reference

11.142 --tiebreaker=option on page 11-374

11.68 --largeregions, --no_largeregions on page 11-293

8.4.3 Execution region attributes on page 8-175

11.127 --split

Splits the default load region, that contains the RO and RW output sections, into separate load regions.

Usage

The default load region is split into the following load regions:

- One region containing the RO output section. The default load address is `0x8000`, but you can specify a different address with the `--ro_base` option.
- One region containing the RW and ZI output sections. The default load address is `0x0`, but you can specify a different address with the `--rw_base` option.

Both regions are root regions.

Considerations when execute-only sections are present

For images containing *execute-only* (XO) sections, an XO execution region is placed at the address specified by `--ro_base`. The RO execution region is placed immediately after the XO region.

If you specify `--xo_base` *`address`*, then the XO execution region is placed at the specified address in a separate load region from the RO execution region.

Restrictions

You cannot use `--split` with `--scatter`.

Related concepts

3.1 The structure of an Arm® ELF image on page 3-33

Related reference

11.120 --scatter=filename on page 11-350

11.128 --startup=symbol, --no_startup

Enables the linker to use alternative C libraries with a different startup symbol if required.

Syntax

`--startup=`*`symbol`*

By default, *`symbol`* is set to `__main`.

`--no_startup` does not take a *`symbol`* argument.

Usage

The linker includes the C library startup code if there is a reference to a symbol that is defined by the C library startup code. This symbol reference is called the startup symbol. It is automatically created by the linker when it sees a definition of `main()`. The `--startup` option enables you to change this symbol reference.

- If the linker finds a definition of `main()` and does not find a definition of *`symbol`*, then it generates an error.
- If the linker finds a definition of `main()` and a definition of *`symbol`*, but no entry point is specified, then it generates a warning.

`--no_startup` does not add a reference.

Default

The default is `--startup=__main`.

Related reference

11.41 --entry=location on page 11-263

11.129 --stdlib

Specifies the C++ library to use.

Note

This topic includes descriptions of [ALPHA] features. See *Support level definitions* on page 1-21.

Syntax

`--stdlib=`*`library_option`*

where *`library_option`* is one of the following:

`libc++`
The standard C++ library.

`threaded_libc++ [ALPHA]`
The threaded standard C++ library.

Usage

C++ objects compiled with `armclang` and linked with `armlink` use libc++ by default.

Related information

C++ libraries and multithreading [ALPHA]

11.130 --strict

Instructs the linker to perform additional conformance checks, such as reporting conditions that might result in failures.

Usage

`--strict` causes the linker to check for taking the address of:

- A non-interworking location from a non-interworking location in a different state.
- A RW location from a location that uses the static base register R9.
- A STKCKD function in an image that contains USESV7 functions.
- A ~STKCKD function in an image that contains STKCKD functions.

Note

STKCKD functions reserve register r10 for Stack Checking, ~STKCKD functions use register r10 as variable register v7 and USESV7 functions use register r10 as v7. See the *Procedure Call Standard for the Arm® Architecture* (AAPCS) for more information about v7.

An example of a condition that might result in failure is taking the address of an interworking function from a non-interworking function.

Related concepts

3.13 The strict family of linker options on page 3-64

Related reference

11.131 --strict_flags, --no_strict_flags on page 11-363

11.132 --strict_ph, --no_strict_ph on page 11-364

11.134 --strict_relocations, --no_strict_relocations on page 11-366

11.135 --strict_symbols, --no_strict_symbols on page 11-367

11.136 --strict_visibility, --no_strict_visibility on page 11-368

11.30 --diag_suppress=tag[,tag,...] on page 11-252

11.31 --diag_warning=tag[,tag,...] on page 11-253

11.27 --diag_error=tag[,tag,...] on page 11-249

11.42 --errors=filename on page 11-264

Related information

Procedure Call Standard for the Arm Architecture (AAPCS)

11.131 --strict_flags, --no_strict_flags

Prevent or allow the generation of the EF_ARM_HASENTRY flag.

Usage

The option `--strict_flags` prevents the EF_ARM_HASENTRY flag from being generated.

Default

The default is `--no_strict_flags`.

Related concepts

3.13 The strict family of linker options on page 3-64

Related reference

11.130 --strict on page 11-362

11.132 --strict_ph, --no_strict_ph on page 11-364

11.134 --strict_relocations, --no_strict_relocations on page 11-366

11.135 --strict_symbols, --no_strict_symbols on page 11-367

11.136 --strict_visibility, --no_strict_visibility on page 11-368

Related information

Arm ELF Specification (SWS ESPC 0003 B-02)

11.132 --strict_ph, --no_strict_ph

Enables or disables the sorting of the Program Header Table entries.

Usage

The linker writes the contents of load regions into the output ELF file in the order that load regions are written in the scatter file. Each load region is represented by one ELF program segment. In RVCT v2.2 the Program Header table entries describing the program segments are given the same order as the program segments in the ELF file. To be more compliant with the ELF specification, in RVCT v3.0 and later the Program Header table entries are sorted in ascending virtual address order.

Use the `--no_strict_ph` command-line option to switch off the sorting of the Program Header table entries.

Default

The default is `--strict_ph`.

Related concepts

3.13 The strict family of linker options on page 3-64

Related reference

11.130 --strict on page 11-362

11.131 --strict_flags, --no_strict_flags on page 11-363

11.134 --strict_relocations, --no_strict_relocations on page 11-366

11.135 --strict_symbols, --no_strict_symbols on page 11-367

11.136 --strict_visibility, --no_strict_visibility on page 11-368

11.133 --strict_preserve8_require8

Enables the generation of the `armlink` diagnostic `L6238E` when a function that is not tagged as preserving eight-byte alignment of the stack calls a function that is tagged as requiring eight-byte alignment of the stack.

Note

This option controls only the instances of error `L6283E` that relate to the preserve eight-byte stack alignment and require eight-byte stack alignment relationship, not any other instances of that error.

When a function is known to preserve eight-byte alignment of the stack, `armclang` assigns the build attribute `Tag_ABI_align_preserved` to that function. However, the `armclang` integrated assembler does not automatically assign this attribute to assembly code.

By default, `armlink` does not check for the build attribute `Tag_ABI_align_preserved`. Therefore, when you specify `--strict_preserve8_require8`, and `armlink` generates error `L6238E`, then you must check that your assembly code preserves eight-byte stack alignment. If it does, then add the following directive to your assembly code:

```
.eabi_attribute Tag_ABI_align_preserved, 1
```

Related information

L6238E

11.134 --strict_relocations, --no_strict_relocations

Enables you to ensure *Application Binary Interface* (ABI) compliance of legacy or third party objects.

Usage

This option checks that branch relocation applies to a branch instruction bit-pattern. The linker generates an error if there is a mismatch.

Use `--strict_relocations` to instruct the linker to report instances of obsolete and deprecated relocations.

Relocation errors and warnings are most likely to occur if you are linking object files built with previous versions of the Arm tools.

Default

The default is `--no_strict_relocations`.

Related concepts

3.13 The strict family of linker options on page 3-64

Related reference

11.130 --strict on page 11-362

11.131 --strict_flags, --no_strict_flags on page 11-363

11.132 --strict_ph, --no_strict_ph on page 11-364

11.135 --strict_symbols, --no_strict_symbols on page 11-367

11.136 --strict_visibility, --no_strict_visibility on page 11-368

11.135 --strict_symbols, --no_strict_symbols

Checks whether or not a mapping symbol type matches an ABI symbol type.

Usage

The option `--strict_symbols` checks that the mapping symbol type matches ABI symbol type. The linker displays a warning if the types do not match.

A mismatch can occur only if you have hand-coded your own assembler.

Default

The default is `--no_strict_symbols`.

Example

In the following assembler code the symbol `sym` has type `STT_FUNC` and is A32:

```
    .section mycode,"x"
    .word sym + 4
    .code 32
    .type sym, "function"
sym:
    mov  r0, r0
    .code 16
    mov  r0, r0
    .end
```

The difference in behavior is the meaning of `.word sym + 4`:

- In pre-ABI linkers the state of the symbol is the state of the mapping symbol at that location. In this example, the state is T32.
- In ABI linkers the type of the symbol is the state of the location of symbol plus the offset.

Related concepts

3.13 The strict family of linker options on page 3-64

6.1 About mapping symbols on page 6-88

Related reference

11.130 --strict on page 11-362

11.131 --strict_flags, --no_strict_flags on page 11-363

11.132 --strict_ph, --no_strict_ph on page 11-364

11.134 --strict_relocations, --no_strict_relocations on page 11-366

11.136 --strict_visibility, --no_strict_visibility on page 11-368

11.136 --strict_visibility, --no_strict_visibility

Prevents or allows a hidden visibility reference to match against a shared object.

Usage

A linker is not permitted to match a symbol reference with `STT_HIDDEN` visibility to a dynamic shared object. Some older linkers might permit this.

Use `--no_strict_visibility` to permit a hidden visibility reference to match against a shared object.

Default

The default is `--strict_visibility`.

Related concepts

3.13 The strict family of linker options on page 3-64

Related reference

11.130 --strict on page 11-362

11.131 --strict_flags, --no_strict_flags on page 11-363

11.132 --strict_ph, --no_strict_ph on page 11-364

11.134 --strict_relocations, --no_strict_relocations on page 11-366

11.135 --strict_symbols, --no_strict_symbols on page 11-367

11.137 --symbols, --no_symbols

Enables or disables the listing of each local and global symbol used in the link step, and its value.

Note

This does not include mapping symbols output to `stdout`. Use `--list_mapping_symbols` to include mapping symbols in the output.

Default

The default is `--no_symbols`.

Related reference

11.76 --list_mapping_symbols, --no_list_mapping_symbols on page 11-303

11.138 --symdefs=filename

Creates a file containing the global symbol definitions from the output image.

Syntax

`--symdefs=`*`filename`*

where *`filename`* is the name of the text file to contain the global symbol definitions.

Default

By default, all global symbols are written to the symdefs file. If a symdefs file called *`filename`* already exists, the linker restricts its output to the symbols already listed in this file.

Note

If you do not want this behavior, be sure to delete any existing symdefs file before the link step.

Usage

If *`filename`* is specified without path information, the linker searches for it in the directory where the output image is being written. If it is not found, it is created in that directory.

You can use the symbol definitions file as input when linking another image.

Related concepts

6.5 Access symbols in another image on page 6-97

11.139 --symver_script=filename

Enables implicit symbol versioning.

Syntax

`--symver_script=`*`filename`*

where *`filename`* is a symbol version script.

11.140 --symver_soname

Enables implicit symbol versioning to force static binding.

Note

Not supported for AArch64 state.

Usage

Where a symbol has no defined version, the linker uses the *shared object name* (`SONAME`) contained in the file being linked.

Default

This is the default if you are generating a *Base Platform Application Binary Interface* (BPABI) compatible executable file but where you do not specify a version script with the option `--symver_script`.

Related concepts

9.5 Symbol versioning on page 9-205

Related information

Base Platform ABI for the Arm Architecture

11.141 --tailreorder, --no_tailreorder

Moves tail calling sections immediately before their target, if possible, to optimize the branch instruction at the end of a section.

Note

Not supported for AArch64 state.

Usage

A tail calling section is a section that contains a branch instruction at the end of the section. The branch must have a relocation that targets a function at the start of a section.

Default

The default is `--no_tailreorder`.

Restrictions

The linker:

- Can only move one tail calling section for each tail call target. If there are multiple tail calls to a single section, the tail calling section with an identical section name is moved before the target. If no section name is found in the tail calling section that has a matching name, then the linker moves the first section it encounters.
- Cannot move a tail calling section out of its execution region.
- Does not move tail calling sections before inline veneers.

Related concepts

4.7 Linker reordering of tail calling sections on page 4-75

4.6 About branches that optimize to a NOP on page 4-74

Related reference

11.12 --branchnop, --no_branchnop on page 11-231

11.142 --tiebreaker=option

A tiebreaker is used when a sorting algorithm requires a total ordering of sections. It is used by the linker to resolve the order when the sorting criteria results in more than one input section with equal properties.

Syntax

`--tiebreaker=`*`option`*

where *`option`* is one of:

`creation`

The order that the linker creates sections in its internal section data structure.

When the linker creates an input section for each ELF section in the input objects, it increments a global counter. The value of this counter is stored in the section as the creation index.

The creation index of a section is unique apart from the special case of inline veneers.

`cmdline`

The order that the section appears on the linker command-line. The command-line order is defined as `File.Object.Section` where:

- `Section` is the section index, `sh_idx`, of the `Section` in the `Object`.
- `Object` is the order that `Object` appears in the `File`.
- `File` is the order the `File` appears on the command line.

The order the `Object` appears in the `File` is only significant if the file is an `ar` archive.

This option is useful if you are doing a binary difference between the results of different links, link1 and link2. If link2 has only small changes from link1, then you might want the differences in one source file to be localized. In general, creation index works well for objects, but because of the multiple pass selection of members from libraries, a small difference such as calling a new function can result in a different order of objects and therefore a different tiebreaker. The command-line index is more stable across builds.

Use this option with the `--scatter` option.

Default

The default option is `creation`.

Related reference

11.126 --sort=algorithm on page 11-357
11.85 --map, --no_map on page 11-314
11.3 --any_sort_order=order on page 11-221

11.143 --unaligned_access, --no_unaligned_access

Enable or disable unaligned accesses to data on Arm architecture-based processors.

Usage

When using `--no_unaligned_access`, the linker:

- Does not select objects from the Arm C library that allow unaligned accesses.
- Gives an error message if any input object allows unaligned accesses.

  ——— **Note** ———

  This error message can be downgraded.

  ———————————

Default

The default is `--unaligned_access`.

11.144 --undefined=symbol

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

Syntax

`--undefined=`*`symbol`*

Usage

Causes the linker to:

1. Create a symbol reference to the specified symbol name.
2. Issue an implicit `--keep=`*`symbol`* to prevent any sections brought in to define that symbol from being removed.

Related reference

11.145 --undefined_and_export=symbol on page 11-377

11.66 --keep=section_id on page 11-290

11.145 --undefined_and_export=symbol

Prevents the removal of a specified symbol if it is undefined, and pushes the symbol into the dynamic symbol table.

Syntax

`--undefined_and_export=`*`symbol`*

Usage

Causes the linker to:

1. Create a symbol reference to the specified symbol name.
2. Issue an implicit `--keep=`*`symbol`* to prevent any sections brought in to define that symbol from being removed.
3. Add an implicit `EXPORT` *`symbol`* to push the specified symbol into the dynamic symbol table.

Considerations

Be aware of the following when using this option:

- It does not change the visibility of a symbol unless you specify the `--override_visibility` option.
- A warning is issued if the visibility of the specified symbol is not high enough.
- A warning is issued if the visibility of the specified symbol is overridden because you also specified the `--override_visibility` option.
- Hidden symbols are not exported unless you specify the `--override_visibility` option.

Related reference

11.95 --override_visibility on page 11-324
11.144 --undefined=symbol on page 11-376
11.66 --keep=section_id on page 11-290
12.1 EXPORT steering file command on page 12-395

11.146 --unresolved=symbol

Takes each reference to an undefined symbol and matches it to the global definition of the specified symbol.

Syntax

`--unresolved=`*`symbol`*

`symbol` must be both defined and global, otherwise it appears in the list of undefined symbols and the link step fails.

Usage

This option is particularly useful during top-down development, because it enables you to test a partially-implemented system by matching each reference to a missing function to a dummy function.

Related reference

11.144 --undefined=symbol on page 11-376
11.145 --undefined_and_export=symbol on page 11-377

11.147 --use_definition_visibility

Enables the linker to use the visibility of the definition in preference to the visibility of a reference when combining symbols.

Usage

When the linker combines global symbols the visibility of the symbol is set with the strictest visibility of the symbols being combined. Therefore, a symbol reference with `STV_HIDDEN` visibility combined with a definition with `STV_DEFAULT` visibility results in a definition with `STV_HIDDEN` visibility.

For example, a symbol reference with `STV_HIDDEN` visibility combined with a definition with `STV_DEFAULT` visibility results in a definition with `STV_DEFAULT` visibility.

This can be useful when you want a reference to not match a Shared Library, but you want to export the definition.

Note

This option is not ELF-compliant and is disabled by default. To create ELF-compliant images, you must use symbol references with the appropriate visibility.

11.148 --userlibpath=pathlist

Specifies a list of paths that the linker is to use to search for user libraries.

Syntax

`--userlibpath=`*`pathlist`*

Where *`pathlist`* is a comma-separated list of paths that the linker is to use to search for the required libraries. Do not include spaces between the comma and the path name when specifying multiple path names, for example, *`path1,path2,path3,…,pathn`*.

Related concepts

3.9 How the linker performs library searching, selection, and scanning on page 3-60

Related reference

11.71 --libpath=pathlist on page 11-297

11.149 --veneerinject, --no_veneerinject

Enables or disables the placement of veneers outside of the sorting order for the Execution Region.

Usage

Use `--veneerinject` to allow the linker to place veneers outside of the sorting order for the Execution Region. This option is a subset of the `--largeregions` command. Use `--veneerinject` if you want to allow the veneer placement behavior described, but do not want to implicitly set the `--api` and `--sort=AvgCallDepth`.

Use `--no_veneerinject` to allow the linker use the sorting order for the Execution Region.

Use `--veneer_inject_type` to control the strategy the linker uses to place injected veneers.

The following command-line options allow stable veneer placement with large Execution Regions:

```
--veneerinject --veneer_inject_type=pool --sort=lexical
```

Default

The default is `--no_veneerinject`. The linker automatically switches to large region mode if it is required to successfully link the image. If large region mode is turned off with `--no_largeregions` then only `--veneerinject` is turned on if it is required to successfully link the image.

Note

`--veneerinject` is the default for large region mode.

Related reference

11.68 --largeregions, --no_largeregions on page 11-293

11.150 --veneer_inject_type=type on page 11-382

11.4 --api, --no_api on page 11-222

11.126 --sort=algorithm on page 11-357

11.150 --veneer_inject_type=type

Controls the veneer layout when `--largeregions` mode is on.

Syntax

`--veneer_inject_type=`*`type`*

Where *`type`* is one of:

`individual`

The linker places veneers to ensure they can be reached by the largest amount of sections that use the veneer. Veneer reuse between execution regions is permitted. This type minimizes the number of veneers that are required but disrupts the structure of the image the most.

`pool`

The linker:

1. Collects veneers from a contiguous range of the execution region.
2. Places all the veneers generated from that range into a pool.
3. Places that pool at the end of the range.

A large execution region might have more than one range and therefore more than one pool. Although this type has much less impact on the structure of image, it has fewer opportunities for reuse. This is because a range of code cannot reuse a veneer in another pool. The linker calculates the range based on the presence of branch instructions that the linker predicts might require veneers. A branch is predicted to require a veneer when either:

- A state change is required.
- The distance from source to target plus a contingency greater than the branch range.

You can set the size of the contingency with the `--veneer_pool_size=`*`size`* option. By default the contingency size is set to 102400 bytes. The `--info=veneerpools` option provides information on how the linker has placed veneer pools.

Restrictions

You must use `--largeregions` with this option.

Related reference

11.59 --info=topic[,topic,...] on page 11-281
11.149 --veneerinject, --no_veneerinject on page 11-381
11.151 --veneer_pool_size=size on page 11-383
11.68 --largeregions, --no_largeregions on page 11-293

11.151 --veneer_pool_size=size

Sets the contingency size for the veneer pool in an execution region.

Syntax

`--veneer_pool_size=pool`

where *pool* is the size in bytes.

Default

The default size is 102400 bytes.

Related reference

11.150 --veneer_inject_type=type on page 11-382

11.152 --veneershare, --no_veneershare

Enables or disables veneer sharing. Veneer sharing can cause a significant decrease in image size.

Default

The default is `--veneershare`.

Related concepts

3.6.2 Veneer sharing on page 3-53

3.6 Linker-generated veneers on page 3-53

3.6.3 Veneer types on page 3-54

3.6.4 Generation of position independent to absolute veneers on page 3-55

Related reference

11.64 --inlineveneer, --no_inlineveneer on page 11-288

11.102 --piveneer, --no_piveneer on page 11-331

11.24 --crosser_veneershare, --no_crosser_veneershare on page 11-246

11.153 --verbose

Prints detailed information about the link operation, including the objects that are included and the libraries from which they are taken.

Usage

This output is particular useful for tracing undefined symbols reference or multiply defined symbols. Because this output is typically quite long, you might want to use this command with the `--list=filename` command to redirect the information to `filename`.

Use `--verbose` to output diagnostics to `stdout`.

Related reference

11.75 --list=filename on page 11-302
11.91 --muldefweak, --no_muldefweak on page 11-320
11.146 --unresolved=symbol on page 11-378

11.154 --version_number

Displays the version of `armlink` you are using.

Usage

The linker displays the version number in the format `Mmmuuxx`, where:

- *`M`* is the major version number, `6`.
- *`mm`* is the minor version number.
- *`uu`* is the update number.
- *`xx`* is reserved for Arm internal use. You can ignore this for the purposes of checking whether the current release is a specific version or within a range of versions.

Related reference

11.56 --help on page 11-278
11.156 --vsn on page 11-388

11.155 --via=filename

Reads an additional list of input filenames and linker options from *filename*.

Syntax

`--via=`*filename*

Where *filename* is the name of a via file containing options to be included on the command line.

Usage

You can enter multiple `--via` options on the linker command line. The `--via` options can also be included within a via file.

Related reference

13.2 Via file syntax rules on page 13-405

11.156 --vsn

Displays the version information and the license details.

Note

`--vsn` is intended to report the version information for manual inspection. The `Component` line indicates the release of Arm Compiler you are using. If you need to access the version in other tools or scripts, for example in build scripts, use the output from `--version_number`.

Example

```
> armlink --vsn
Product: ARM Compiler N.n
Component: ARM Compiler N.n
Tool: armlink [tool_id]
license_type
Software supplied by: ARM Limited
```

Related reference

11.56 --help on page 11-278
11.154 --version_number on page 11-386

11.157 --xo_base=address

Specifies the base address of an *execute-only* (XO) execution region.

Syntax

`--xo_base=address`

Where *address* must be word-aligned.

Usage

When you specify `--xo_base`:

- XO sections are placed in a separate load and execution region, at the address specified.
- No ER_XO region is created when no XO sections are present.

Restrictions

You can use `--xo_base` only with the bare-metal linking model.

Note

XO memory is supported only for Armv7-M and Armv8-M architectures.

You cannot use `--xo_base` with:

- `--reloc`.
- `--ropi`.
- `--rwpi`.
- `--scatter`.

Related concepts

2.2 Bare-metal linking model on page 2-27

Related reference

11.114 --ro_base=address on page 11-344
11.115 --ropi on page 11-345
11.116 --rosplit on page 11-346
11.117 --rw_base=address on page 11-347
11.161 --zi_base=address on page 11-393
11.120 --scatter=filename on page 11-350

11.158 --xref, --no_xref

Lists to stdout all cross-references between input sections.

Default

The default is `--no_xref`.

Related reference

11.159 --xrefdbg, --no_xrefdbg on page 11-391

11.160 --xref{from|to}=object(section) on page 11-392

11.75 --list=filename on page 11-302

11.159 --xrefdbg, --no_xrefdbg

Lists to stdout all cross-references between input debug sections.

Default

The default is `--no_xrefdbg`.

Related reference

11.158 --xref, --no_xref on page 11-390

11.160 --xref{from|to}=object(section) on page 11-392

11.75 --list=filename on page 11-302

11.160 --xref{from|to}=object(section)

Lists to stdout cross-references from and to input sections.

Syntax

`--xref{from|to}=object(section)`

Usage

This option lists to stdout cross-references:

- From input *section* in *object* to other input sections.
- To input *section* in *object* from other input sections.

This is a useful subset of the listing produced by the `--xref` linker option if you are interested in references from or to a specific input section. You can have multiple occurrences of this option to list references from or to more than one input section.

Related reference

11.158 --xref, --no_xref on page 11-390
11.159 --xrefdbg, --no_xrefdbg on page 11-391
11.75 --list=filename on page 11-302

11.161 --zi_base=address

Specifies the base address of an ER_ZI execution region.

Syntax

`--zi_base=`*`address`*
Where *`address`* must be word-aligned.

Note

This option does not affect the placement of execute-only sections.

Restrictions

The linker ignores `--zi_base` if one of the following options is also specified:

- `--bpabi`.
- `--base_platform`.
- `--reloc`.
- `--rwpi`.
- `--split`.

You cannot use `--zi_base` with `--scatter`.

Related reference

11.114 --ro_base=address on page 11-344
11.115 --ropi on page 11-345
11.116 --rosplit on page 11-346
11.117 --rw_base=address on page 11-347
11.157 --xo_base=address on page 11-389
11.120 --scatter=filename on page 11-350
11.11 --bpabi on page 11-230

Chapter 12
Linker Steering File Command Reference

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

It contains the following sections:

12.1 EXPORT steering file command

Specifies that a symbol can be accessed by other shared objects or executables.

Note

A symbol can be exported only if the definition has `STV_DEFAULT` or `STV_PROTECTED` visibility. You must use the `--override_visibility` command-line option to enable the linker to override symbol visibility to `STV_DEFAULT`.

Syntax

`EXPORT` *`pattern`* `AS` *`replacement_pattern`*`[,`*`pattern`* `AS` *`replacement_pattern`*`]`

where:

`pattern`

is a string, optionally including wildcard characters (either `*` or `?`), that matches zero or more defined global symbols. If *`pattern`* does not match any defined global symbol, the linker ignores the command. The operand can match only defined global symbols.

If the symbol is not defined, the linker issues:

```
Warning: L6331W: No eligible global symbol matches pattern symbol
```

`replacement_pattern`

is a string, optionally including wildcard characters (either `*` or `?`), to which the defined global symbol is to be renamed. Wild characters must have a corresponding wildcard in *`pattern`*. The characters matched by the *`replacement_pattern`* wildcard are substituted for the *`pattern`* wildcard.

For example:

```
EXPORT my_func AS func1
```

renames and exports the defined symbol `my_func` as `func1`.

Usage

You cannot export a symbol to a name that already exists. Only one wildcard character (either `*` or `?`) is permitted in `EXPORT`.

The defined global symbol is included in the dynamic symbol table (as *`replacement_pattern`* if given, otherwise as *`pattern`*), if a dynamic symbol table is present.

Related concepts

6.6 Edit the symbol tables with a steering file on page 6-100

Related reference

12.3 IMPORT steering file command on page 12-397

11.95 --override_visibility on page 11-324

12.2 HIDE steering file command

Makes defined global symbols in the symbol table anonymous.

Syntax

`HIDE` *`pattern`*`[,`*`pattern`*`]`

where:

`pattern`

is a string, optionally including wildcard characters, that matches zero or more defined global symbols. If *`pattern`* does not match any defined global symbol, the linker ignores the command. You cannot hide undefined symbols.

Usage

You can use `HIDE` and `SHOW` to make certain global symbols anonymous in an output image or partially linked object. Hiding symbols in an object file or library can be useful as a means of protecting intellectual property, as shown in the following example:

```
; steer.txt
; Hides all global symbols
HIDE *
; Shows all symbols beginning with 'os_'
SHOW os_*
```

This example produces a partially linked object with all global symbols hidden, except those beginning with `os_`.

Link this example with the command:

```
armlink --partial input_object.o --edit steer.txt -o partial_object.o
```

You can link the resulting partial object with other objects, provided they do not contain references to the hidden symbols. When symbols are hidden in the output object, `SHOW` commands in subsequent link steps have no effect on them. The hidden references are removed from the output symbol table.

Related concepts

6.6 Edit the symbol tables with a steering file on page 6-100

Related reference

12.7 SHOW steering file command on page 12-402

11.36 --edit=file_list on page 11-258

11.100 --partial on page 11-329

12.3 IMPORT steering file command

Specifies that a symbol is defined in a shared object at runtime.

Note

A symbol can be imported only if the reference has `STV_DEFAULT` visibility. You must use the `--override_visibility` command-line option to enable the linker to override symbol visibility to `STV_DEFAULT`.

Syntax

`IMPORT` *`pattern`* `AS` *`replacement_pattern`*`[,`*`pattern`* `AS` *`replacement_pattern`*`]`

where:

`pattern`

is a string, optionally including wildcard characters (either `*` or `?`), that matches zero or more undefined global symbols. If *`pattern`* does not match any undefined global symbol, the linker ignores the command. The operand can match only undefined global symbols.

`replacement_pattern`

is a string, optionally including wildcard characters (either `*` or `?`), to which the symbol is to be renamed. Wild characters must have a corresponding wildcard in *`pattern`*. The characters matched by the *`pattern`* wildcard are substituted for the *`replacement_pattern`* wildcard.

For example:

```
IMPORT my_func AS func
```

imports and renames the undefined symbol `my_func` as `func`.

Usage

You cannot import a symbol that has been defined in the current shared object or executable. Only one wildcard character (either `*` or `?`) is permitted in `IMPORT`.

The undefined symbol is included in the dynamic symbol table (as *`replacement_pattern`* if given, otherwise as *`pattern`*), if a dynamic symbol table is present.

Note

The `IMPORT` command only affects undefined global symbols. Symbols that have been resolved by a shared library are implicitly imported into the dynamic symbol table. The linker ignores any `IMPORT` directive that targets an implicitly imported symbol.

Related concepts

6.6 Edit the symbol tables with a steering file on page 6-100

Related reference

11.95 --override_visibility on page 11-324

12.1 EXPORT steering file command on page 12-395

12.4 RENAME steering file command

Renames defined and undefined global symbol names.

Syntax

`RENAME` *`pattern`* `AS` *`replacement_pattern`*`[,`*`pattern`* `AS` *`replacement_pattern`*`]`

where:

`pattern`

is a string, optionally including wildcard characters (either `*` or `?`), that matches zero or more global symbols. If *`pattern`* does not match any global symbol, the linker ignores the command. The operand can match both defined and undefined symbols.

`replacement_pattern`

is a string, optionally including wildcard characters (either `*` or `?`), to which the symbol is to be renamed. Wildcard characters must have a corresponding wildcard in *`pattern`*. The characters matched by the *`pattern`* wildcard are substituted for the *`replacement_pattern`* wildcard.

For example, for a symbol named `func1`:

```
RENAME f* AS my_f*
```

renames `func1` to `my_func1`.

Usage

You cannot rename a symbol to a global symbol name that already exists, even if the target symbol name is being renamed itself.

You cannot rename a symbol to the same name as another symbol. For example, you cannot do the following:

```
RENAME foo1 AS bar
RENAME foo2 AS bar

Error: L6281E: Cannot rename both foo2 and foo1 to bar.
```

Renames only take effect at the end of the link step. Therefore, renaming a symbol does not remove its original name. For example, given an image containing the symbols `func1` and `func2`, you cannot do the following:

```
RENAME func1 AS func2
RENAME func2 AS func3

Error: L6282E: Cannot rename func1 to func2 as a global symbol of that name exists
```

Only one wildcard character (either `*` or `?`) is permitted in `RENAME`.

Example

Given an image containing the symbols `func1`, `func2`, and `func3`, you might have a steering file containing the following commands:

```
; invalid, func2 already exists
RENAME func1 AS func2

; valid
RENAME func3 AS b2

; invalid, func3 still exists because the link step is not yet complete
RENAME func2 AS func3
```

Related concepts

6.6 Edit the symbol tables with a steering file on page 6-100

12.5 REQUIRE steering file command

Creates a DT_NEEDED tag in the dynamic array.

DT_NEEDED tags specify dependencies to other shared objects used by the application, for example, a shared library.

Syntax

REQUIRE *pattern*[,*pattern*]

where:

pattern

is a string representing a filename. No wild characters are permitted.

Usage

The linker inserts a DT_NEEDED tag with the value of *pattern* into the dynamic array. This tells the dynamic loader that the file it is currently loading requires *pattern* to be loaded.

Note

DT_NEEDED tags inserted as a result of a REQUIRE command are added after DT_NEEDED tags generated from shared objects or *dynamically linked libraries* (DLLs) placed on the command line.

Related concepts

6.6 Edit the symbol tables with a steering file on page 6-100

12.6 RESOLVE steering file command

Matches specific undefined references to a defined global symbol.

Syntax

`RESOLVE` *`pattern`* `AS` *`defined_pattern`*

where:

`pattern`

is a string, optionally including wildcard characters (either `*` or `?`), that matches zero or more undefined global symbols. If *`pattern`* does not match any undefined global symbol, the linker ignores the command. The operand can match only undefined global symbols.

`defined_pattern`

is a string, optionally including wildcard characters, that matches zero or more defined global symbols. If *`defined_pattern`* does not match any defined global symbol, the linker ignores the command. You cannot match an undefined reference to an undefined symbol.

Usage

`RESOLVE` is an extension of the existing `armlink --unresolved` command-line option. The difference is that `--unresolved` enables all undefined references to match one single definition, whereas `RESOLVE` enables more specific matching of references to symbols.

The undefined references are removed from the output symbol table.

`RESOLVE` works when performing partial-linking and when linking normally.

Example

You might have two files `file1.c` and `file2.c`, as shown in the following example:

```
file1.c
extern int foo;
extern void MP3_Init(void);
extern void MP3_Play(void);
int main(void)
{
  int x = foo + 1;
  MP3_Init();
  MP3_Play();
  return x;
}

file2.c:
int foobar;
void MyMP3_Init()
{
}
void MyMP3_Play()
{
}
```

Create a steering file, `ed.txt`, containing the line:

`RESOLVE MP3* AS MyMP3*.`

Enter the following command:

```
armlink file1.o file2.o --edit ed.txt --unresolved foobar
```

This command has the following effects:

- The references from `file1.o` (`foo`, `MP3_Init()` and `MP3_Play()`) are matched to the definitions in `file2.o` (`foobar`, `MyMP3_Init()` and `MyMP3_Play()` respectively), as specified by the steering file `ed.txt`.
- The `RESOLVE` command in `ed.txt` matches the `MP3` functions and the `--unresolved` option matches any other remaining references, in this case, `foo` to `foobar`.
- The output symbol table, whether it is an image or a partial object, does not contain the symbols `foo`, `MP3_Init` or `MP3_Play`.

Related concepts

6.6 Edit the symbol tables with a steering file on page 6-100

Related reference

11.36 --edit=file_list on page 11-258
11.146 --unresolved=symbol on page 11-378

12.7 SHOW steering file command

Makes global symbols visible.

The `SHOW` command is useful if you want to make a specific symbol visible that is hidden using a `HIDE` command with a wildcard.

Syntax

`SHOW pattern[,pattern]`

where:

pattern

is a string, optionally including wildcard characters, that matches zero or more global symbols. If *pattern* does not match any global symbol, the linker ignores the command.

Usage

The usage of `SHOW` is closely related to that of `HIDE`.

Related concepts

6.6 Edit the symbol tables with a steering file on page 6-100

Related reference

12.2 HIDE steering file command on page 12-396

Chapter 13
Via File Syntax

Describes the syntax of via files accepted by `armlink`.

It contains the following sections:

13.1 Overview of via files

Via files are plain text files that allow you to specify linker command-line arguments and options.

Typically, you use a via file to overcome the command-line length limitations. However, you might want to create multiple via files that:

- Group similar arguments and options together.
- Contain different sets of arguments and options to be used in different scenarios.

Note

In general, you can use a via file to specify any command-line option to a tool, including `--via`. This means that you can call multiple nested via files from within a via file.

Via file evaluation

When the linker is invoked it:

1. Replaces the first specified `--via` *`via_file`* argument with the sequence of argument words extracted from the via file, including recursively processing any nested `--via` commands in the via file.
2. Processes any subsequent `--via` *`via_file`* arguments in the same way, in the order they are presented.

That is, via files are processed in the order you specify them, and each via file is processed completely including processing nested via files before processing the next via file.

Related reference

13.2 Via file syntax rules on page 13-405
11.155 --via=filename on page 11-387

13.2 Via file syntax rules

Via files must conform to some syntax rules.

- A via file is a text file containing a sequence of words. Each word in the text file is converted into an argument string and passed to the tool.
- Words are separated by whitespace, or the end of a line, except in delimited strings, for example:

  `--paged --pagesize=0x4000` (two words)

  `--paged--pagesize=0x4000` (one word)
- The end of a line is treated as whitespace, for example:

  ```
  --paged
  --pagesize=0x4000
  ```

  This is equivalent to:

  `--paged --pagesize=0x4000`
- Strings enclosed in quotation marks (`"`), or apostrophes (`'`) are treated as a single word. Within a quoted word, an apostrophe is treated as an ordinary character. Within an apostrophe delimited word, a quotation mark is treated as an ordinary character.

  Use quotation marks to delimit filenames or path names that contain spaces, for example:

  `--errors C:\My Project\errors.txt` (three words)

  `--errors "C:\My Project\errors.txt"` (two words)

  Use apostrophes to delimit words that contain quotes, for example:

  `-DNAME='"Arm Compiler"'` (one word)
- Characters enclosed in parentheses are treated as a single word, for example:

  `--option(x, y, z)` (one word)

  `--option (x, y, z)` (two words)
- Within quoted or apostrophe delimited strings, you can use a backslash (\) character to escape the quote, apostrophe, and backslash characters.
- A word that occurs immediately next to a delimited word is treated as a single word, for example:

  `--errors"C:\Project\errors.txt"`

  This is treated as the single word:

  `--errorsC:\Project\errors.txt`
- Lines beginning with a semicolon (`;`) or a hash (`#`) character as the first nonwhitespace character are comment lines. A semicolon or hash character that appears anywhere else in a line is not treated as the start of a comment, for example:

  ```
  -o objectname.axf      ;this is not a comment
  ```

  A comment ends at the end of a line, or at the end of the file. There are no multi-line comments, and there are no part-line comments.

Related concepts

13.1 Overview of via files on page 13-404

Related reference

11.155 --via=filename on page 11-387