ARM® Compiler v5.04 for µVision

Version 5

armlink User Guide

ARM DUI0377E

ARM® Compiler v5.04 for µVision

armlink User Guide

Release Information

Document History

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

Non-Confidential Proprietary Notice

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

www.arm.com

Contents

ARM® Compiler v5.04 for µVision armlink User Guide

List of Figures
ARM® Compiler v5.04 for µVision armlink User Guide

List of Tables
ARM® Compiler v5.04 for µVision armlink User Guide

Preface

This preface introduces the *ARM® Compiler v5.04 for µVision armlink User Guide* .

It contains the following:

- *About this book* on page 15.

About this book

ARM Compiler for µVision armlink User Guide. This manual provides user information for the ARM linker. It describes the basic linker functionality, image structure, how to access image symbols, and how to use scatter files. It is also available as a PDF.

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 Linker Command-line Options

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

Chapter 10 Linker Steering File Command Reference

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

Chapter 11 Via File Syntax

Describes the syntax of via files accepted by the `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.

`mono`space
: 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.
- The number ARM DUI0377E.
- The page number(s) to which your comments refer.
- A concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

Other information

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

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 references

Linker command-line options listed in groups.

9.59 input-file-list on page 9-307.

9 Linker Command-line Options on page 9-238.

1.3 Linker command-line options listed in groups

Groupings of the linker command-line options.

Options that control library files and paths

- --libpath=pathlist.
- --library_type=lib.
- --reduce_paths, --no_reduce_paths.
- --scanlib, --no_scanlib.
- --thumb2_library, --no_thumb2_library.
- --userlibpath=pathlist.

Options that control the linking of object files

- --match=crossmangled.
- --strict.
- --strict_ph, --no_strict_ph.
- --strict_relocations, --no_strict_relocations.
- --unresolved=symbol.

Options that control the output

- --combreloc, --no_combreloc.
- --ldpartial.
- --output=filename.
- --partial.
- --reloc.

Options to specify the image memory map

- --fpic.
- --predefine="string".
- --ro_base=address.
- --ropi.
- --rosplit.
- --rw_base=address.
- --rwpi.
- --scatter=filename.
- --split.
- --xo_base=address.
- --zi_base=address.

Options that control debug information in an image

- --bestdebug, --no_bestdebug.
- --compress_debug, --no_compress_debug.
- --debug, --no_debug.
- --dynamic_debug.
- --eager_load_debug, --no_eager_load_debug.
- --emit_debug_overlay_relocs.
- --emit_debug_overlay_section.
- --emit_non_debug_relocs.

Options that control the content of an image

- --any_contingency.
- --any_placement=algorithm.
- --any_sort_order=order.
- --api, --no_api.
- --arm_only.
- --autoat, --no_autoat.
- --blx_arm_thumb, --no_blx_arm_thumb.
- --blx_thumb_arm, --no_blx_thumb_arm.
- --branchnop, --no_branchnop.
- --comment_section, --no_comment_section.
- --cppinit, --no_cppinit.
- --cpu=name.
- --datacompressor=opt.
- --edit=file_list.
- --emit_relocs.
- --entry=location.
- --exceptions, --no_exceptions.
- --exceptions_tables=action.
- --filtercomment, --no_filtercomment.
- --fini=symbol.
- --first=section_id.
- --force_explicit_attr.
- --force_so_throw, --no_force_so_throw.
- --fpu=name.
- --import_unresolved, --no_import_unresolved.
- --init=symbol.
- --inline, --no_inline.
- --inline_type=type.
- --keep=section_id.
- --keep_protected_symbols.
- --largeregions, --no_largeregions.
- --last=section_id.
- --locals, --no_locals.
- --max_visibility=type.
- --merge, --no_merge.
- --muldefweak, --no_muldefweak.
- --override_visibility.
- --pad=num.
- --paged.
- --pagesize=pagesize.
- --ref_cpp_init, --no_ref_cpp_init.
- --remove, --no_remove.
- --sort=algorithm.
- --startup=symbol, --no_startup.
- --strict_flags, --no_strict_flags.
- --tailreorder, --no_tailreorder.
- --tiebreaker=option.
- --undefined=symbol.
- --undefined_and_export=symbol.

- --use_definition_visibility.
- --vfemode=mode.

Options that control veneer generation

- --crosser_veneershare, --no_crosser_veneershare.
- --inlineveneer, --no_inlineveneer.
- --max_veneer_passes=value.
- --piveneer, --no_piveneer.
- --veneerinject,--no_veneerinject.
- --veneer_inject_type=type.
- --veneer_pool_size=size.
- --veneershare, --no_veneershare.

Options that control byte addressing mode

- --be8.
- --be32.

Options that control the extraction and presentation of image information

- --callgraph, --no_callgraph.
- --callgraph_file=filename.
- --callgraph_output=fmt.
- --cgfile=type.
- --cgsymbol=type.
- --cgundefined=type.
- --feedback=filename.
- --feedback_image=option.
- --feedback_type=type.
- --info=topic[,topic,...].
- --info_lib_prefix=opt.
- --list_mapping_symbols, --no_list_mapping_symbols.
- --load_addr_map_info, --no_load_addr_map_info.
- --mangled, --unmangled.
- --map, --no_map.
- --section_index_display=type.
- --symbols, --no_symbols.
- --symdefs=filename.
- --xref, --no_xref.
- --xrefdbg, --no_xrefdbg.
- --xref{from|to}=object(section).

——— **Note** ———

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

———————————

Options that control diagnostic messages

- --diag_error=tag[,tag,...].
- --diag_remark=tag[,tag,...].
- --diag_style=arm|ide|gnu.
- --diag_suppress=tag[,tag,...].
- --diag_warning=tag[,tag,...].
- --errors=filename.
- --list=filename.
- --remarks.
- --show_full_path.
- --show_parent_lib.
- --show_sec_idx.
- --strict_enum_size, --no_strict_enum_size.
- --strict_symbols, --no_strict_symbols.
- --strict_visibility, --no_strict_visibility.
- --strict_wchar_size, --no_strict_wchar_size.
- --verbose.

Options that control alignment in legacy images

- --legacyalign, --no_legacyalign.

Miscellaneous options

- --cpu=list.
- --fpu=list.
- --help.
- --licretry.
- --show_cmdline.
- --version_number.
- --via=filename.
- --vsn.

1.4 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 the *ELF for the ARM Architecture (ARM IHI 0044)*.

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.

Related concepts

6.14 Access symbols in another image on page 6-118.

Related references

7 Scatter-loading Features on page 7-130.

10 Linker Steering File Command Reference on page 10-391.

8 Scatter File Syntax on page 8-199.

Related information

ELF for the ARM Architecture (ARM IHI 0044).

1.5 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 relocatable ELF object.

Note

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

Related concepts

2.3 Partial linking model on page 2-31.

3.11 Section placement with the linker on page 3-50.

3.1 The structure of an ARM ELF image on page 3-34.

Related information

Overview of the fromelf image converter.

1.6 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 groups and common code, data, and debug sections.
- 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

4.1 Elimination of common debug sections on page 4-74.
4.3 Elimination of unused sections on page 4-76.
3.1 The structure of an ARM ELF image on page 3-34.

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.

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.

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

Note

Execute-only (XO) memory is supported only for the bare-metal linking model. For other linking models, `armlink` generates an error if an object file contains XO sections.

Related concepts

2.2 Bare-metal linking model on page 2-29.
2.3 Partial linking model on page 2-31.

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 the following options with this model:

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

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

- `--reloc`.
- `--ro_base=`*`address`*.
- `--ropi`.
- `--rosplit`.
- `--rw_base=`*`address`*.
- `--rwpi`.
- `--split`.
- `--xo_base=`*`address`*.
- `--zi_base`.

Note

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

Related concepts

3.4 Methods of specifying an image memory map with the linker on page 3-39.

Related references

9.136 --xo_base=address on page 9-386.
9.34 --edit=file_list on page 9-279.
9.91 --reloc on page 9-340.
9.94 --ro_base=address on page 9-343.
9.95 --ropi on page 9-344.
9.96 --rosplit on page 9-345.
9.97 --rw_base=address on page 9-346.
9.98 --rwpi on page 9-347.
9.100 --scatter=filename on page 9-349.
9.107 --split on page 9-357.

9.140 --zi_base=address on page 9-390.
10 Linker Steering File Command Reference on page 10-391.

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. Other linker command-line options supported by this model are:

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

Note

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

Related concepts

6.19 What is a steering file? on page 6-124.

Related references

6.22 Steering file format on page 6-127.
10 Linker Steering File Command Reference on page 10-391.
9.34 --edit=file_list on page 9-279.
9.42 --exceptions_tables=action on page 9-287.
9.86 --partial on page 9-334.

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.

Each link stage has a different view of the image:

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

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

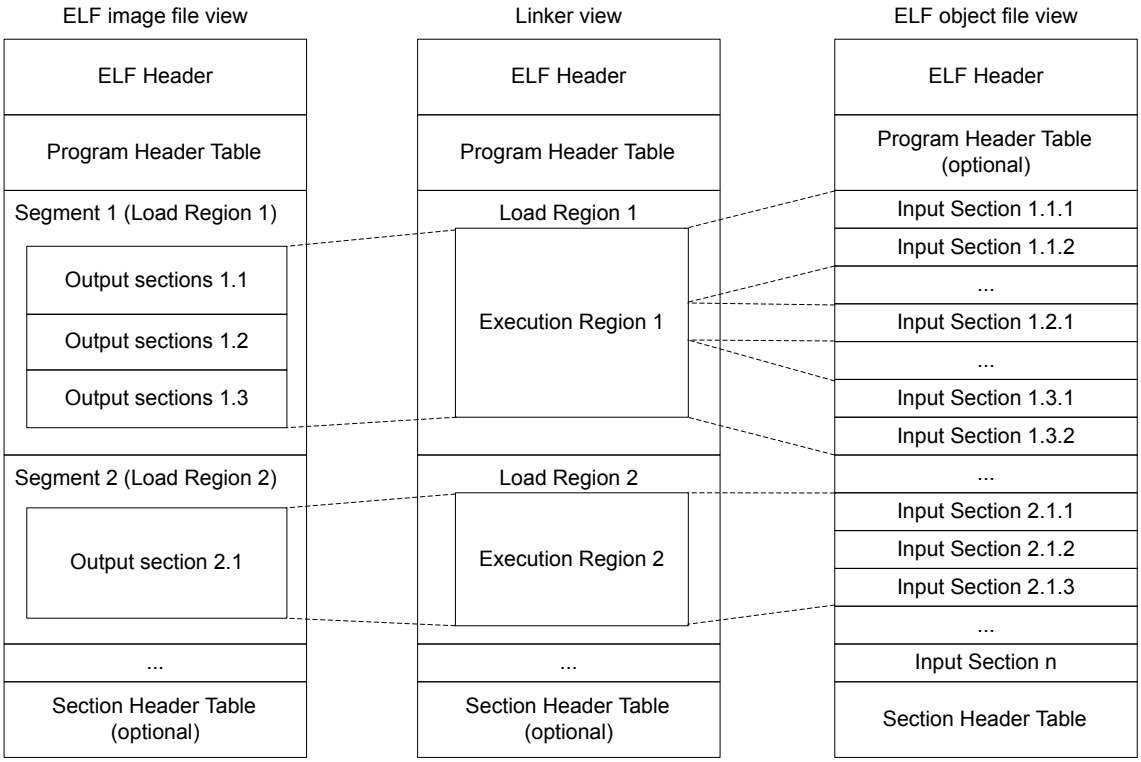

Figure 3-1 Relationship between sections, regions, and segments

Related concepts

3.2 Input sections, output sections, regions, and program segments on page 3-36.
3.3 Load view and execution view of an image on page 3-37.

3.2 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 four 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. Any XO output section is first, followed by the RO output section, then the RW output section, and finally the ZI output section. A region typically maps onto a physical memory device, such as ROM, RAM, or peripheral. 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.

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. However, this 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 ELF segment. In the final image, the XO segment immediately precedes the RO segment, unless otherwise specified by a scatter file or the `--xo-base` option.

Related concepts

3.1 The structure of an ARM ELF image on page 3-34.
3.4 Methods of specifying an image memory map with the linker on page 3-39.
3.11 Section placement with the linker on page 3-50.

3.3 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:

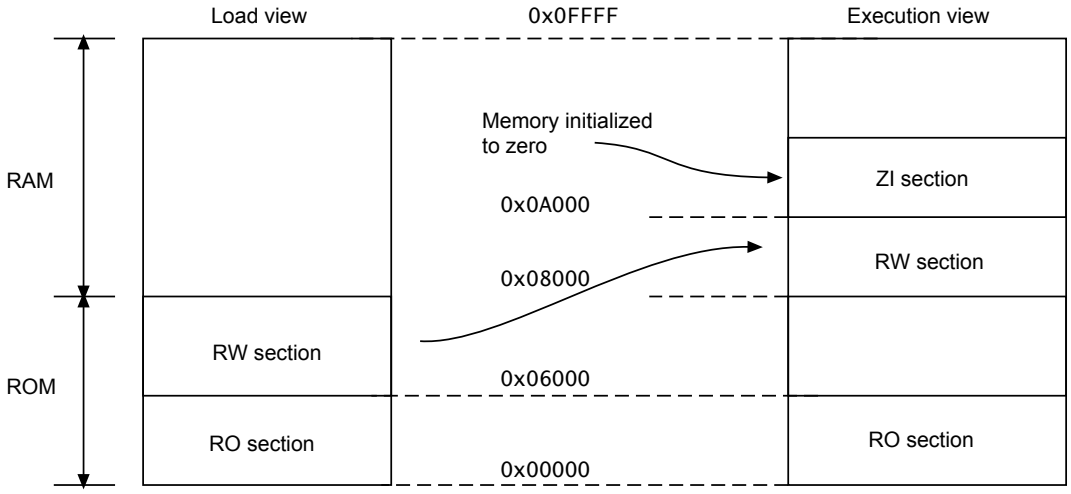

Figure 3-2 Load and execution memory maps for an image without an XO section

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

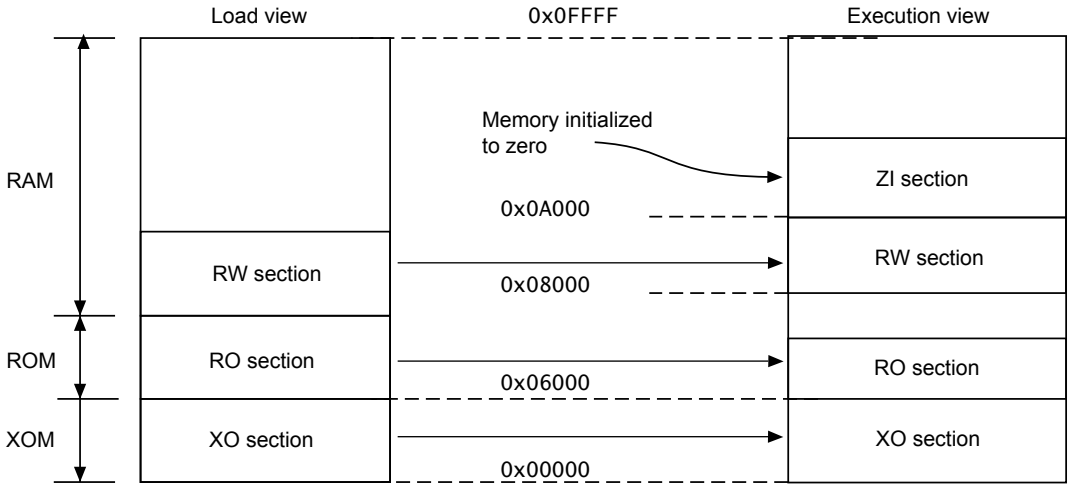

Figure 3-3 Load and execution memory maps for an image with an XO section

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 The structure of an ARM ELF image on page 3-34.

3.4 Methods of specifying an image memory map with the linker on page 3-39.

3.11 Section placement with the linker on page 3-50.

3.2 Input sections, output sections, regions, and program segments on page 3-36.

3.4 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 maps.

Depending on the complexity of the memory maps 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`.
- `--rw_base`.
- `--ropi`.
- `--rwpi`.
- `--split`.
- `--rosplit`.
- `--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.

Note

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

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`

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

Scatter file	Equivalent command-line options

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

`    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

3.3 Load view and execution view of an image on page 3-37.
3.5 Types of simple image on page 3-41.
3.1 The structure of an ARM ELF image on page 3-34.
3.2 Input sections, output sections, regions, and program segments on page 3-36.

Related references

9.48 --first=section_id on page 9-293.
9.63 --last=section_id on page 9-311.
9.94 --ro_base=address on page 9-343.
9.95 --ropi on page 9-344.
9.96 --rosplit on page 9-345.
9.97 --rw_base=address on page 9-346.
9.98 --rwpi on page 9-347.
9.100 --scatter=filename on page 9-349.
9.107 --split on page 9-357.
9.136 --xo_base=address on page 9-386.
9.140 --zi_base=address on page 9-390.

3.5 Types of simple image

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.

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

7.38 Equivalent scatter-loading descriptions for simple images on page 7-185.
3.6 Type 1 image structure, one load region and contiguous execution regions on page 3-42.
3.7 Type 2 image structure, one load region and non-contiguous execution regions on page 3-44.
3.8 Type 3 image structure, multiple load regions and non-contiguous execution regions on page 3-46.

Related references

9.94 --ro_base=address on page 9-343.
9.96 --rosplit on page 9-345.
9.97 --rw_base=address on page 9-346.
9.100 --scatter=filename on page 9-349.
9.107 --split on page 9-357.
9.136 --xo_base=address on page 9-386.
9.140 --zi_base=address on page 9-390.

3.6 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 execution regions placed contiguously in the memory map.

This approach is suitable for systems that load programs into RAM, for example, an OS bootloader or a desktop system. The following figure shows the load and execution view for a Type 1 image:

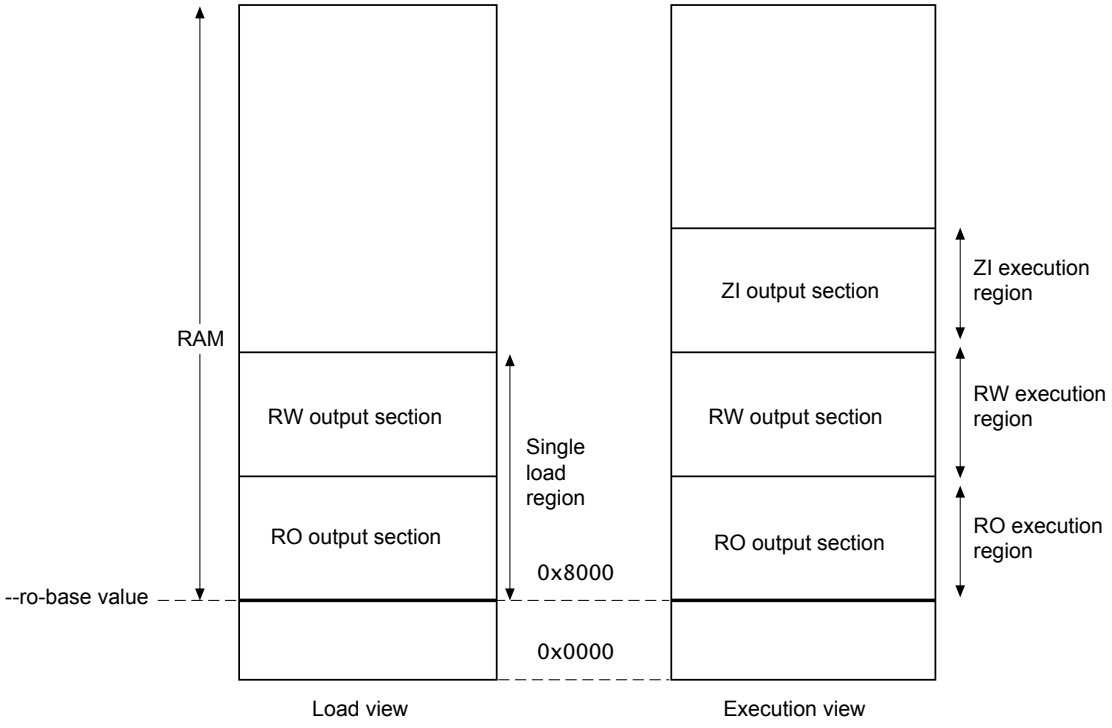

Figure 3-4 Simple Type 1 image

Use the following command for images of this type:

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

Note

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

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 *execute-only* (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, RW, and ZI execution regions are placed contiguously and immediately after the XO execution region.

Related concepts

3.1 The structure of an ARM ELF image on page 3-34.
3.2 Input sections, output sections, regions, and program segments on page 3-36.
3.3 Load view and execution view of an image on page 3-37.

Related references

9.94 --ro_base=address on page 9-343.
9.136 --xo_base=address on page 9-386.
9.140 --zi_base=address on page 9-390.

3.7 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:

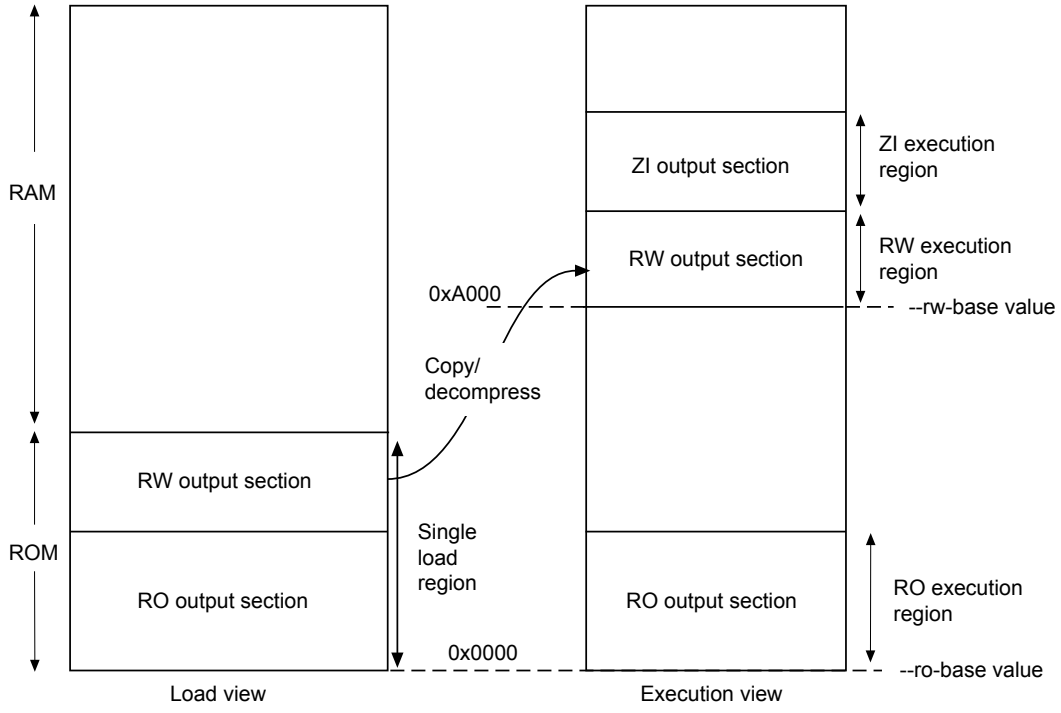

Figure 3-5 Simple Type 2 image

Use the following command for images of this type:

```
armlink --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 *execute-only* (XO) sections, the XO output section is placed at the address specified by `--ro_base`. The RO and RW output sections are placed consecutively and immediately after the XO section.

Execution view for images containing execute-only regions

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

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

Related concepts

3.1 The structure of an ARM ELF image on page 3-34.
3.2 Input sections, output sections, regions, and program segments on page 3-36.
3.3 Load view and execution view of an image on page 3-37.
3.6 Type 1 image structure, one load region and contiguous execution regions on page 3-42.

Related references

9.94 --ro_base=address on page 9-343.
9.97 --rw_base=address on page 9-346.
9.136 --xo_base=address on page 9-386.
9.140 --zi_base=address on page 9-390.

3.8 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:

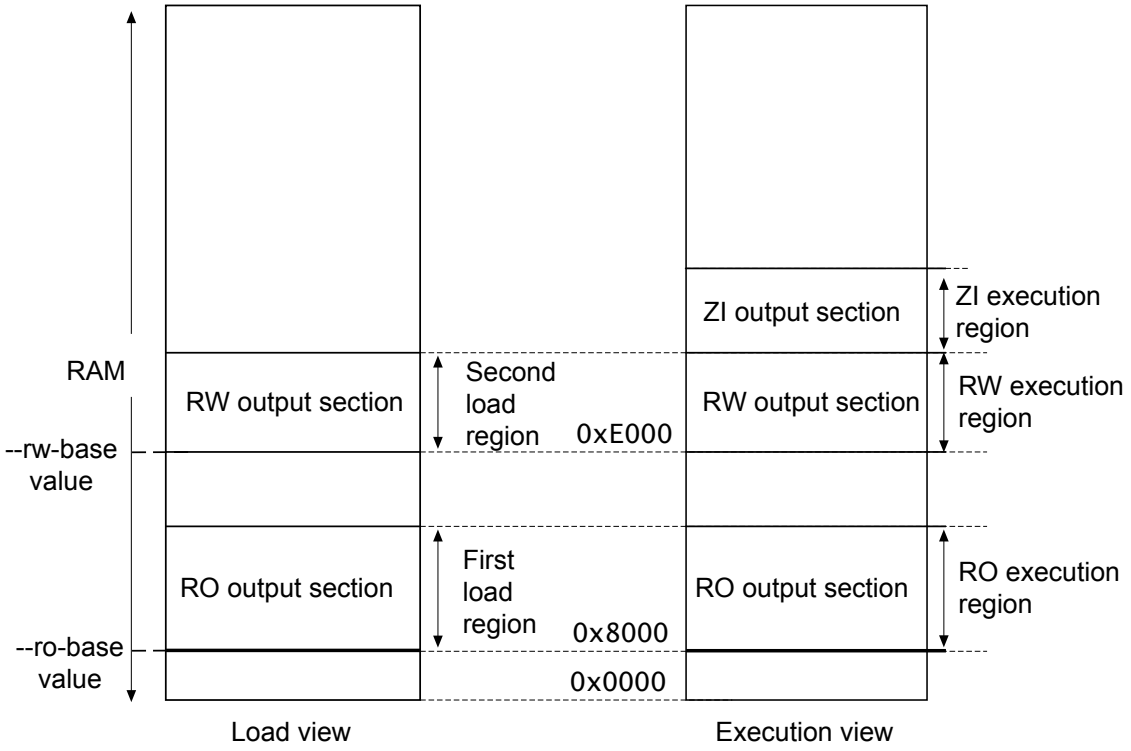

Figure 3-6 Simple Type 3 image

Use the following command for images of this type:

```
armlink --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. Both RO and RW are root regions.

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

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

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

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

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

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

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

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.

Related concepts

3.1 The structure of an ARM ELF image on page 3-34.
3.2 Input sections, output sections, regions, and program segments on page 3-36.
3.3 Load view and execution view of an image on page 3-37.
3.7 Type 2 image structure, one load region and non-contiguous execution regions on page 3-44.

Related references

9.94 --ro_base=address on page 9-343.
9.97 --rw_base=address on page 9-346.
9.136 --xo_base=address on page 9-386.
9.107 --split on page 9-357.

3.9 Image entry points

An entry point in an image is a location where program execution can start. Although there can be more than one entry point in an image, only one can be specified when linking.

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

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.

Related concepts

3.10 The initial entry point for an image on page 3-49.

Related references

9.39 --entry=location on page 9-284.

Related information

ENTRY.

3.10 The initial entry point for an image

There can be only one initial entry point for an image, otherwise the linker produces a warning.

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

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

Related concepts

7.9 Root execution regions on page 7-142.

Related references

9.39 --entry=location on page 9-284.

Related information

ENTRY.

3.11 Section placement with the linker

The linker places input sections in a specific order by default.

By default, the linker places input sections in the following order within an execution region:

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 this with the `FIRST` or `LAST` execution region 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.

Handling unassigned sections

The linker might not be able to place some input sections in any execution region. When this happens, the linker generates an error message. This might occur because your current scatter file does not permit all possible module select patterns and input section selectors. How you fix this depends on the importance of placing these sections correctly:

- If the sections must be placed at specific locations, then modify your scatter file to include specific module selectors and input section selectors as required.

- If the placement of the unassigned sections is not important, you can use one or more `.ANY` module selectors with optional input section selectors.

Examples

The following scatter file shows how the linker places sections:

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

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

Related concepts

3.3 Load view and execution view of an image on page 3-37.

Related references

9.106 --sort=algorithm on page 9-355.

3.12 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.

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

Related concepts

Default section placement by the linker.

3.1 The structure of an ARM ELF image on page 3-34.

3.2 Input sections, output sections, regions, and program segments on page 3-36.

3.3 Load view and execution view of an image on page 3-37.

7.1 About scatter-loading on page 7-132.

Related references

8.16 Syntax of an input section description on page 8-221.

9.48 --first=section_id on page 9-293.

9.63 --last=section_id on page 9-311.

3.13 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.

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

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

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

Related concepts

7.33 Creation of regions on page boundaries on page 7-179.

8.24 Example of aligning a base address in execution space but still tightly packed in load space on page 8-234.

Related references

9.65 --legacyalign, --no_legacyalign on page 9-313.

8.5 Load region attributes on page 8-205.

8.8 Execution region attributes on page 8-210.

3.14 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.

Note

ELF files produced with the `--sysv` option are already demand-paged compliant. `--arm_linux` also implies `--sysv`.

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.

Examples

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

7.33 Creation of regions on page boundaries on page 7-179.

7.1 About scatter-loading on page 7-132.

Related references

9.100 --scatter=filename on page 9-349.

8.26 GetPageSize() function on page 8-236.
8.27 SizeOfHeaders() function on page 8-237.
9.84 --paged on page 9-332.
9.85 --pagesize=pagesize on page 9-333.

3.15 Linker reordering of execution regions containing Thumb code

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

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

The Thumb 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

Default section placement by the linker.

3.16 Overview of veneers on page 3-57.

Related references

9.62 --largeregions, --no_largeregions on page 9-310.

3.16 Overview of veneers

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

The `BL` instruction is PC-relative and has a limited branch range. Therefore, `armlink` must generate veneers when a branch involves a destination beyond the branching range of the `BL` instruction.

The range of a `BL` instruction is 32MB for ARM 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.

A veneer extends the range of a 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 ARM and Thumb code, the veneer also handles instruction set state change.

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.

If you are using ARMv4T, `armlink` generates veneers when a branch involves change of state between ARM and Thumb®. You still get interworking veneers for ARMv5TE and later when using conditional branches, because there is no conditional `BL` instruction. Veneers for state changes are also required for `B` instructions in ARMv5 and later.

Note

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

Related concepts

3.17 Veneer sharing on page 3-58.
3.18 Veneer types on page 3-59.
3.19 Generation of position independent to absolute veneers on page 3-60.
3.20 Reuse of veneers when scatter-loading on page 3-61.

3.17 Veneer sharing

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

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

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

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

Related concepts

3.16 Overview of veneers on page 3-57.

7.1 About scatter-loading on page 7-132.

Related references

8 Scatter File Syntax on page 8-199.

9.130 --veneershare, --no_veneershare on page 9-380.

3.18 Veneer types

Veneers have different capabilities and use different code pieces.

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

- Inline veneer:
  - Performs only a state change.
  - The veneer must be inserted just before the target section to be in range.
  - An ARM-Thumb interworking veneer has a range of 256 bytes so the function entry point must appear within 256 bytes of the veneer.
  - A Thumb-ARM 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 Thumb to ARM short branch veneer has a range of 32MB, the range for an ARM instruction.
  - A short branch veneer is always position-independent.
  - A Range Extension Thumb to Thumb short branch veneer for processors that support Thumb-2 technology.
- Long branch veneer:
  - Can branch anywhere in the 4GB address space.
  - All long branch veneers are also interworking veneers.
  - There are different long branch veneers for absolute or position-independent code.

When you are using veneers be aware of the following:

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

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

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

Related concepts

3.16 Overview of veneers on page 3-57.

Related references

9.77 --max_veneer_passes=value on page 9-325.
9.58 --inlineveneer, --no_inlineveneer on page 9-306.

3.19 Generation of position independent to absolute veneers

Calling from position independent code to absolute code requries a veneer.

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

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

Related concepts

3.16 Overview of veneers on page 3-57.

Related references

9.77 --max_veneer_passes=value on page 9-325.
9.87 --piveneer, --no_piveneer on page 9-335.

3.20 Reuse of veneers when scatter-loading

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

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

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

If it is not possible to reuse a veneer, new veneers are created instead. Unless you have instructed the linker to place veneers somewhere specific using scatter-loading, a veneer is 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

3.16 Overview of veneers on page 3-57.

Related references

8.9 Address attributes for load and execution regions on page 8-213.
8.5 Load region attributes on page 8-205.

3.21 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.

The linker can create exception tables for legacy objects that contain debug frame information. The linker can do this safely for C and assembly language objects. By default, the linker does not create exception tables. This is the same as using the linker option `--exceptions_tables=nocreate`.

The linker option `--exceptions_tables=unwind` enables the linker to use the `.debug_frame` information to create a register-restoring unwinding table for each section in your image that does not already have an exception table. If this is not possible, the linker creates a nounwind table instead.

Use the linker option `--exceptions_tables=cantunwind` to create a nounwind table for each section in your image that does not already have an exception table.

Note

Be aware of the following:

- With the default settings, that is, `--exceptions --exceptions_tables=nocreate`, it is not safe to throw an exception through C or assembly code, unless the C code is compiled with the option `--exceptions`.
- The linker can generate frame unwinding instructions from objects with `.debug_frame` information. Frame unwinding is sufficient for C and assembler code. It is not sufficient for C++ code, because it does not call the destructors for the objects on the stack that is being unwound.

  The cleanup code for C++ must be generated by the compiler with the `--exceptions` option.

Related references

9.32 --diag_warning=tag[,tag,...] on page 9-277.

9.41 --exceptions, --no_exceptions on page 9-286.

9.42 --exceptions_tables=action on page 9-287.

Related information

--exceptions, --no_exceptions compiler option.

3.22 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 references to 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

A function definition, or an exported label in assembler, can also be marked as weak, as can a variable definition. In this case, a weak symbol definition is created in the object file.

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

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 `--split_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()`.
- The linker feedback mechanism, `--feedback`, records `init_foo()` as being unused during the link step. This causes the compiler to place `init_foo()` into its own section during subsequent compilations, so that it can be removed.
- The compiler directive `#pragma arm section` also instructs the compiler to generate a separate ELF section for some functions.

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:

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

Example of a weak definition

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

Related concepts

3.23 How the linker performs library searching, selection, and scanning on page 3-66.
3.26 How the linker resolves references on page 3-70.

Related references

9.43 --feedback=filename on page 9-288.
9.60 --keep=section_id on page 9-308.
9.93 --remove, --no_remove on page 9-342.

Related information

--split_sections compiler option.
__weak.
__attribute__((weak)) function attribute.
__attribute__((weak)) variable attribute.
EXPORT or GLOBAL.
IMPORT and EXTERN.
__attribute__((weakref("target"))) variable attribute.
#pragma arm section [section_type_list].
NOP.
B.
ENTRY.
EXPORT or GLOBAL.

3.23 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

3.22 Weak references and definitions on page 3-63.

3.24 How the linker searches for the ARM standard libraries on page 3-67.

Related references

9.60 --keep=section_id on page 9-308.

9.93 --remove, --no_remove on page 9-342.

9.99 --scanlib, --no_scanlib on page 9-348.

9.61 --keep_protected_symbols on page 9-309.

3.24 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. You can also control how the linker searches for the ARM standard libraries with either the `--libpath` command-line option or the `ARMLIB` or `ARMCC5LIB` environment variables.

Some libraries are stored in subdirectories. If the compiler requires a library from a particular subdirectory, it adds an import of a special symbol to identify the subdirectory to the linker. The names of subdirectories are placed in each compiled object by using a symbol of the form `Lib$$Request$$`*`sub_dir_name`*.

The `--libpath` command-line option

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

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

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

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

```
mylibs
mylibs\armlib
```

Note

When the linker command-line option `--libpath` is used, any paths specified by the `ARMCC5LIB` variable are not searched.

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.

The `ARMCC5LIB` or `ARMLIB` environment variable

You can use either of the `ARMLIB` or `ARMCC5LIB` environment variables to specify a library path.

The linker searches subdirectories given by the symbol `Lib$$Request$$`*`sub_dir_name`*, if you include the path separator character on the end of the path specified in `ARMCC5LIB`:

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

For example, if `ARMCC5LIB` is set to *`install_directory`*`\lib\`, the linker searches the directories:

```
lib
lib\armlib
lib\cpplib
```

Library search order

The linker searches for libraries in the following order:

1. Relative to the current path.
2. At the location specified with the command-line option `--libpath`.
3. At the location specified in `ARMCC5LIB`.

4. At the location specified in `ARMLIB`.
5. At the location specified in `../lib`.

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.

Related concepts

3.23 How the linker performs library searching, selection, and scanning on page 3-66.

Related references

9.66 --libpath=pathlist on page 9-314.

Related information

C and C++ library naming conventions.

The C and C++ libraries.

Toolchain environment variables.

3.25 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.

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 either the linker command-line option `--libpath` or the `ARMLIB` or `ARMCC5LIB` environment variables are not searched for user libraries.

Related concepts

3.24 How the linker searches for the ARM standard libraries on page 3-67.

Related references

9.66 --libpath=pathlist on page 9-314.
9.126 --userlibpath=pathlist on page 9-376.

Related information

The C and C++ libraries.
Toolchain environment variables.

3.26 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. List of system libraries found in `../lib`, or the directories specified by `--libpath`, `ARMCC5LIB`, or `ARMLIB`. These might also be specified by the `-J`*`dir`*`[,`*`dir`*`,…]` compiler option.
2. The list of all other files that have been loaded. These might be specified by the `-I`*`dir`*`[,`*`dir`*`,…]` compiler option.

Each list is scanned using the following process:

1. Search all specified directories to select the most compatible library variants.
2. Add the variants to the list of libraries.
3. Scan each of the libraries to load the required members:
   a. For each currently unsatisfied non-weak reference, search sequentially through the list of libraries for a matching definition. The first definition found is marked for processing in step *3.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 *3.a*. As each member is loaded it might satisfy some unresolved references, possibly including weak ones. Loading a library member might also create new unresolved weak and non-weak references.
   c. Repeat these stages until all non-weak references are either resolved or cannot be resolved by any library.
4. If any non-weak reference remains unsatisfied at the end of the scanning operation, generate an error message.

Related concepts

3.22 Weak references and definitions on page 3-63.

3.23 How the linker performs library searching, selection, and scanning on page 3-66.

3.24 How the linker searches for the ARM standard libraries on page 3-67.

Related tasks

3.25 Specifying user libraries when linking on page 3-69.

Related references

9.66 --libpath=pathlist on page 9-314.

Related information

Toolchain environment variables.

-Idir[,dir,...] compiler option.

-Jdir[,dir,...] compiler option.

List of the armlink error and warning messages.

3.27 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 family of options are:

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

Related references

9.109 --strict on page 9-359.
9.110 --strict_enum_size, --no_strict_enum_size on page 9-360.
9.111 --strict_flags, --no_strict_flags on page 9-361.
9.112 --strict_ph, --no_strict_ph on page 9-362.
9.113 --strict_relocations, --no_strict_relocations on page 9-363.
9.114 --strict_symbols, --no_strict_symbols on page 9-364.
9.115 --strict_visibility, --no_strict_visibility on page 9-365.
9.116 --strict_wchar_size, --no_strict_wchar_size on page 9-366.

3.28 Avoiding the BLX (immediate) instruction issue on an ARM1176JZ-S or ARM1176JZF-S processor

The ARM Linker can work around the possible issue on an ARM1176JZ-S or ARM1176JZF-S processor, where a `BLX` (immediate) instruction might corrupt the instruction stream.

If your software is likely to run on an ARM1176JZ-S or ARM1176JZF-S processor, see the *ARM1176JZ-S™ and ARM1176JZF-S™ Programmers Advice Notice Use of BLX (immediate)* for more details.

If you decide to apply the workaround, you must use the linker option `--no_blx_thumb_arm`.

Related references

9.10 --blx_arm_thumb, --no_blx_arm_thumb on page 9-252.

Related information

ARM1176JZ-S and ARM1176JZF-S Programmers Advice Notice Use of BLX (immediate) (ARM UAN 0002).

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 debug sections

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

In DWARF 2, the compiler and assembler generate one set of debug sections for each source file that contributes to a compilation unit. `armlink` can detect multiple copies of a debug section for a particular source file and discard all but one copy in the final image. This can result in a considerable reduction in image debug size. DWARF 3, common debug sections are placed in common groups. `armlink` discards all but one copy of each group with the same signature.

Related concepts

4.2 Elimination of common groups or sections on page 4-75.

4.3 Elimination of unused sections on page 4-76.

4.4 Elimination of unused virtual functions on page 4-78.

3.2 Input sections, output sections, regions, and program segments on page 3-36.

Related information

--debug, --no_debug compiler option.

--debug assembler option.

The DWARF Debugging Standard web site.

4.2 Elimination of common groups or sections

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

The ARM® compiler generates complete objects for linking. Therefore:

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

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

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

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

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

You control this optimization with the following linker options:

- Use the `--bestdebug` option to use the largest common data (COMDAT) group (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.

  Because `--no_bestdebug` is the default, the final image is the same regardless of whether you generate debug tables during compilation with `--debug`.

Related concepts

4.1 Elimination of common debug sections on page 4-74.

4.3 Elimination of unused sections on page 4-76.

4.4 Elimination of unused virtual functions on page 4-78.

3.2 Input sections, output sections, regions, and program segments on page 3-36.

Related references

9.9 --bestdebug, --no_bestdebug on page 9-251.

Related information

Inline functions.

--debug, --no_debug compiler option.

4.3 Elimination of unused sections

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

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

Note

By default, unused section elimination is disabled if you are building DLLs with `--dll`, or shared libraries with `--shared`. Therefore, you must explicitly include `--remove` to re-enable unused section elimination.

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

Note

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

You can mark a function or variable in source code with the `__attribute__((used))` attribute. This causes `armcc` to generate the symbol `__tagsym$$used` for each of the functions and variables. A section containing a definition of `__tagsym$$used` is not removed by unused section elimination.

You can also use the `--split_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 debug sections on page 4-74.
4.2 Elimination of common groups or sections on page 4-75.
4.4 Elimination of unused virtual functions on page 4-78.
3.2 Input sections, output sections, regions, and program segments on page 3-36.
3.22 Weak references and definitions on page 3-63.

Related references

9.39 --entry=location on page 9-284.
9.48 --first=section_id on page 9-293.
9.60 --keep=section_id on page 9-308.

9.63 --last=section_id on page 9-311.
9.53 --info=topic[,topic,...] on page 9-299.
9.93 --remove, --no_remove on page 9-342.

Related information

--split_sections compiler option.
__attribute__((used)) function attribute.
__attribute__((used)) variable attribute.

4.4 Elimination of unused virtual functions

Unused virtual function elimination is a refinement of unused section elimination.

Unused section elimination efficiently removes unused functions from C code. In C++ applications, virtual functions and *RunTime Type Information* (RTTI) objects are referenced by pointer tables, known as vtables. Without extra information, the linker cannot determine which vtable entries are accessed at runtime. This means that the standard unused section elimination algorithm used by the linker cannot guarantee to remove unused virtual functions and RTTI objects. *Virtual Function Elimination* (VFE) is a refinement of unused section elimination to reduce ROM size in images generated from C++ code. You can use this optimization to eliminate unused virtual functions and RTTI objects from your code.

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

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

Note

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

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

Related concepts

4.1 Elimination of common debug sections on page 4-74.

4.2 Elimination of common groups or sections on page 4-75.

4.3 Elimination of unused sections on page 4-76.

Related references

9.133 --vfemode=mode on page 9-383.

Related information

--rtti, --no_rtti compiler option.

4.5 About linker feedback

Linker feedback is a collaboration between the compiler and linker that can increase the amount of unused code that can be removed from an ELF image.

The feedback option produces a text file containing a list of unused functions, and functions that have been inlined by the linker. This information can be fed back to the compiler, which can rebuild the objects, placing these functions in their own sections. These sections can then be removed by the linker during usual unused section elimination.

The feedback file has the following format:

```
;#<FEEDBACK># ARM Linker, N.nn [Build num]: Last Updated: day mmm dd hh:mm:ss yyyy
;VERSION 0.2
;FILE filename.o
unused_function <= USED 0
inlined_function <= LINKER_INLINED
…
```

The feedback file contains an entry for each object file. Each entry contains:

- The object filename specified as a comment:

  `;FILE filename.o`
- A list of the functions in that file that are not used:

  `unused_function <= USED 0`
- A list of the functions in that file that are inlined by the linker:

  `inlined_function <= LINKER_INLINED`

To use linker feedback, specify `--feedback file` on the linker and compiler command lines.

Note

The compiler issues a warning message if no feedback file exists. Therefore, you might want to leave the `--feedback file` option off the first invocation of the compiler.

Additional feedback options are provided by the linker:

- If you are using scatter-loading then an executable ELF image cannot be created if your code does not fit into the region limits described in your scatter file. In this case use the `--feedback_image=option` command-line option.
- To control the information that the linker puts into the feedback file, use the `--feedback_type=type` command-line option. You can control whether or not to list functions that require interworking or unused functions.

Related concepts

4.12 Function inlining with the linker on page 4-88.

Related tasks

4.6 Example of using linker feedback on page 4-81.

Related references

7 Scatter-loading Features on page 7-130.
9.56 --inline, --no_inline on page 9-304.
9.100 --scatter=filename on page 9-349.
9.43 --feedback=filename on page 9-288.
9.44 --feedback_image=option on page 9-289.
9.45 --feedback_type=type on page 9-290.

Related information

--feedback=filename compiler option.

4.6 Example of using linker feedback

This is an example to show how linker feedback works.

Procedure

1. Create a file `fb.c` containing the code shown in this example:

```
#include <stdio.h>
void legacy()
{
    printf("This is a legacy function that is no longer used.\n");
}
int cubed(int i)
{
    return i*i*i;
}
void main(void)
{
    int n = 3;
    printf("%d cubed = %d\n",n,cubed(n));
}
```

2. Compile the program, and ignore the warning that the feedback file does not exist:

```
armcc --asm -c --feedback fb.txt fb.c
```

   This inlines the `cubed()` function by default, and creates an assembler file `fb.s` and an object file `fb.o`. In the assembler file, the code for `legacy()` and `cubed()` is still present. Because of the inlining, there is no call to `cubed()` from `main`.

   An out-of-line copy of `cubed()` is kept because it is not declared as **`static`**.

3. Link the object file to create the linker feedback file with the command line:

```
armlink --info sizes --list fbout1.txt --feedback fb.txt fb.o -o fb.axf
```

   Linker diagnostics are output to the file `fbout1.txt`.

   The linker feedback file identifies the source file that contains the unused functions in a comment (not used by the compiler) and includes entries for the `legacy()` and `cubed()` functions:

```
;#<FEEDBACK># ARM Linker, 5.01 [Build num]: Last Updated: Date
;VERSION 0.2
;FILE fb.o
cubed <= USED 0
legacy <= USED 0
```

   This shows that the functions are not used.

4. Repeat the compile and link stages with a different diagnostics file:

```
armcc --asm -c --feedback fb.txt fb.c
```

```
armlink --info sizes --list fbout2.txt fb.o -o fb.axf
```

5. Compare the two diagnostics files, `fbout1.txt` and `fbout2.txt`, to see the sizes of the image components (for example, Code, RO Data, RW Data, and ZI Data). The Code component is smaller.

   In the assembler file, `fb.s`, the `legacy()` and `cubed()` functions are no longer in the same area as the main() function. They are compiled into their own ELF sections. Therefore, `armlink` can remove the `legacy()` and `cubed()` functions from the final image.

Note

To get the maximum benefit from linker feedback you have to do a full compile and link at least twice. However, a single compile and link using feedback from a previous build is usually sufficient.

Related concepts

4.5 About linker feedback on page 4-79.

Related references

9.43 --feedback=filename on page 9-288.
9.44 --feedback_image=option on page 9-289.
9.45 --feedback_type=type on page 9-290.
9.53 --info=topic[,topic,...] on page 9-299.
9.69 --list=filename on page 9-317.
9.100 --scatter=filename on page 9-349.

Related information

--asm compiler option.
-c compiler option.
--feedback=filename compiler option.
--inline, --no_inline compiler option.

4.7 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.

Related concepts

4.10 How compression is applied on page 4-86.
4.9 Options available to override the compression algorithm used by the linker on page 4-85.
4.11 Considerations when working with RW data compression on page 4-87.
4.8 How the linker chooses a compressor on page 4-84.

4.8 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

4.7 Optimization with RW data compression on page 4-83.
4.10 How compression is applied on page 4-86.
4.11 Considerations when working with RW data compression on page 4-87.
4.9 Options available to override the compression algorithm used by the linker on page 4-85.
4.10 How compression is applied on page 4-86.

4.9 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 ARM code, then ARM decompressors are used automatically. If the image contains any Thumb code, Thumb 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

4.7 Optimization with RW data compression on page 4-83.

4.10 How compression is applied on page 4-86.

4.8 How the linker chooses a compressor on page 4-84.

4.11 Considerations when working with RW data compression on page 4-87.

Related references

9.26 --datacompressor=opt on page 9-271.

4.10 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.

Limpel-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

4.7 Optimization with RW data compression on page 4-83.

4.8 How the linker chooses a compressor on page 4-84.

4.9 Options available to override the compression algorithm used by the linker on page 4-85.

4.8 How the linker chooses a compressor on page 4-84.

Related references

9.26 --datacompressor=opt on page 9-271.

4.11 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

4.7 Optimization with RW data compression on page 4-83.

4.8 How the linker chooses a compressor on page 4-84.

4.9 Options available to override the compression algorithm used by the linker on page 4-85.

4.10 How compression is applied on page 4-86.

Related references

6.5 Load$$ execution region symbols on page 6-108.

7 Scatter-loading Features on page 7-130.

9.74 --map, --no_map on page 9-322.

8 Scatter File Syntax on page 8-199.

4.12 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.

The following options are available to control function inlining:

- `--inline` and `--no_inline` command-line options allow you to control branch inlining. However, `--no_inline` only turns off inlining for user-supplied objects. The linker still inlines functions from the ARM C Library by default.
- `--inline_type=`*`type`* command-line option gives you more control over inlining. You can also inline functions from the ARM C Library, and turn off inlining completely. This option overrides `--inline` if both are present on the command-line.

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

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

Note

The linker can inline two 16-bit encoded Thumb instructions in place of the 32-bit encoded Thumb `BL` instruction.

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

Related concepts

4.13 Factors that influence function inlining on page 4-89.

4.3 Elimination of unused sections on page 4-76.

Related references

9.57 --inline_type=type on page 9-305.

9.53 --info=topic[,topic,...] on page 9-299.

9.56 --inline, --no_inline on page 9-304.

4.13 Factors that influence function inlining

There are a number of factors that influence 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 ARM and Thumb 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 Thumb instructions. However, an ARM calling function can only inline functions containing a single 16-bit encoded Thumb instruction or 32-bit encoded Thumb 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
ARM	ARM	4 to 8 bytes
ARM	Thumb	2 to 6 bytes
Thumb	Thumb	2 to 6 bytes

  The linker can inline in different states if there is an equivalent instruction available. For example, if a Thumb instruction is `adds r0, r0` then the linker can inline the equivalent ARM instruction. It is not possible to inline from ARM to Thumb because there is less chance of Thumb equivalent to an ARM 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 ARM instruction can only be inlined if either:
  - The condition on the `BL` matches the condition on the instruction being inlined. For example, `BLEQ` can only inline an instruction with a matching condition like `ADDEQ`.
  - The `BL` instruction or the instruction to be inlined is unconditional. An unconditional ARM `BL` can inline any conditional or unconditional instruction that satisfies all the other criteria. An instruction that cannot be conditionally executed cannot be inlined if the `BL` instruction is conditional.
- A `BL` that is the last instruction of a Thumb *If-Then* (IT) block cannot inline a 16-bit encoded Thumb 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

4.14 About branches that optimize to a NOP on page 4-91.

Related information

Conditional instructions.

ADD.

B.

CPS.
IT.
MOV.
MRS (PSR to general-purpose register).
MSR (general-purpose register to PSR).

4.14 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

4.15 Linker reordering of tail calling sections on page 4-92.

Related references

9.12 --branchnop, --no_branchnop on page 9-254.

4.15 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.

Related concepts

4.14 About branches that optimize to a NOP on page 4-91.

4.16 Restrictions on reordering of tail calling sections on page 4-93.

3.18 Veneer types on page 3-59.

Related references

9.53 --info=topic[,topic,...] on page 9-299.

9.119 --tailreorder, --no_tailreorder on page 9-369.

4.16 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

4.15 Linker reordering of tail calling sections on page 4-92.

4.17 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 `.common` section while retaining all useful information.

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

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

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

Note

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

Related references

9.20 --comment_section, --no_comment_section on page 9-263.
9.46 --filtercomment, --no_filtercomment on page 9-291.

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

3.13 Section alignment with the linker on page 3-53.
4.7 Optimization with RW data compression on page 4-83.

Related tasks

5.2 Identifying the source of some link errors on page 5-97.
5.3 Example of using the --info linker option on page 5-98.

Related references

9.69 --list=filename on page 9-317.
9.74 --map, --no_map on page 9-322.
9.102 --show_cmdline on page 9-351.
9.117 --symbols, --no_symbols on page 9-367.
9.131 --verbose on page 9-381.
9.137 --xref, --no_xref on page 9-387.
9.138 --xrefdbg, --no_xrefdbg on page 9-388.
9.53 --info=topic[,topic,...] on page 9-299.

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 references

5.1 Options for getting information about linker-generated files on page 5-96.
9.53 --info=topic[,topic,...] on page 9-299.
9.131 --verbose on page 9-381.

5.3 Example of using the --info linker option

This is an example of the output generated by the `--info` option

To display the component sizes when linking enter:

```
armlink --info sizes …
```

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

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

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

In this example:

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

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

RW Data
: Shows how many bytes are occupied by RW data.

ZI Data
: Shows how many bytes are occupied by ZI data.

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

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

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

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

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

(incl. Padding)

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

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

Grand Totals

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

ELF Image Totals

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

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

ROM Totals

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

Related references

5.1 Options for getting information about linker-generated files on page 5-96.
9.53 --info=topic[,topic,...] on page 9-299.

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.

To find where a symbol is placed when linking, use the `--keep=`*`section_id`* and `--symbols` options. For example, if *`object`*`(`*`section`*`)` is the section containing the symbol, enter:

```
armlink --keep=object(section) --symbols s.o --output=s.axf
```

Note

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

Examples

Do the following:

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

```
long long altstack[10] __attribute__ ((section ("STACK"), zero_init));
int main(void)
{
    return sizeof(altstack);
}
```

2. Compile the source:

```
armcc -c s.c -o s.o
```

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

```
armlink --keep=s.o(STACK) --symbols s.o --output=s.axf
```

4. Locate the `STACK` symbol in the output, for example:

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

Image Symbol Table

    Local Symbols

    Symbol Name                          Value     Ov Type        Size  Object(Section)
…
    STACK                                0x00008200   Section       80  s.o(STACK)

    Global Symbols

    Symbol Name                          Value     Ov Type        Size  Object(Section)
…
    altstack                             0x00008200   Data          80  s.o(STACK)
    Image$$ZI$$Limit                     0x00008250   Number         0  s.o(STACK)
```

This shows that the stack is placed in the ZI execution region.

5.5 How to find the location of a symbol within the map file

To find the location of a symbol within the map file you must find the section that defines the symbol, and ensure that the linker has not removed the section.

To find the location of a symbol within the map file, use the `--keep=section_id` and `--map` options to view the image memory map. For example, if `object(section)` is the section containing the symbol, enter:

```
armlink --keep=object(section) --map s.o --output=s.axf
```

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

Examples

Do the following:

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

```
long long altstack[10] __attribute__ ((section ("STACK"), zero_init));
int main(void)
{
    return sizeof(altstack);
}
```

2. Compile the source:

```
armcc -c s.c -o s.o
```

3. Link the object `s.o`, keeping the `STACK` symbol and displaying the image memory map:

```
armlink --keep=s.o(STACK) --map s.o --output=s.axf
```

4. Locate the `STACK` symbol in the output, for example:

```
…
Execution Region ER_RW (…)
**** No section assigned to this execution region ****
Execution Region ER_ZI (…)
Base Addr    Size         Type   Attr      Idx    E Section Name        Object
…
0x00008228   0x00000050   Zero   RW            2    STACK               s.o
```

This shows that the stack is placed in execution region ER_ZI.

Related references

9.60 --keep=section_id on page 9-308.
9.81 --output=filename on page 9-329.
9.74 --map, --no_map on page 9-322.

Related information

Using fromelf to find where a symbol is placed in an executable ELF image.
-c compiler option.
-o filename 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 inline transitions between code and data at literal pool boundaries, and between ARM code and Thumb code, such as ARM/Thumb interworking veneers.

The mapping symbols are:

$a
Start of a sequence of ARM instructions.

$t
Start of a sequence of Thumb instructions.

$t.x
Start of a sequence of ThumbEE instructions.

$d
Start of a sequence of data items, such as a literal pool.

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

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

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

Note

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

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

Related references

9.70 --list_mapping_symbols, --no_list_mapping_symbols on page 9-318.
9.114 --strict_symbols, --no_strict_symbols on page 9-364.

Related information

Symbol naming rules.
--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.

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

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

Be aware that:

- If you use the `--strict` compiler command-line option, the compiler does not accept symbol names containing dollar symbols. To re-enable support, include the `--dollar` option on the compiler command line.
- 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.

Related concepts

6.9 Methods of importing linker-defined symbols in C and C++ on page 6-113.
6.10 Methods of importing linker-defined symbols in ARM assembly language on page 6-114.

Related information

--dollar, --no_dollar compiler option.
--strict, --no_strict compiler option.

6.3 Region-related symbols

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

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

- `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

6.7 Region name values when not scatter-loading on page 6-111.

Related references

6.4 Image$$ execution region symbols on page 6-107.
6.5 Load$$ execution region symbols on page 6-108.
6.6 Load$$LR$$ load region symbols on page 6-110.

6.4 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$$Length`	Length of the RW output section in bytes.
`Image$$`*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

6.9 Methods of importing linker-defined symbols in C and C++ on page 6-113.
6.10 Methods of importing linker-defined symbols in ARM assembly language on page 6-114.
6.7 Region name values when not scatter-loading on page 6-111.

Related references

6.3 Region-related symbols on page 6-106.

6.5 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 `Load$$` execution region present in the image. All the symbols refer to load addresses after 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.
`Load$$`*region_name*`$$ZI$$Length`	Load length of the ZI output section in bytes. The Load Length of ZI is zero unless *region_name* has the `ZEROPAD` scatter-loading keyword set. If `ZEROPAD` is set then: Load Length = `Image$$`*region_name*`$$ZI$$Length`
`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 Initialized data that is written to the file is taken into account by the Limit and Length values. Zero Initialized data is written into the file when the `ZEROPAD` scatter-loading keyword is used.

Related concepts

6.7 Region name values when not scatter-loading on page 6-111.

4.7 Optimization with RW data compression on page 4-83.

Related references

6.3 Region-related symbols on page 6-106.

6.4 Image$$ execution region symbols on page 6-107.

6.6 Load$$LR$$ load region symbols on page 6-110.

8.8 Execution region attributes on page 8-210.

6.6 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$$LR$$` 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

3.1 The structure of an ARM ELF image on page 3-34.

3.2 Input sections, output sections, regions, and program segments on page 3-36.

3.3 Load view and execution view of an image on page 3-37.

Related references

6.3 Region-related symbols on page 6-106.

6.7 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`.

Note

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

Related concepts

6.11 Section-related symbols on page 6-115.

Related references

6.3 Region-related symbols on page 6-106.

6.4 Image$$ execution region symbols on page 6-107.

6.5 Load$$ execution region symbols on page 6-108.

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

Related references

7 Scatter-loading Features on page 7-130.
9.100 --scatter=filename on page 9-349.

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

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

Import by value

```
extern unsigned int symbol_name;
```

Import by reference

```
extern void *symbol_name;
```

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

Importing a linker-defined symbol

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

Importing symbols that define a ZI output section

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

Related references

6.4 Image$$ execution region symbols on page 6-107.

6.10 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 `IMPORT` directive and create a 32-bit data word to hold the value of the symbol, for example:

```
        IMPORT |Image$$ZI$$Limit|
…
zi_limit DCD |Image$$ZI$$Limit|
```

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

```
        LDR r1, zi_limit
```

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

Related references

6.4 Image$$ execution region symbols on page 6-107.

Related information

ARM and Thumb Instructions.

IMPORT and EXTERN.

6.11 Section-related symbols

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

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

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

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

Related references

6.12 Image symbols on page 6-116.
6.13 Input section symbols on page 6-117.

6.12 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.

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

3.5 Types of simple image on page 3-41.

3.22 Weak references and definitions on page 3-63.

Related tasks

7.8 Specifying stack and heap using the scatter file on page 7-141.

Related references

7.7 Linker-defined symbols that are not defined when scatter-loading on page 7-140.

Related information

Stack use in C and C++.

__user_setup_stackheap().

6.13 Input section symbols

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

The following table shows the input section symbols:

Table 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

3.2 Input sections, output sections, regions, and program segments on page 3-36.

Related references

7 Scatter-loading Features on page 7-130.

6.14 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.

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.

You specify a symdefs file with the `--symdefs` command-line option.

Related tasks

6.15 Creating a symdefs file on page 6-119.
6.17 Reading a symdefs file on page 6-121.

Related references

6.18 Symdefs file format on page 6-122.

6.15 Creating a symdefs file

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

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 concepts

6.14 Access symbols in another image on page 6-118.

Related references

6.18 Symdefs file format on page 6-122.
9.118 --symdefs=filename on page 9-368.

6.16 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 now use the symdefs file to link additional applications.

Related concepts

6.14 Access symbols in another image on page 6-118.

Related references

6.18 Symdefs file format on page 6-122.
9.118 --symdefs=filename on page 9-368.

6.17 Reading a symdefs file

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

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

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

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

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

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

Note

The same function name or symbol name cannot be defined in both ARM code and in Thumb code.

Related references

6.18 Symdefs file format on page 6-122.

6.18 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, N.nn [Build num]:`
- Date and time of the most recent update of the symdefs file, in the format:

  `Last Updated: Date`

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

Comments

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

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

Symbol information

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

Symbol value

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

Type flag

A single letter to show symbol type:

A
ARM code

T
Thumb code

D
Data

N
Number.

Symbol name

The symbol name.

Examples

This example shows a typical symdefs file format:

```
#<SYMDEFS># ARM Linker, 5.04 [Build num]: Last Updated: Date
;value type name, this is an added comment
0x00008000 A __main
0x00008004 A __scatterload
```

```
0x000080E0 T main
0x0000814D T _main_arg
0x0000814D T __argv_alloc
0x00008199 T __rt_get_argv
…
   # This is also a comment, blank lines are ignored
…
0x0000A4FC D __stdin
0x0000A540 D __stdout
0x0000A584 D __stderr
0xFFFFFFFD N __SIG_IGN
```

The example shows that the first line has wrapped, but it is not wrapped in the file.

Related concepts

6.14 Access symbols in another image on page 6-118.

Related tasks

6.15 Creating a symdefs file on page 6-119.

6.16 Outputting a subset of the global symbols on page 6-120.

6.17 Reading a symdefs file on page 6-121.

6.19 What is a steering file?

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

Steering file commands enable you to:

- Manage symbols in the symbol table.
- Control the copying of symbols from the static symbol table to the dynamic symbol table.
- Store information about the libraries that a link unit depends on.

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

Related tasks

6.20 Specifying steering files on the linker command-line on page 6-125.

Related references

6.21 Steering file command summary on page 6-126.
6.22 Steering file format on page 6-127.
9.34 --edit=file_list on page 9-279.

6.20 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 concepts

6.19 What is a steering file? on page 6-124.

Related references

6.21 Steering file command summary on page 6-126.

6.22 Steering file format on page 6-127.

6.21 Steering file command summary

A summary of the commands you can use in a streering file.

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 concepts

6.19 What is a steering file? on page 6-124.

Related tasks

6.20 Specifying steering files on the linker command-line on page 6-125.

Related references

6.22 Steering file format on page 6-127.
10.1 EXPORT steering file command on page 10-392.
10.2 HIDE steering file command on page 10-393.
10.3 IMPORT steering file command on page 10-394.
10.4 RENAME steering file command on page 10-395.
10.5 REQUIRE steering file command on page 10-396.
10.6 RESOLVE steering file command on page 10-397.
10.7 SHOW steering file command on page 10-399.

6.22 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 concepts

6.19 What is a steering file? on page 6-124.

Related tasks

6.20 Specifying steering files on the linker command-line on page 6-125.

Related references

6.21 Steering file command summary on page 6-126.
10.1 EXPORT steering file command on page 10-392.
10.2 HIDE steering file command on page 10-393.
10.3 IMPORT steering file command on page 10-394.
10.4 RENAME steering file command on page 10-395.
10.5 REQUIRE steering file command on page 10-396.
10.6 RESOLVE steering file command on page 10-397.
10.7 SHOW steering file command on page 10-399.

6.23 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

6.19 What is a steering file? on page 6-124.

Related tasks

6.20 Specifying steering files on the linker command-line on page 6-125.

Related references

6.21 Steering file command summary on page 6-126.
6.18 Symdefs file format on page 6-122.
10.2 HIDE steering file command on page 10-393.
10.4 RENAME steering file command on page 10-395.
9.34 --edit=file_list on page 9-279.

6.24 Use of $Super$$ and $Sub$$ to patch symbol definitions

There are special patterns you can use for situations where an existing symbol cannot be modified.

An existing symbol cannot be modified, for example, if 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.

Examples

To patch the definition of the function `foo()`:

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

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

Note

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

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

Using $Super$$ and $Sub$$

```
extern void ExtraFunc(void); extern void $Super$$foo(void):
/* this function is called instead of the original foo() */
void $Sub$$foo(void)
{
  ExtraFunc();    /* does some extra setup work */
  $Super$$foo();  /* calls the original foo() function */
                  /* To avoid calling the original foo() function
                   * omit the $Super$$foo(); function call.
                   */
}
```

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 About scatter-loading

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

Scatter-loading gives you complete control over the grouping and placement of image components. You can use scatter-loading to create simple images, but it is generally only used for images that have a complex memory map. That is, where multiple memory regions are scattered in the memory map at load and execution time.

An image memory map is made up of regions and output sections. Every region in the memory map can have a different load and execution address.

To construct the memory map of an image, the linker must have:

- Grouping information that describes how input sections are grouped into output sections and regions.
- Placement information that describes the addresses where regions are to be located in the memory maps.

When the linker creates an image using a scatter file, it creates some region-related symbols. The linker creates these special symbols only if your code references them.

Related concepts

7.2 When to use scatter-loading on page 7-133.

7.45 Scatter file to ELF mapping on page 7-197.

3.1 The structure of an ARM ELF image on page 3-34.

Related references

6.3 Region-related symbols on page 6-106.

Related information

Scatter file with link to bit-band objects.

7.2 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 About scatter-loading on page 7-132.

7.3 Scatter-loading command-line options

The command-line options to the linker give some control over the placement of data and code, but complete control of placement requires more detailed instructions than can be entered on the command line.

Complex memory maps

Placement of code and data in complex memory maps must be specified in a scatter file. You specify the scatter file with the option:

`--scatter=`*`scatter_file`*

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

Simple memory maps

For simple memory maps, you can place code and data with with the following memory map related command-line options:

- `--partial`.
- `--ro_base`.
- `--rw_base`.
- `--ropi`.
- `--rwpi`.
- `--rosplit`.
- `--split`.
- `--reloc`.
- `--startup`.
- `--xo_base`
- `--zi_base`.

Note

You cannot use `--scatter` with these options.

Related concepts

7.1 About scatter-loading on page 7-132.
7.2 When to use scatter-loading on page 7-133.
7.38 Equivalent scatter-loading descriptions for simple images on page 7-185.

Related references

9.86 --partial on page 9-334.
9.91 --reloc on page 9-340.
9.94 --ro_base=address on page 9-343.
9.95 --ropi on page 9-344.
9.96 --rosplit on page 9-345.
9.97 --rw_base=address on page 9-346.
9.98 --rwpi on page 9-347.
9.100 --scatter=filename on page 9-349.
9.107 --split on page 9-357.
9.108 --startup=symbol, --no_startup on page 9-358.
9.136 --xo_base=address on page 9-386.
9.140 --zi_base=address on page 9-390.
8 Scatter File Syntax on page 8-199.

7.4 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.

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

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

The following figure shows a simple memory map:

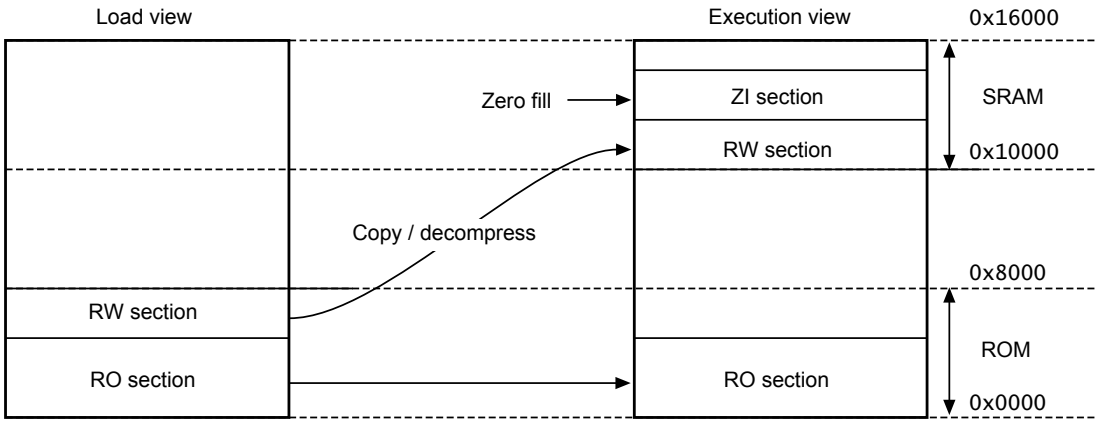

Figure 7-1 Simple scatter-loaded memory map

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

7.45 Scatter file to ELF mapping on page 7-197.
7.1 About scatter-loading on page 7-132.
7.2 When to use scatter-loading on page 7-133.

Related references

9.94 --ro_base=address on page 9-343.
9.97 --rw_base=address on page 9-346.
9.136 --xo_base=address on page 9-386.

7.5 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:

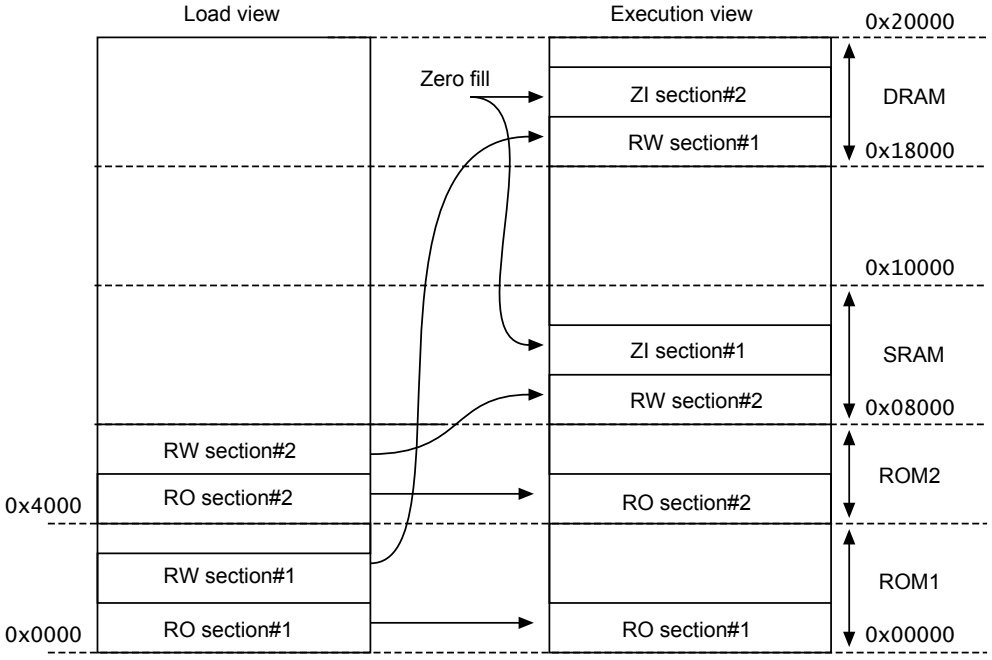

Figure 7-2 Complex memory map

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, it is advisable to use the `*` or `.ANY` specifier to place leftover code and data.

Related concepts

7.1 About scatter-loading on page 7-132.
7.10 Root execution regions and the ABSOLUTE attribute on page 7-143.
7.11 Root execution regions and the FIXED attribute on page 7-144.
8.21 Scatter files containing relative base address load regions and a ZI execution region on page 8-230.
7.45 Scatter file to ELF mapping on page 7-197.
7.2 When to use scatter-loading on page 7-133.

7.6 Scatter file with link to bit-band objects

In devices with the ARMv7-M architecture, the SRAM and Peripheral regions each have a bit-band feature.

You can access each bit in the bit-band region individually at a different address, called the bit-band alias. For example, to access bit[13] of the word at `0x20000001`, you can use the address `0x22000054`.

The following table shows the bit-band regions and aliases within the SRAM and Peripheral memory regions.

Table 7-1 ARMv7-M bit-band regions and aliases

Memory region	Description	Address range
SRAM	Bit-band region	0x20000000-0x200FFFFF
	Bit-band alias	0x22000000-0x23FFFFFF
Peripheral	Bit-band region	0x40000000-0x400FFFFF
	Bit-band alias	0x42000000-0x43FFFFFF

The following is an example scatter file that links bit-band objects.

```
FLASH_LOAD 0x20000000
{
    RW 0x20000000  ; RW data at the start of bit band region
  {
    * (+RW-DATA)
  }
  RO +0 FIXED        ; Followed by the RO Data
  {
     * (+RO-DATA)
  }
  CODEDATA +0   ; Followed by everything else
  {
    * (+RO-CODE)
    * (+ZI)             ; ZI follows straight after
  }
  ARM_LIB_HEAP  +0 EMPTY 0x10000  ; heap starts after that
  {
  }
  ARM_LIB_STACK 0x20100000 EMPTY -0x10000    ; stack starts at the
                                             ; top of bit band region
  {
  }
}
```

Related references

Scatter File Syntax.

7.7 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 6-105.

Related tasks

7.8 Specifying stack and heap using the scatter file on page 7-141.

7.8 Specifying stack and heap using the scatter file

The ARM C library provides multiple implementations of the function `__user_setup_stackheap()`, and can select the correct one for you automatically from information given in a scatter file.

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

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

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

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

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

——— **Note** ———

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

———————————

Related references

6.3 Region-related symbols on page 6-106.

Related information

__user_setup_stackheap().
Legacy function __user_initial_stackheap().

7.9 Root execution regions

A root region is a region with the same load and execution address.

The initial entry point of the image must be in a root region. If the initial entry point is not in a root region, the link fails and the linker gives an error message.

Examples

Root region with the same load and execution address.

```
LR_1 0x040000          ; load region starts at 0x40000
{                      ; start of execution region descriptions
    ER_RO 0x040000     ; load address = execution address
    {
        * (+RO)        ; all RO sections (must include section with
                       ; initial entry point)
    }
    …                ; rest of scatter-loading description
}
```

Related concepts

7.10 Root execution regions and the ABSOLUTE attribute on page 7-143.
7.27 Placement of ARM C and C++ library code on page 7-172.
7.11 Root execution regions and the FIXED attribute on page 7-144.
3.1 The structure of an ARM ELF image on page 3-34.

7.10 Root execution regions and the ABSOLUTE attribute

You can use the `ABSOLUTE` attribute to specify root execution regions.

Specify `ABSOLUTE` as the attribute for the execution region, either explicitly or by permitting it to default, and use the same address for the first execution region and the enclosing load region.

To make the execution region address the same as the load region address, either:

- Specify the same numeric value for both the base address for the execution region and the base address for the load region.
- Specify a `+0` offset for the first execution region in the load region.

  If an offset of zero (`+0`) is specified for all subsequent execution regions in the load region, then all execution regions not following an execution region containing ZI are also root regions.

Examples

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     ; load address = execution address
    {
        * (+RO)        ; all RO sections (must include section with
                       ; initial entry point)
    }
    …                ; rest of scatter-loading description
}
```

Related concepts

7.9 Root execution regions on page 7-142.

7.11 Root execution regions and the FIXED attribute on page 7-144.

8.3 Load region descriptions on page 8-203.

8.6 Execution region descriptions on page 8-207.

8.10 Considerations when using a relative address +offset for load regions on page 8-215.

8.11 Considerations when using a relative address +offset for execution regions on page 8-216.

Related references

8.5 Load region attributes on page 8-205.

8.8 Execution region attributes on page 8-210.

8.9 Address attributes for load and execution regions on page 8-213.

Related information

ENTRY directive.

7.11 Root execution regions and the FIXED attribute

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:

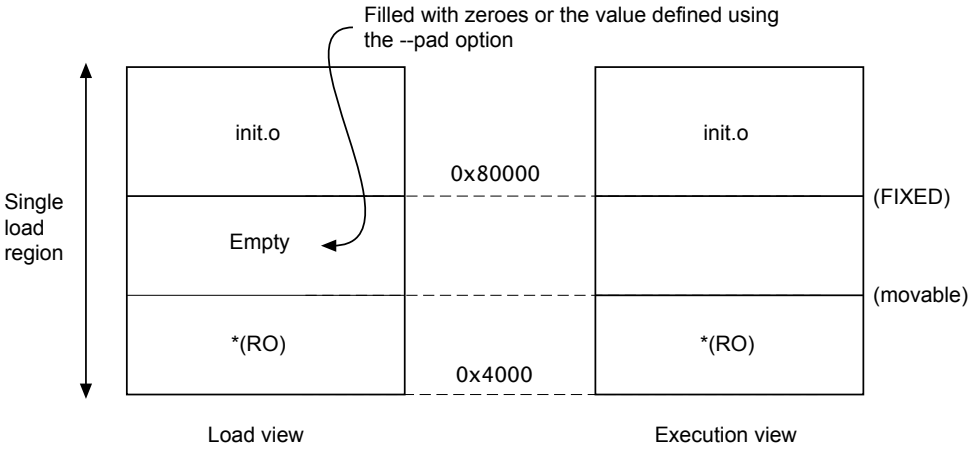

Figure 7-3 Memory map for fixed execution regions

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 to place a function or a block of data, such as a constant table or a checksum, at a fixed address in ROM so that it can be accessed easily through pointers.

If you specify, for example, that some initialization code is to be placed at start of ROM and a checksum at the end of ROM, some of the memory contents might be unused. Use the `*` or `.ANY` module selector to flood fill the region between the end of the initialization block and the start of the data block.

To make your code easier to maintain and debug, it is suggested that you use the minimum amount of placement specifications in scatter files and leave the detailed placement of functions and data to the linker.

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

8.6 Execution region descriptions on page 8-207.

Related references

8.5 Load region attributes on page 8-205.

8.8 Execution region attributes on page 8-210.

8.9 Address attributes for load and execution regions on page 8-213.

7.12 Methods of placing functions and data at specific addresses

There are various methods available to place functions and data at specific addresses.

Where they are required, the compiler normally produces RO, RW, ZI, and XO sections from a single source file. These sections contain all the code and data from the source file. To place a single function or data item at a fixed address, you must enable the linker to process the function or data separately from the rest of the input files.

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

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

To place a function or variable at a specific address it must be placed in its own section. There are several ways to do this:

- Place the function or data item in its own source file.
- Use `__attribute__((at(`*address*`)))` to place variables in a separate section at a specific address.
- Use `__attribute__((section("`*name*`")))` to place functions and variables in a named section.
- Use the `AREA` directive from assembly language. In assembly code, the smallest locatable unit is an `AREA`.
- Use the `--split_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`.

Example of how to place a variable at a specific address without scatter-loading

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);
int gSquared __attribute__((at(0x5000)));  // Place at 0x5000
int main()
{
    gSquared=sqr(3);
    printf("Value squared is: %d\n", gSquared);
}
```

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

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

3. Compile and link the sources:

```
armcc -c -g function.c
armcc -c -g 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__((at(0x5000)))` specifies that the global variable `gSquared` is to be placed at the absolute address `0x5000`. `gSquared` is placed in the execution region `ER$$.ARM.__at_0x00005000` and load region `LR$$.ARM.__at_0x00005000`.

Note

Although the address is specified as `0x5000` in the source file, the region names and section name addresses are normalized to eight hexadecimal digits.

The memory map shows:

```
…
  Load Region LR$$.ARM.__at_0x00005000 (Base: 0x00005000, Size: 0x00000000, Max: 0x00000004,
ABSOLUTE)

    Execution Region ER$$.ARM.__at_0x00005000 (Base: 0x00005000, Size: 0x00000004, Max:
0x00000004, ABSOLUTE, UNINIT)

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x00005000   0x00000004   Zero   RW           13    .ARM.__at_0x00005000  main.o
```

Example of how to place a variable in a named section with scatter-loading

To modify your source code to place code and data in a specific section using a scatter file:

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

```
#include <stdio.h>
extern int sqr(int n1);
int gSquared __attribute__((section("foo")));  // Place in section foo
int main()
{
    gSquared=sqr(3);
    printf("Value squared is: %d\n", gSquared);
}
```

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

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

3. Create the scatter file `scatter.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:

```
armcc -c -g function.c
armcc -c -g 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.

Example of how to place a variable at a specific address with scatter-loading

To modify your source code to place code and data at a specific address using a scatter file:

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

```
#include <stdio.h>
extern int sqr(int n1);
// Place at address 0x10000
const int gValue __attribute__((section(".ARM.__at_0x10000"))) = 3;
int main()
{
    int squared;
    squared=sqr(gValue);
    printf("Value squared is: %d\n", squared);
}
```

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

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

3. Create the scatter file `scatter.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:

```
armcc -c -g function.c
armcc -c -g 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: 0x00001578, Size: 0x0000ea8c, Max: 0xffffffff, ABSOLUTE)

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x00001578   0x0000000c   Code   RO            3    .text               function.o
    0x00001584   0x0000ea7c   PAD
    0x00010000   0x00000004   Data   RO           15    .ARM.__at_0x10000   main.o
…
```

In this example, the size of `ER1` is uknown. 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 to placed in a separate load region `LR$$.ARM.__at_0x10000` that contains the execution region `ER$$.ARM.__at_0x.ARM.__at_0x10000`.

Related concepts

7.20 Placement of __at sections at a specific address on page 7-163.

7.13 Example of how to explicitly place a named section with scatter-loading on page 7-150.

7.21 Restrictions on placing __at sections on page 7-164.

Related references

9.6 --autoat, --no_autoat on page 9-248.

9.74 --map, --no_map on page 9-322.

9.100 --scatter=filename on page 9-349.

9.81 --output=filename on page 9-329.

Related information

--split_sections.

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

__attribute__((at(address))) variable attribute.

__attribute__((section("name"))) variable attribute.

#pragma arm section [section_type_list].

-c compiler option.

-g compiler option.

AREA directive.

7.13 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.

To place a named section explicitly using scatter-loading:

```
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`.

Related concepts

7.11 Root execution regions and the FIXED attribute on page 7-144.

8.3 Load region descriptions on page 8-203.

8.6 Execution region descriptions on page 8-207.

7.6 Scatter file with link to bit-band objects on page 7-139.

Related references

8.5 Load region attributes on page 8-205.

8.8 Execution region attributes on page 8-210.

8.9 Address attributes for load and execution regions on page 8-213.

Related information

ENTRY.

7.14 Placement of unassigned sections with the .ANY module selector

The linker attempts to place input sections into specific execution regions. For any input sections that cannot be resolved, and where the placement of those sections is not important, you can use the `.ANY` module selector in the scatter file.

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

Placement rules 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:

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

You can specify a maximum amount of space to use for unassigned sections with the exection region attribute `ANY_SIZE`.

Prioritization of .ANY sections

Prioritze the order of multiple `.ANY` sections with the `.ANY`*num* selector, where *num* is a positive integer from zero upwards.

The highest priority is given to the selector with the highest integer.

The following example shows how to use `.ANY`*num*:

```
lr1 0x8000 1024
{
    er1 +0 512
    {
        .ANY1(+RO) ; evenly distributed with er3
    }
    er2 +0 256
    {
        .ANY2(+RO) ; Highest priority, so filled first
    }
    er3 +0 256
    {
        .ANY1(+RO) ; evenly distributed with er1
    }
}
```

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

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.

Specify the maximum region size permitted for placing 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.
- You can use `ANY_SIZE` on a region without a `.ANY` selector but it is ignored by `armlink`.

When `ANY_SIZE` is present, `armlink`:

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

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

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

If you specify `--any_contingency` on the command line, it is ignored for regions that have `ANY_SIZE` specified. It is used as normal for regions that do not have `ANY_SIZE` specified.

The following example shows how to use `ANY_SIZE`:

```
LOAD_REGION 0x0 0x3000
{
    ER_1 0x0 ANY_SIZE 0xF00 0x1000
    {
        .ANY
    }
    ER_2 0x0 ANY_SIZE 0xFB0 0x1000
    {
        .ANY
    }
    ER_3 0x0 ANY_SIZE 0x1000 0x1000
    {
        .ANY
    }
}
```

In this example:

- `ER_1` has `0x100` reserved for linker-generated content.
- `ER_2` has `0x50` reserved for linker-generated content. That is about the same as the automatic contingency of `--any_contingency`.

- `ER_3` has no reserved space. Therefore 100% of the region is filled, with no contingency for veneers. Omitting the `ANY_SIZE` parameter causes 98% of the region to be filled, with a two percent contingency for veneers.

Related concepts

7.15 Examples of using placement algorithms for .ANY sections on page 7-154.

7.16 Example of next_fit algorithm showing behavior of full regions, selectors, and priority on page 7-156.

7.17 Examples of using sorting algorithms for .ANY sections on page 7-158.

7.42 How the linker resolves multiple matches when processing scatter files on page 7-193.

7.43 Behavior when .ANY sections overflow because of linker-generated content on page 7-195.

7.42 How the linker resolves multiple matches when processing scatter files on page 7-193.

7.43 Behavior when .ANY sections overflow because of linker-generated content on page 7-195.

Related references

9.3 --any_sort_order=order on page 9-245.

9.74 --map, --no_map on page 9-322.

9.101 --section_index_display=type on page 9-350.

9.121 --tiebreaker=option on page 9-371.

9.2 --any_placement=algorithm on page 9-243.

9.1 --any_contingency on page 9-242.

8.16 Syntax of an input section description on page 8-221.

9.53 --info=topic[,topic,...] on page 9-299.

7.15 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 7-2 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 used for the examples 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 situation where several sections of equal size are assigned to two regions with one selector. The selectors are equally specific, equivalent to `.ANY(+R0)` and have no priority.

```
   Execution Region ER_1 (Base: 0x00000100, Size: 0x00000010, Max: 0x00000010, ABSOLUTE)
   Base Addr    Size         Type   Attr      Idx    E Section Name        Object
   0x00000100   0x00000004   Code   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 completely full. `ER_2` is then considered.

- For `best_fit` the linker assigns `sec1` to `ER_1`. It then has two regions of equal priority and specificity, but `ER_1` has less space remaining. Therefore, the linker assigns `sec2` to `ER_1`, and continues assigning sections until `ER_1` is full.

Example for worst_fit

This example shows the image memory map when using the `worst_fit` algorithm.

```
Execution Region ER_1 (Base: 0x00000100, Size: 0x0000000c, Max: 0x00000010, ABSOLUTE)
Base Addr    Size         Type   Attr      Idx    E Section Name        Object
0x00000100   0x00000004   Code   RO            1    sec1                sections.o
0x00000104   0x00000004   Code   RO            3    sec3                sections.o
0x00000108   0x00000004   Code   RO            5    sec5                sections.o
Execution Region ER_2 (Base: 0x00000200, Size: 0x0000000c, Max: 0x00000010, ABSOLUTE)
Base Addr    Size         Type   Attr      Idx    E Section Name        Object
0x00000200   0x00000004   Code   RO            2    sec2                sections.o
0x00000204   0x00000004   Code   RO            4    sec4                sections.o
0x00000208   0x00000004   Code   RO            6    sec6                sections.o
```

The linker first assigns `sec1` to `ER_1`. It then has two equally specific and priority regions. It assigns `sec2` to the one with the most free space, `ER_2` in this example. The regions now have the same amount of space remaining, so the linker assigns `sec3` to the first one that appears in the scatter file, that is `ER_1`.

Note

The behavior of `worst_fit` is the default behavior in this version of the linker, and it is the only algorithm available and earlier linker versions.

Related concepts

7.14 Placement of unassigned sections with the .ANY module selector on page 7-151.

7.16 Example of next_fit algorithm showing behavior of full regions, selectors, and priority on page 7-156.

Related references

9.100 --scatter=filename on page 9-349.

9.2 --any_placement=algorithm on page 9-243.

7.16 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 7-3 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

7.14 Placement of unassigned sections with the .ANY module selector on page 7-151.

7.15 Examples of using placement algorithms for .ANY sections on page 7-154.

7.42 How the linker resolves multiple matches when processing scatter files on page 7-193.

7.43 Behavior when .ANY sections overflow because of linker-generated content on page 7-195.

Related references

9.100 --scatter=filename on page 9-349.

9.2 --any_placement=algorithm on page 9-243.

7.17 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 tables:

Table 7-4 Input section properties for sections_a.o

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

Table 7-5 Input section properties for sections_b.o

Name	Size
secb_1	0x4
secb_2	0x4
secb_3	0x10
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 order that the sections are processed by the `.ANY` assignment algorithm is:

Table 7-6 Sort order for descending_size algorithm

Name	Size
seca_4	0x14
secb_4	0x14
seca_3	0x10
secb_3	0x10
seca_1	0x4
seca_2	0x4
secb_1	0x4
secb_2	0x4

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

Command-line example

The following linker command-line options are used for this example:

```
--any_sort_order=cmdline sections_a.o sections_b.o --scatter scatter.txt
```

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

Table 7-7 Sort order for cmdline algorithm

Name	Size
seca_1	0x4
secb_1	0x4
seca_2	0x4
secb_2	0x4
seca_3	0x10
secb_3	0x10
seca_4	0x14
secb_4	0x14

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

Related concepts

7.14 Placement of unassigned sections with the .ANY module selector on page 7-151.

Related references

9.3 --any_sort_order=order on page 9-245.
9.100 --scatter=filename on page 9-349.
9.121 --tiebreaker=option on page 9-371.

7.18 Placement of veneer input sections in a scatter file

You can place veneers at a specific location with a linker-generated symbol.

Veneers allow switching between ARM and Thumb code or allow a longer program jump than can be specified in a single instruction. 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 encoded Thumb code, 16Mb of 32-bit encoded Thumb code, and 32Mb of ARM code.

Related concepts

3.16 Overview of veneers on page 3-57.

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

You can place code and data by separating them into their own objects without having to use toolchain-specific pragmas or attributes.

However, you can also use `__attribute__((section("name")))` to place an item in a separate ELF section. You can then use a scatter file to place the named sections at specific locations.

Examples

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

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

   Naming a section

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

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

   Placing a section

   ```
   FLASH 0x24000000 0x4000000
   {
       …                                ; rest of code
       ADDER 0x08000000
       {
           file.o (foo)                  ; select section foo from file.o
       }
   }
   ```

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

```
…
  Load Region FLASH (Base: 0x24000000, Size: 0x00000004, Max: 0x04000000, ABSOLUTE)
    Execution Region ADDER (Base: 0x08000000, Size: 0x00000004, Max: 0xffffffff, ABSOLUTE)
    Base Addr    Size         Type   Attr      Idx    E Section Name        Object
    0x08000000   0x00000004   Data   RW          16    foo                 file.o
…
```

Be aware of the following:

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

You can also place a function at a specific address using `.ARM.__at_address` as the section name. For example, to place the function `sqr` at `0x20000`, specify:

```
int sqr(int n1) __attribute__((section(".ARM.__at_0x20000")));
int sqr(int n1)
{
    return n1*n1;
}
```

Related concepts

7.20 Placement of __at sections at a specific address on page 7-163.
7.21 Restrictions on placing __at sections on page 7-164.

Related references

9.6 --autoat, --no_autoat on page 9-248.

9.100 --scatter=filename on page 9-349.

Related information

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

__attribute__((section("name"))) variable attribute.

#pragma arm section [section_type_list].

7.20 Placement of __at sections at a specific address

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

You specify the special name as follows:

```
.ARM.__at_address
```

Where *address* is the required address of the section, normalized to eight hexadecimal digits. You can specify this in hexadecimal or decimal. Sections in the form of **.ARM.__at_***address* are referred to by the abbreviation __at.

In the compiler, you can assign variables to __at sections by:

- Explicitly naming the section using the __attribute__((section("*name*"))).
- Using the attribute __at that sets up the name of the section for you.

Assigning variables to __at sections in C or C++ code

```
// place variable1 in a section called .ARM.__AT_0x00008000
int variable1 __attribute__((at(0x8000))) = 10;
// place variable2 in a section called .ARM.__at_0x8000
int variable2 __attribute__((section(".ARM.__at_0x8000"))) = 10;
```

Note

When using __attribute__((at(*address*))), the part of the __at section name representing *address* is normalized to an eight digit hexadecimal number. 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

7.12 Methods of placing functions and data at specific addresses on page 7-146.
7.19 Placement of code and data with __attribute__((section("name"))) on page 7-161.
7.21 Restrictions on placing __at sections on page 7-164.
7.22 Automatic placement of __at sections on page 7-165.
7.23 Manual placement of __at sections on page 7-167.
7.24 Placement of a key in flash memory with an __at section on page 7-168.

Related tasks

7.25 Mapping a structure over a peripheral register with an __at section on page 7-169.

Related references

9.6 --autoat, --no_autoat on page 9-248.

Related information

__attribute__((section("name"))) function attribute.
__attribute__((at(address))) variable attribute.
__attribute__((section("name"))) variable attribute.

7.21 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 *System V* (SysV) and *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 concepts

7.20 Placement of __at sections at a specific address on page 7-163.

Related information

Base Platform ABI for the ARM Architecture.

7.22 Automatic placement of __at sections

The linker automatically places __at sections, but you can override this.

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

Note

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

When linking with the --autoat option, the __at sections are not placed by the scatter-loading selectors. Instead, the linker places the __at section in a compatible region. If no compatible region is found, the linker creates a load and execution region for the __at section.

All linker --autoat created execution regions have the UNINIT scatter-loading attribute. If you require a ZI __at section to be zero-initialized then it must be placed within a compatible region. A linker --autoat created execution region must have a base address that is at least 4 byte-aligned. The linker produces an error message if any region is incorrectly aligned.

A compatible region is one where:

- The __at address lies within the execution region base and limit, where limit is the base address + maximum size of execution region. If no maximum size is set, the linker sets the limit for placing __at sections as the current size of the execution region without __at sections plus a constant. 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, RW or ZI) as the __at section.
  - It does not have the EMPTY attribute.

    Note

    The linker considers an __at section with type RW compatible with RO.

Examples

The following example shows the manual placement of variables is achieved in C or C++ code, with the sections .ARM.__at_0x0000 type RO, .ARM.__at_0x2000 type RW, .ARM.__at_0x4000 type ZI, and .ARM.__at_0x8000 type ZI:

```
// place the RW variable in a section called .ARM.__at_0x02000
int foo __attribute__((section(".ARM.__at_0x2000"))) = 100;
// place the ZI variable in a section called .ARM.__at_0x04000
int bar __attribute__((section(".ARM.__at_0x4000"), zero_init));
// place the ZI variable in a section called .ARM.__at_0x8000
int variable __attribute__((section(".ARM.__at_0x8000"), zero_init));
```

The following scatter file shows how the placement of __at sections is achieved automatically:

```
LR1 0x0
{
    ER_RO 0x0 0x2000
    {
        *(+RO)      ; .ARM.__at_0x0000 lies within the bounds of ER_RO
    }
    ER_RW 0x2000 0x2000
    {
        *(+RW)      ; .ARM.__at_0x2000 lies within the bounds of ER_RW
```

```
    }
    ER_ZI 0x4000 0x2000
    {
        *(+ZI)      ; .ARM.__at_0x4000 lies within the bounds of ER_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

7.20 Placement of __at sections at a specific address on page 7-163.
7.23 Manual placement of __at sections on page 7-167.
7.24 Placement of a key in flash memory with an __at section on page 7-168.
8.6 Execution region descriptions on page 8-207.
7.19 Placement of code and data with __attribute__((section("name"))) on page 7-161.
7.21 Restrictions on placing __at sections on page 7-164.

Related tasks

7.25 Mapping a structure over a peripheral register with an __at section on page 7-169.

Related references

9.6 --autoat, --no_autoat on page 9-248.
9.94 --ro_base=address on page 9-343.
9.97 --rw_base=address on page 9-346.
9.136 --xo_base=address on page 9-386.
9.140 --zi_base=address on page 9-390.
8.8 Execution region attributes on page 8-210.
9.76 --max_er_extension=size on page 9-324.

7.23 Manual placement of __at sections

You can have direct control over the placement of `__at` sections, if required.

You can use the standard section placement rules to place `__at` sections when using the `--no_autoat` command-line option.

Note

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

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 the manual placement of __at sections is achieved:

```
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

7.20 Placement of __at sections at a specific address on page 7-163.

7.22 Automatic placement of __at sections on page 7-165.

7.24 Placement of a key in flash memory with an __at section on page 7-168.

8.6 Execution region descriptions on page 8-207.

7.19 Placement of code and data with __attribute__((section("name"))) on page 7-161.

7.21 Restrictions on placing __at sections on page 7-164.

Related tasks

7.25 Mapping a structure over a peripheral register with an __at section on page 7-169.

Related references

9.6 --autoat, --no_autoat on page 9-248.

8.8 Execution region attributes on page 8-210.

7.24 Placement of 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.

Placement of the flash key variable in C or C++ code

Assuming a device has flash memory from `0x8000` to `0x10000` and a key is required in address `0x8000`. To do this with an `__at` section, you must declare a variable so that the compiler can generate a section called `.ARM.__at_0x8000`.

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

Manual placement of flash execution regions

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

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

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

Automatic placement of flash execution regions

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

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

Related concepts

7.20 Placement of __at sections at a specific address on page 7-163.

7.22 Automatic placement of __at sections on page 7-165.

7.23 Manual placement of __at sections on page 7-167.

8.6 Execution region descriptions on page 8-207.

3.12 Section placement with the FIRST and LAST attributes on page 3-52.

Related references

9.6 --autoat, --no_autoat on page 9-248.

Related information

__attribute__((section("name"))) variable attribute.

7.25 Mapping a structure over a peripheral register with an __at section

You can place a structure over a peripheral register with scatter-loading.

To place an uninitialized variable over a peripheral register, you can use a ZI `__at` section. Assuming a register is available for use at `0x10000000`, define a ZI `__at` section called `.ARM.__at_0x10000000`. For example:

```
int foo __attribute__((section(".ARM.__at_0x10000000"), zero_init));
```

The following example shows the a scatter file with the manual placement of the ZI `__at` section:

```
ER_PERIPHERAL 0x10000000 UNINIT
{
    *(.ARM.__at_0x10000000)
}
```

Using automatic placement, and assuming that there is no other execution region near `0x10000000`, the linker automatically creates a region with the `UNINIT` attribute at `0x10000000`. The `UNINIT` attribute creates an execution region containing uninitialized data or memory-mapped I/O.

Related concepts

7.20 Placement of __at sections at a specific address on page 7-163.

7.22 Automatic placement of __at sections on page 7-165.

7.23 Manual placement of __at sections on page 7-167.

8.6 Execution region descriptions on page 8-207.

Related references

8.8 Execution region attributes on page 8-210.

Related information

__attribute__((section("name"))) variable attribute.

7.26 Placement of sections with overlays

You can place multiple execution regions at the same address with overlays.

The OVERLAY attribute allows you to place multiple execution regions at the same address. An overlay manager is required to make sure that only one execution region is instantiated at a time. µVision does not provide an overlay manager.

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

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

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

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

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

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

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

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

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

Table 7-8 Using relative offset in overlays

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

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

```
EMB_APP 0x8000{
    CODE 0x8000
    {
```

```
        *(+RO)
    }
    # REGION1 Base = CODE limit
    REGION1 +0 OVERLAY
    {
        module1.o(*)
    }
    # REGION2 Base = REGION1 Base
    REGION2 +0 OVERLAY
    {
        module2.o(*)
    }
    # REGION3 Base = REGION2 Base = REGION1 Base
    REGION3 +0 OVERLAY
    {
        module3.o(*)
    }
    # REGION4 Base = REGION3 Limit + 4
    Region4 +4 OVERLAY
    {
        module4.o(*)
    }
}
```

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

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

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

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

Related concepts

7.20 Placement of __at sections at a specific address on page 7-163.
8.3 Load region descriptions on page 8-203.
8.10 Considerations when using a relative address +offset for load regions on page 8-215.
8.11 Considerations when using a relative address +offset for execution regions on page 8-216.
6.2 Linker-defined symbols on page 6-105.
8.6 Execution region descriptions on page 8-207.

Related references

9.35 --emit_debug_overlay_relocs on page 9-280.
9.36 --emit_debug_overlay_section on page 9-281.
8.5 Load region attributes on page 8-205.
8.8 Execution region attributes on page 8-210.
8.9 Address attributes for load and execution regions on page 8-213.

Related information

__attribute__((section("name"))) variable attribute.
ABI for the ARM Architecture: Support for Debugging Overlaid Programs.

7.27 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 `*cpplib*` so that the linker can resolve library naming in your scatter file.

Some ARM C and C++ library sections must be placed in a root region, for example `__main.o`, `__scatter*.o`, `__dc*.o`, and `*Region$$Table`. This list can change between releases. The linker can place all these sections automatically in a future-proof way with `InRoot$$Sections`.

Related concepts

7.9 Root execution regions on page 7-142.
7.28 Example of placing code in a root region on page 7-173.
7.29 Example of placing ARM C library code on page 7-174.
7.30 Example of placing ARM C++ library code on page 7-175.
7.10 Root execution regions and the ABSOLUTE attribute on page 7-143.
7.11 Root execution regions and the FIXED attribute on page 7-144.

7.28 Example of placing code in a root region

This example shows how to use a scatter file to specify a root section. It is similar to placing a named section.

The section selector `InRoot$$Sections` in this example places all sections that must be in a root region:

```
ROM_LOAD 0x0000 0x4000
{
  ROM_EXEC 0x0000 0x4000      ; root region at 0x0
  {
    vectors.o (Vect, +FIRST)  ; Vector table
    * (InRoot$$Sections)      ; All library sections that must be in a
                              ; root region, for example, __main.o,
                              ; __scatter*.o, __dc*.o, and * Region$$Table
  }
  RAM 0x10000 0x8000
  {
    * (+RO, +RW, +ZI)         ; all other sections
  }
}
```

Related concepts

7.27 Placement of ARM C and C++ library code on page 7-172.
7.29 Example of placing ARM C library code on page 7-174.
7.30 Example of placing ARM C++ library code on page 7-175.
7.10 Root execution regions and the ABSOLUTE attribute on page 7-143.
7.11 Root execution regions and the FIXED attribute on page 7-144.
7.9 Root execution regions on page 7-142.

7.29 Example of placing ARM C library code

You can place C library code using a scatter file.

The following example shows how to place C library code:

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

Related concepts

7.27 Placement of ARM C and C++ library code on page 7-172.
7.28 Example of placing code in a root region on page 7-173.
7.30 Example of placing ARM C++ library code on page 7-175.

Related information

C and C++ library naming conventions.

7.30 Example of placing ARM C++ library code

You can place C++ library code using a scatter file.

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

```
#include <iostream>
using namespace std;
extern "C" int foo ()
{
  cout << "Hello" << endl;
  return 1;
}
```

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

```
LR 0x0
{
    ER1 0x0
    {
        *armlib*(+RO)
    }
    ER2 +0
    {
        *cpplib*(+RO)
        *(.init_array)   ; Section .init_array must be placed explicitly,
                         ; otherwise it is shared between two regions, and
                         ; the linker is unable to decide where to place it.
    }
    ER3 +0
    {
        *(+RO)
    }
    ER4 +0
    {
        *(+RW,+ZI)
    }
}
```

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

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

Related concepts

7.27 Placement of ARM C and C++ library code on page 7-172.

7.28 Example of placing code in a root region on page 7-173.

7.29 Example of placing ARM C library code on page 7-174.

Related information

C and C++ library naming conventions.

7.31 Example of placing ARM library helper functions

Placing ARM library helper functions using a scatter file cannot be done using `armlib` and `cpplib`.

ARM library helper functions are generated by the compiler in the resulting object files. Therefore, you cannot use `armlib` and `cpplib` in a scatter file to place these functions.

To place the helper functions specify `*.* (i.__ARM_*)` in your scatter file. The `*.*` part is important if you have `* (+RO)` in your scatter file.

Be aware that if you use `* (i.__ARM_*)` the following error is generated:

```
Error: L6223E: Ambiguous selectors…
```

This is because of the scatter-loading rules for resolving multiple matches.

Related concepts

7.42 How the linker resolves multiple matches when processing scatter files on page 7-193.

7.32 Reserving an empty region

You can reserve an empty block of memory with a scatter file, such as the area used for the stack. Use the `EMPTY` attribute for the execution region in the scatter-loading description.

to reserve an empty block of memory for the stack.

The block of memory does not form part of the load region, but is assigned for use at execution time. Because it is created as a dummy ZI region, the linker uses the following symbols to access it:

- `Image$$`*`region_name`*`$$ZI$$Base`.
- `Image$$`*`region_name`*`$$ZI$$Limit`.
- `Image$$`*`region_name`*`$$ZI$$Length`.

If the length is given as a negative value, the address is taken to be the end address of the region. This must be an absolute address and not a relative one.

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

```
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.

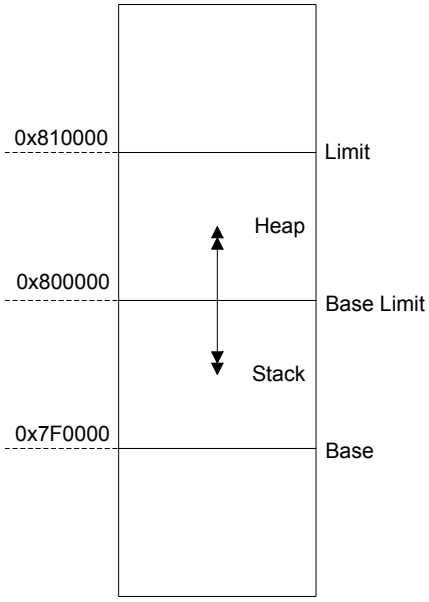

Figure 7-4 Reserving a region for the stack

In this example, the linker generates the symbols:

```
Image$$STACK$$ZI$$Base      = 0x7f0000
Image$$STACK$$ZI$$Limit     = 0x800000
Image$$STACK$$ZI$$Length    = 0x10000
Image$$HEAP$$ZI$$Base       = 0x800000
Image$$HEAP$$ZI$$Limit      = 0x810000
Image$$HEAP$$ZI$$Length     = 0x10000
```

Note

The `EMPTY` attribute applies only to an execution region. The linker generates a warning and ignores an `EMPTY` attribute used in a load region definition.

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

Related concepts

8.6 Execution region descriptions on page 8-207.

Related references

6.4 Image$$ execution region symbols on page 6-107.

8.8 Execution region attributes on page 8-210.

7.33 Creation of regions on page boundaries

You can produce an ELF file that can be loaded directly to a target with each execution region starting at a page boundary.

The linker provides the following built-in functions to help create load and execution regions on page boundaries:

- `AlignExpr`.
- `GetPageSize`. You must also use the `--paged` command-line option if you use this function.

Note

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

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

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

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

Related concepts

7.34 Overalignment of execution regions and input sections on page 7-180.

3.14 Linker support for creating demand-paged files on page 3-54.

8.17 Expression evaluation in scatter files on page 8-225.

7.37 Example of using expression evaluation in a scatter file to avoid padding on page 7-184.

8.24 Example of aligning a base address in execution space but still tightly packed in load space on page 8-234.

Related references

8.25 AlignExpr(expr, align) function on page 8-235.

8.26 GetPageSize() function on page 8-236.

9.84 --paged on page 9-332.

8.5 Load region attributes on page 8-205.

8.8 Execution region attributes on page 8-210.

9.85 --pagesize=pagesize on page 9-333.

7.34 Overalignment of execution regions and input sections

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

- If you have access to the original source code, you can do this at compile time with the `__align(n)` keyword or the `--min_array_alignment` command-line option, for example.
- If you do not have access to the source code, then you must use the following alignment specifiers in a scatter file:

`ALIGNALL`
: Increases the section alignment of all the sections in an execution region, for example:

```
ER_DATA … ALIGNALL 8
{
    … ;selectors
}
```

`OVERALIGN`
: Increases the alignment of a specific section, for example:

```
ER_DATA …
{
   *.o(.bar, OVERALIGN 8)
   … ;selectors
}
```

Related concepts

7.33 Creation of regions on page boundaries on page 7-179.

8.15 Input section descriptions on page 8-220.

Related references

8.8 Execution region attributes on page 8-210.

Related information

__align.

--min_array_alignment=opt compiler option.

7.35 Preprocessing of 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 [pre_processor_flags]
```

Most typically the command is `#! armcc -E`. This passes the scatter file through the `armcc` 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:

```
#! armcc -E
#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 preprocessing of a scatter file in conjunction with the `--predefine` command-line option. For this example:

1. Modify `file.scat` to delete the directive `#define ADDRESS 0x20000000`.
2. Specify the command:

   ```
   armlink --predefine="-DADDRESS=0x20000000" --scatter=file.scat
   ```

Default behavior for `armcc -E`

`armlink` behaves in the same way as `armcc` when invoking other ARM tools. It searches for the `armcc` binary in the following order:

- The same location as `armlink`.
- The `PATH` locations.

`armcc` is invoked with the option `-I`*`scatter_file_path`* so that any relative `#include`s work. The linker only adds this option if the full name of the preprocessor tool given is `armcc` or `armcc.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 `armcc.exe` is considered the same as `armcc`. Executable names are case insensitive, so `ARMCC` is considered the same as `armcc`. The portable way to write scatter file preprocessing lines is to use correct capitalization, and omit the `.exe` suffix.

Using other preprocessors

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

8.17 Expression evaluation in scatter files on page 8-225.

Related references

9.88 --predefine="string" on page 9-336.

7.36 Available operators for expression evaluation

The linker can carry out simple expression evaluation with a restricted set of operators.

The operators are `+`, `-`, `*`, `/`, `AND`, `OR`, and parentheses. The implementation of `OR` and `AND` follows C operator precedence rules.

Examples

Use the directives:

```
#define BASE_ADDRESS 0x8000
#define ALIAS_NUMBER 0x2
#define ALIAS_SIZE 0x400
#define AN_ADDRESS (BASE_ADDRESS+(ALIAS_NUMBER*ALIAS_SIZE))
```

The scatter file might contain:

```
LOAD_FLASH AN_ADDRESS    ; start address
```

After preprocessing, this evaluates to:

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

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

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

Related concepts

7.37 Example of using expression evaluation in a scatter file to avoid padding on page 7-184.
7.35 Preprocessing of a scatter file on page 7-181.

7.37 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.

Examples

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

7.36 Available operators for expression evaluation on page 7-183.

8.24 Example of aligning a base address in execution space but still tightly packed in load space on page 8-234.

Related references

8.25 AlignExpr(expr, align) function on page 8-235.

8.8 Execution region attributes on page 8-210.

7.38 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.

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

- Type 1 image, one load region and contiguous execution regions.
- Type 2 image, one load region and non-contiguous execution regions.
- Type 3 image, two load regions and non-contiguous execution regions.

You can create the same image types by using the `--scatter` command-line option and a file containing one of the corresponding scatter-loading descriptions.

Related concepts

3.5 Types of simple image on page 3-41.
7.39 Type 1 image, one load region and contiguous execution regions on page 7-186.
8.3 Load region descriptions on page 8-203.
7.40 Type 2 image, one load region and non-contiguous execution regions on page 7-188.
7.41 Type 3 image, multiple load regions and non-contiguous execution regions on page 7-190.

Related references

9.91 --reloc on page 9-340.
9.94 --ro_base=address on page 9-343.
9.95 --ropi on page 9-344.
9.97 --rw_base=address on page 9-346.
9.98 --rwpi on page 9-347.
9.100 --scatter=filename on page 9-349.
9.107 --split on page 9-357.
9.136 --xo_base=address on page 9-386.
8.5 Load region attributes on page 8-205.

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

A Type 1 image consists of a single load region in the load view and three execution regions in the execution view. The execution regions are placed contiguously in the memory map.

`--ro_base address` specifies the load and execution address of the region containing the RO output section. The following example shows the scatter-loading description equivalent to using `--ro_base 0x040000`:

```
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, 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.

ROPI example variant

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

Be aware that if an image contains execute-only sections, ROPI is not supported. `armlink` gives an error if you use `--ropi` to link such an image.

Related concepts

7.38 Equivalent scatter-loading descriptions for simple images on page 7-185.

8.3 Load region descriptions on page 8-203.

8.10 Considerations when using a relative address +offset for load regions on page 8-215.

8.11 Considerations when using a relative address +offset for execution regions on page 8-216.

Related references

9.94 --ro_base=address on page 9-343.

9.95 --ropi on page 9-344.

8.5 Load region attributes on page 8-205.

9.91 --reloc on page 9-340.

7.40 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.

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

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.

Note

Be aware that if an image contains execute-only memory, RWPI is not supported. `armlink` gives an error if you use `--rwpi` to link such an image.

Related concepts

8.3 Load region descriptions on page 8-203.

8.10 Considerations when using a relative address +offset for load regions on page 8-215.

8.11 Considerations when using a relative address +offset for execution regions on page 8-216.

Related references

9.94 --ro_base=address on page 9-343.

9.97 --rw_base=address on page 9-346.

9.98 --rwpi on page 9-347.

9.136 --xo_base=address on page 9-386.

8.5 Load region attributes on page 8-205.

7.41 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.

Note

You cannot use this option if an object file contains execute-only sections.

`--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.

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 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 8-203.

8.10 Considerations when using a relative address +offset for load regions on page 8-215.

8.11 Considerations when using a relative address +offset for execution regions on page 8-216.

Related references

9.91 --reloc on page 9-340.

9.94 --ro_base=address on page 9-343.

9.97 --rw_base=address on page 9-346.

9.107 --split on page 9-357.

9.136 --xo_base=address on page 9-386.

8.5 Load region attributes on page 8-205.

8.9 Address attributes for load and execution regions on page 8-213.

7.42 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.

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 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 sequence to determine the most specific *module_select_pattern*, *input_section_selector* pair:

1. 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*.
2. For the input section selectors:
   - If *s1* and *s2* are both patterns matching section names, the same definition as for module selector patterns is used.
   - If one of *s1*, *s2* matches the input section name 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 input section attributes, the determination of whether *s1* is more specific than *s2* is defined by the relationships below:
     - `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.
3. For the *module_select_pattern, input_section_selector* pair, *m1,s1* is more specific than *m2,s2* only if any of the following are true:

   a. *s1* is a literal input section name that is, it contains no pattern characters, and *s2* matches input section attributes other than `+ENTRY`.

   b. *m1* is more specific than *m2*.

   c. *s1* is more specific than *s2*.

   The conditions are tested in order so condition *3.a* takes precedence over condition *3.b* and *3.c*, and condition *3.b* takes precedence over condition *3.c*.

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 fully names an input section and the other selects by attribute. In this case, the explicit input section name is more specific than any attribute, other than `ENTRY`, that selects exactly one input section from one object. 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.

Multiple execution regions and pattern matching

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

7.14 Placement of unassigned sections with the .ANY module selector on page 7-151.

7.14 Placement of unassigned sections with the .ANY module selector on page 7-151.

8.15 Input section descriptions on page 8-220.

Related references

8.2 Syntax of a scatter file on page 8-202.

8.16 Syntax of an input section description on page 8-221.

7.43 Behavior when .ANY sections overflow because of linker-generated content

Because linker-generated content might cause `.ANY` regions 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 additional two percent for veneers. To enable this algorithm use the `--any_contingency` command-line option.

The following diagram represents the notional image layout during `.ANY` placement:

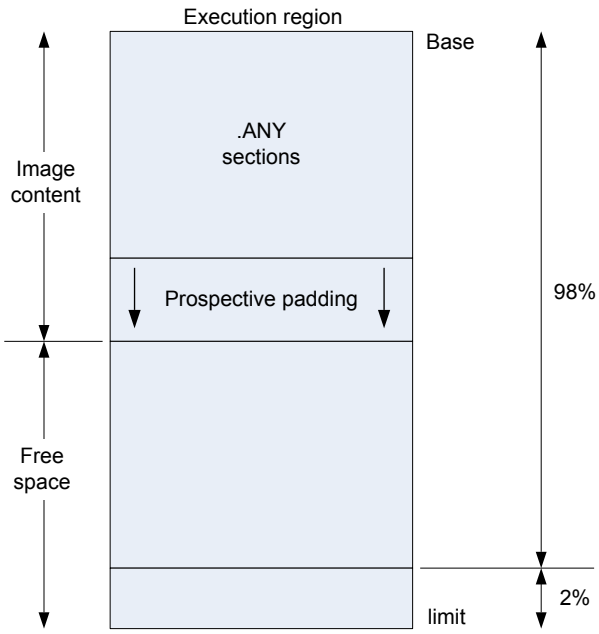

Figure 7-5 .ANY contingency

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.

Related concepts

7.14 Placement of unassigned sections with the .ANY module selector on page 7-151.
7.14 Placement of unassigned sections with the .ANY module selector on page 7-151.
7.42 How the linker resolves multiple matches when processing scatter files on page 7-193.

Related references

8.8 Execution region attributes on page 8-210.
8.16 Syntax of an input section description on page 8-221.
9.1 --any_contingency on page 9-242.

7.44 How the linker resolves path names when processing scatter files

The linker matches wildcard patterns in scatter files against any combination of forward slashes and backslashes it finds in path names.

This might be useful where the paths are taken from environment variables or multiple sources, or where you want to use the same scatter file to build on Windows or Unix platforms.

Note

Use forward slashes in path names to ensure they are understood on Windows and Unix platforms.

Related references

8.2 Syntax of a scatter file on page 8-202.

7.45 Scatter file to ELF mapping

Shows how scatter file components map onto ELF.

For simple images, ELF executable files contain segments:

- A load region is represented by an ELF program segment with type PT_LOAD.
- An execution region is represented by 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 About scatter-loading on page 7-132.

7.4 Scatter-loading images with a simple memory map on page 7-135.

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 references

8.2 Syntax of a scatter file on page 8-202.

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:

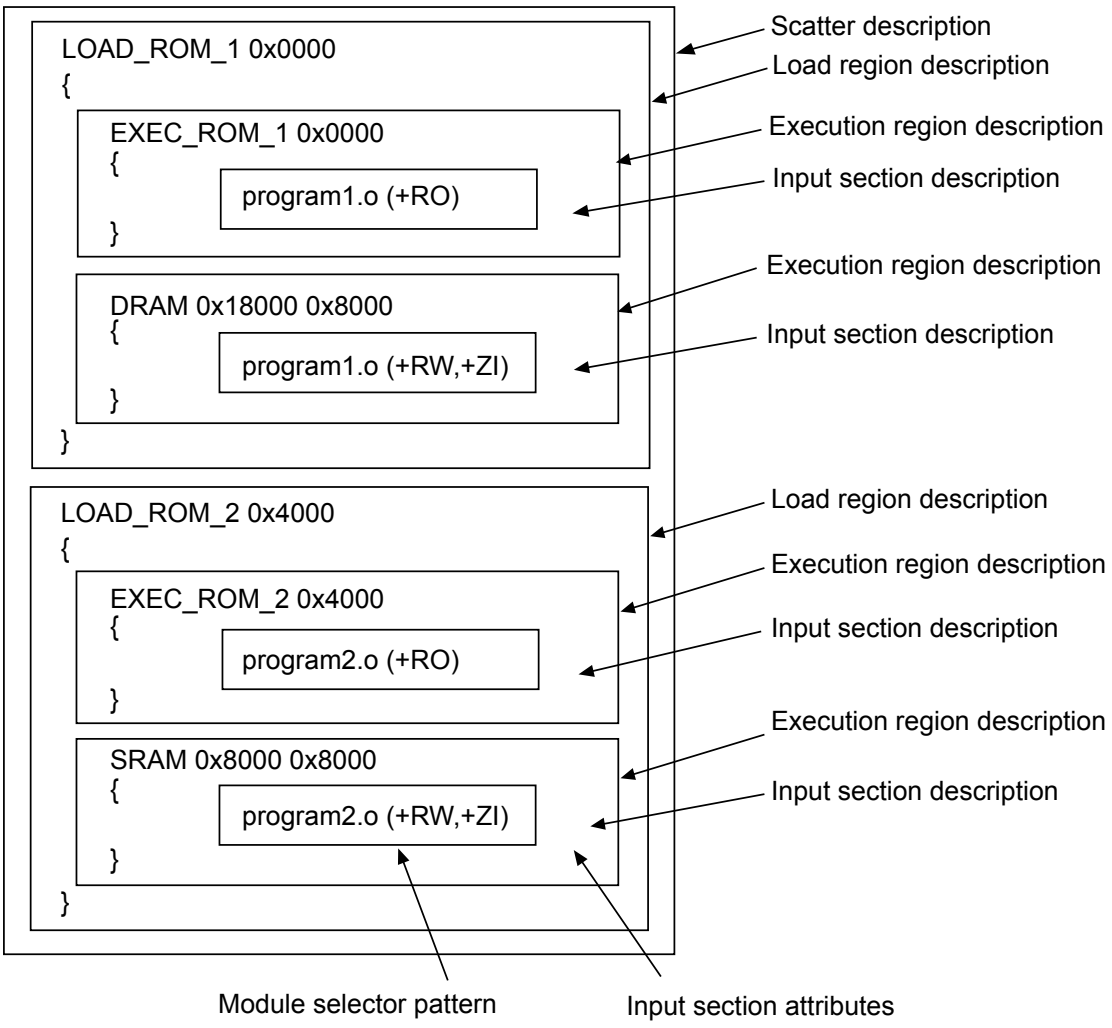

Figure 8-1 Components of a scatter file

Related concepts

8.3 Load region descriptions on page 8-203.

8.6 Execution region descriptions on page 8-207.

Related references

7 Scatter-loading Features on page 7-130.

8.3 Load region descriptions

A load region description specifies the region of memory where its child execution regions are to be placed.

A load region description has:

- 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 the components of a typical load region description:

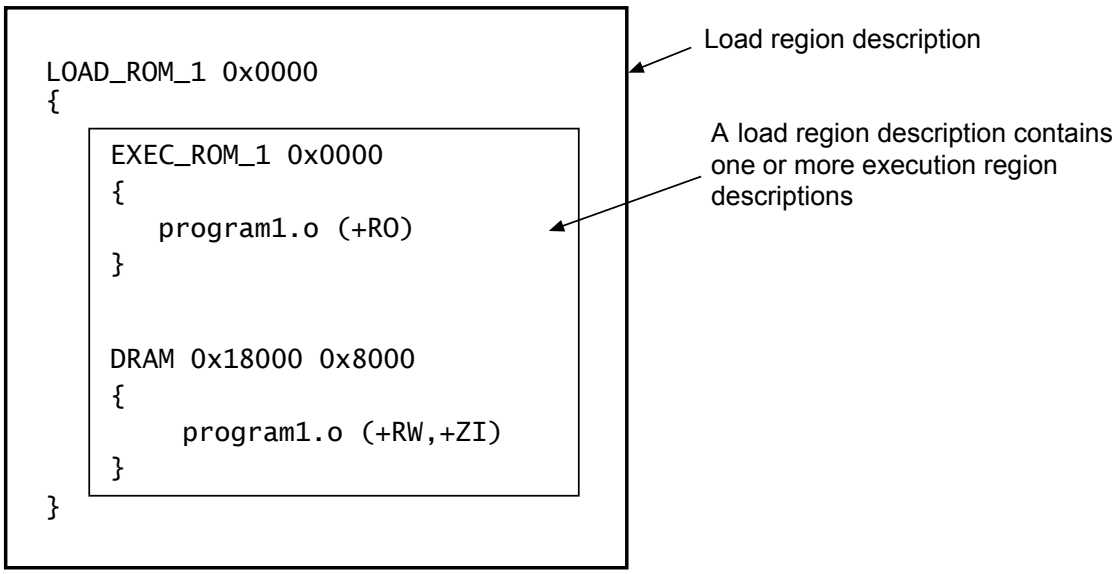

Figure 8-2 Components of a load region description

Related concepts

7.33 Creation of regions on page boundaries on page 7-179.

8.17 Expression evaluation in scatter files on page 8-225.

Related references

8.2 Syntax of a scatter file on page 8-202.

8.4 Syntax of a load region description on page 8-204.

8.5 Load region attributes on page 8-205.

8.9 Address attributes for load and execution regions on page 8-213.

7 Scatter-loading Features on page 7-130.

8.4 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 Load region descriptions on page 8-203.
8.10 Considerations when using a relative address +offset for load regions on page 8-215.
8.12 Inheritance rules for load region address attributes on page 8-217.
8.17 Expression evaluation in scatter files on page 8-225.

Related references

8.5 Load region attributes on page 8-205.
8.9 Address attributes for load and execution regions on page 8-213.
8.1 BNF notation used in scatter-loading description syntax on page 8-201.
8.2 Syntax of a scatter file on page 8-202.
6.3 Region-related symbols on page 6-106.

8.5 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
Absolute address. 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.

PI
This region is position independent.

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
This region is relocatable.

Note

This attribute is not supported if an image contains execute-only sections.

Related concepts

8.3 Load region descriptions on page 8-203.
8.24 Example of aligning a base address in execution space but still tightly packed in load space on page 8-234.
3.13 Section alignment with the linker on page 3-53.
3.20 Reuse of veneers when scatter-loading on page 3-61.
7.33 Creation of regions on page boundaries on page 7-179.
7.26 Placement of sections with overlays on page 7-170.
8.10 Considerations when using a relative address +offset for load regions on page 8-215.

8.14 Inheritance rules for the RELOC address attribute on page 8-219.
3.17 Veneer sharing on page 3-58.
3.19 Generation of position independent to absolute veneers on page 3-60.
4.7 Optimization with RW data compression on page 4-83.

Related references

8.4 Syntax of a load region description on page 8-204.
9.79 --merge, --no_merge on page 9-327.

8.6 Execution region descriptions

An execution region description specifies the region of memory where parts of your image are to be placed at run-time.

An execution region description has:

- 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:

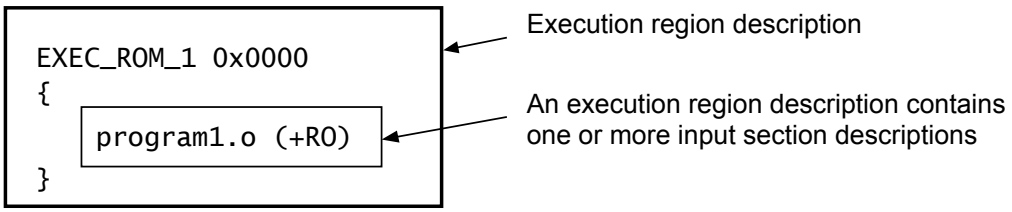

Figure 8-3 Components of an execution region description

Related concepts

7.26 Placement of sections with overlays on page 7-170.
8.17 Expression evaluation in scatter files on page 8-225.
7.33 Creation of regions on page boundaries on page 7-179.
8.15 Input section descriptions on page 8-220.

Related references

8.2 Syntax of a scatter file on page 8-202.
8.9 Address attributes for load and execution regions on page 8-213.
8.7 Syntax of an execution region description on page 8-208.
8.8 Execution region attributes on page 8-210.
7 Scatter-loading Features on page 7-130.
8.5 Load region attributes on page 8-205.

8.7 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

8.6 Execution region descriptions on page 8-207.
8.11 Considerations when using a relative address +offset for execution regions on page 8-216.
8.17 Expression evaluation in scatter files on page 8-225.
Base Platform linking model.
7.26 Placement of sections with overlays on page 7-170.

7.33 Creation of regions on page boundaries on page 7-179.
8.12 Inheritance rules for load region address attributes on page 8-217.
8.13 Inheritance rules for execution region address attributes on page 8-218.
8.14 Inheritance rules for the RELOC address attribute on page 8-219.
8.15 Input section descriptions on page 8-220.

Related references

8.8 Execution region attributes on page 8-210.
8.9 Address attributes for load and execution regions on page 8-213.
7 Scatter-loading Features on page 7-130.
6.3 Region-related symbols on page 6-106.

8.8 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 run-time.

The execution region attributes are:

ABSOLUTE
Absolute address. The execution address of the region is specified by the base designator.

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 this 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 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** ———

max_size is not the contingency, but the maximum size permitted for placing unassigned sections in an execution region. For example, if an execution region is to be filled only with `.ANY` sections, a two percent contingency is still set aside for veneers. This leaves 98% of the region for `.ANY` section assignements.

———————————

Be aware of the following restrictions when using this keyword:

- *max_size* must be less than or equal to the region size.
- You can use `ANY_SIZE` on a region without a `.ANY` selector but it is ignored by `armlink`.

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 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.

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, for example, simulation, this 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. This makes the region a root region. If this is not possible 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.

PADVALUE

Defines the value of any padding. If you specify `PADVALUE`, you must give a value, for example:

```
EXEC 0x10000 PADVALUE 0xFFFFFFFF EMPTY ZEROPAD 0x2000
```

This 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.

Note

This attribute is not supported if an image contains execute-only sections.

SORTTYPE

Specifies the sorting algorithm for the execution region, for example:

```
ER1 +0 SORTTYPE CallTree
```

UNINIT

Use to create execution regions containing uninitialized data or memory-mapped I/O.

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 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, for example, simulation, this is preferable to spending a long time in a zeroing loop.

Related concepts

8.6 Execution region descriptions on page 8-207.
7.43 Behavior when .ANY sections overflow because of linker-generated content on page 7-195.
3.13 Section alignment with the linker on page 3-53.

7.33 Creation of regions on page boundaries on page 7-179.
7.34 Overalignment of execution regions and input sections on page 7-180.
7.37 Example of using expression evaluation in a scatter file to avoid padding on page 7-184.
8.24 Example of aligning a base address in execution space but still tightly packed in load space on page 8-234.
7.26 Placement of sections with overlays on page 7-170.
8.11 Considerations when using a relative address +offset for execution regions on page 8-216.
8.17 Expression evaluation in scatter files on page 8-225.
4.7 Optimization with RW data compression on page 4-83.
8.14 Inheritance rules for the RELOC address attribute on page 8-219.

Related references

8.7 Syntax of an execution region description on page 8-208.
6.5 Load$$ execution region symbols on page 6-108.
8.25 AlignExpr(expr, align) function on page 8-235.
8.1 BNF notation used in scatter-loading description syntax on page 8-201.
9.1 --any_contingency on page 9-242.
6.4 Image$$ execution region symbols on page 6-107.
8.16 Syntax of an input section description on page 8-221.
9.106 --sort=algorithm on page 9-355.

8.9 Address attributes for load and execution regions

A subset of the load and execution region attributes inform the linker about the content of the region and how it behaves after linking.

These attributes are:

ABSOLUTE
The content is placed at a fixed address that does not change after linking.

PI
The content does not depend on any fixed address and might be moved after linking without any extra processing.

——— **Note** ———

This attribute is not supported if an image contains execute-only sections.

———————————

RELOC
The content depends on fixed addresses. Relocation information is output to enable the content to be moved to another location by another tool.

——— **Note** ———

You cannot explicitly use this attribute for an execution region.

———————————

——— **Note** ———

This attribute is not supported if an image contains execute-only sections.

———————————

OVERLAY
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.

Inheritance rules for address attributes

In general, all the execution regions within a load region have the same address attribute. To make this easy to select, the address attributes can be inherited from a previous region so that they only have to be set in one place. The rules for setting and inheriting address attributes are:

- Explicitly setting the address attribute:
  - A load region address attribute can be explicitly set with the `ABSOLUTE`, `PI`, `RELOC`, or `OVERLAY` attributes.
  - An execution region address attribute can be explicitly set with the `ABSOLUTE`, `PI`, or `OVERLAY` attributes. An execution region can only inherit the `RELOC` attribute from the parent load region.
- Implicitly setting the address attribute when none 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.
  - A *`+offset`* execution region inherits the address attribute from the previous execution region or parent load region if no previous execution region exists.

Related concepts

8.3 Load region descriptions on page 8-203.
8.6 Execution region descriptions on page 8-207.
7.26 Placement of sections with overlays on page 7-170.

8.10 Considerations when using a relative address +offset for load regions on page 8-215.
8.11 Considerations when using a relative address +offset for execution regions on page 8-216.

Related references

8.4 Syntax of a load region description on page 8-204.
8.7 Syntax of an execution region description on page 8-208.

8.10 Considerations when using a relative address +offset for load regions

There are some considerations to be aware of when using a relative address for load regions.

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`.

Related concepts

8.12 Inheritance rules for load region address attributes on page 8-217.
8.20 Execution address built-in functions for use in scatter files on page 8-228.

8.11 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

8.13 Inheritance rules for execution region address attributes on page 8-218.
8.14 Inheritance rules for the RELOC address attribute on page 8-219.

8.12 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.

Examples

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

8.10 Considerations when using a relative address +offset for load regions on page 8-215.

Related references

8.9 Address attributes for load and execution regions on page 8-213.

8.13 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 `OVERLAY` attribute.

You can explicitly set an execution region with the `ABSOLUTE`, `PI`, or `OVERLAY` attributes. However, an execution region can only inherit the `RELOC` attribute from the parent load region.

Examples

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

8.11 Considerations when using a relative address +offset for execution regions on page 8-216.

Related references

8.9 Address attributes for load and execution regions on page 8-213.

8.14 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

For a Base Platform linking model, if a load region has the `RELOC` attribute, then all execution regions within that load region must have a *`+offset`* base address. This ensures the execution regions inherit the relocations from the parent load region.

Examples

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

8.11 Considerations when using a relative address +offset for execution regions on page 8-216.

8.10 Considerations when using a relative address +offset for load regions on page 8-215.

Base Platform linking model.

Related references

8.9 Address attributes for load and execution regions on page 8-213.

8.15 Input section descriptions

An input section description is a pattern that identifies input sections.

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, or input section 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.

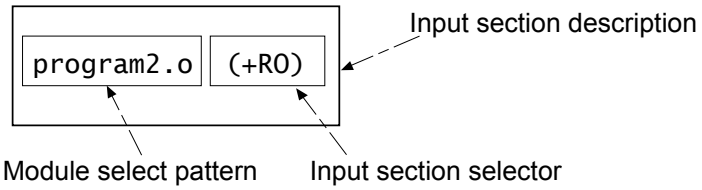

Figure 8-4 Components of an input section description

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 references

8.16 Syntax of an input section description on page 8-221.
8.2 Syntax of a scatter file on page 8-202.
9.86 --partial on page 9-334.

8.16 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_symbol_pattern |
section_properties)
```

where:

module_select_pattern

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`.

A pattern constructed from literal text. 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 matched against the input section attributes. Each *input_section_attr* follows a +.

If you are specifying a pattern to match the input section name, the name must be preceded by a +. You can omit any comma immediately followed by 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.

There can be only one `FIRST` or one `LAST` attribute for an execution region, and it must follow a single *`input_section_attr`*. 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_symbol_pattern`
You can select the input section by the name of a global symbol that the section defines. This 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.

The order of input section descriptors is not significant.

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

- 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 generated by the compiler, or used by ARM library code. These can change between compilations if, for example, different compiler options are used. In addition, section naming conventions used by the compiler are not guaranteed to remain constant between releases.
- 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.

Examples of module select patterns

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 patterns

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 concepts

8.15 Input section descriptions on page 8-220.
7.43 Behavior when .ANY sections overflow because of linker-generated content on page 7-195.
7.14 Placement of unassigned sections with the .ANY module selector on page 7-151.
7.15 Examples of using placement algorithms for .ANY sections on page 7-154.
7.16 Example of next_fit algorithm showing behavior of full regions, selectors, and priority on page 7-156.
7.17 Examples of using sorting algorithms for .ANY sections on page 7-158.
7.34 Overalignment of execution regions and input sections on page 7-180.

Related references

8.1 BNF notation used in scatter-loading description syntax on page 8-201.
8.2 Syntax of a scatter file on page 8-202.

8.17 Expression evaluation in scatter files

Scatter files frequently contain numeric constants. These can be specific values, or the result of an expression.

You can specify numeric constants using:

- Expressions.
- Execution address built-in functions.
- `ScatterAssert` function with load address related functions that take an expression as a parameter. An error message is generated if this expression does not evaluate to true.
- The symbol related function, `defined(`*`global_symbol_name`*`) ? `*`expr1`*` : `*`expr2`*.

Related concepts

7.35 Preprocessing of a scatter file on page 7-181.
8.18 Expression usage in scatter files on page 8-226.
8.19 Expression rules in scatter files on page 8-227.
8.20 Execution address built-in functions for use in scatter files on page 8-228.
8.22 ScatterAssert function and load address related functions on page 8-231.
8.23 Symbol related function in a scatter file on page 8-233.
8.24 Example of aligning a base address in execution space but still tightly packed in load space on page 8-234.

8.18 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.

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

8.17 Expression evaluation in scatter files on page 8-225.

8.19 Expression rules in scatter files on page 8-227.

8.20 Execution address built-in functions for use in scatter files on page 8-228.

8.22 ScatterAssert function and load address related functions on page 8-231.

8.23 Symbol related function in a scatter file on page 8-233.

8.10 Considerations when using a relative address +offset for load regions on page 8-215.

8.11 Considerations when using a relative address +offset for execution regions on page 8-216.

8.24 Example of aligning a base address in execution space but still tightly packed in load space on page 8-234.

Related references

8.2 Syntax of a scatter file on page 8-202.

8.4 Syntax of a load region description on page 8-204.

8.7 Syntax of an execution region description on page 8-208.

8.19 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

8.17 Expression evaluation in scatter files on page 8-225.
8.18 Expression usage in scatter files on page 8-226.
8.20 Execution address built-in functions for use in scatter files on page 8-228.
8.22 ScatterAssert function and load address related functions on page 8-231.
8.23 Symbol related function in a scatter file on page 8-233.
8.10 Considerations when using a relative address +offset for load regions on page 8-215.
8.11 Considerations when using a relative address +offset for execution regions on page 8-216.
8.24 Example of aligning a base address in execution space but still tightly packed in load space on page 8-234.

Related references

8.2 Syntax of a scatter file on page 8-202.
8.4 Syntax of a load region description on page 8-204.
8.7 Syntax of an execution region description on page 8-208.

8.20 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

8.10 Considerations when using a relative address +offset for load regions on page 8-215.

8.17 Expression evaluation in scatter files on page 8-225.

8.18 Expression usage in scatter files on page 8-226.

8.19 Expression rules in scatter files on page 8-227.

8.22 ScatterAssert function and load address related functions on page 8-231.

8.23 Symbol related function in a scatter file on page 8-233.

8.21 Scatter files containing relative base address load regions and a ZI execution region on page 8-230.

8.11 Considerations when using a relative address +offset for execution regions on page 8-216.

8.24 Example of aligning a base address in execution space but still tightly packed in load space on page 8-234.

Related references

8.2 Syntax of a scatter file on page 8-202.

8.4 Syntax of a load region description on page 8-204.

8.7 Syntax of an execution region description on page 8-208.

8.25 AlignExpr(expr, align) function on page 8-235.

6.4 Image$$ execution region symbols on page 6-107.

8.21 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

8.17 Expression evaluation in scatter files on page 8-225.

8.18 Expression usage in scatter files on page 8-226.

8.19 Expression rules in scatter files on page 8-227.

8.20 Execution address built-in functions for use in scatter files on page 8-228.

Related references

8.2 Syntax of a scatter file on page 8-202.

8.4 Syntax of a load region description on page 8-204.

8.7 Syntax of an execution region description on page 8-208.

6.4 Image$$ execution region symbols on page 6-107.

8.22 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

8.17 Expression evaluation in scatter files on page 8-225.
8.18 Expression usage in scatter files on page 8-226.
8.19 Expression rules in scatter files on page 8-227.
8.20 Execution address built-in functions for use in scatter files on page 8-228.
8.23 Symbol related function in a scatter file on page 8-233.
8.24 Example of aligning a base address in execution space but still tightly packed in load space on page 8-234.

Related references

8.2 Syntax of a scatter file on page 8-202.
8.4 Syntax of a load region description on page 8-204.
8.7 Syntax of an execution region description on page 8-208.
6.5 Load$$ execution region symbols on page 6-108.

8.23 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.

Examples

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

8.17 Expression evaluation in scatter files on page 8-225.

8.18 Expression usage in scatter files on page 8-226.

8.19 Expression rules in scatter files on page 8-227.

8.20 Execution address built-in functions for use in scatter files on page 8-228.

8.22 ScatterAssert function and load address related functions on page 8-231.

8.24 Example of aligning a base address in execution space but still tightly packed in load space on page 8-234.

Related references

8.2 Syntax of a scatter file on page 8-202.

8.4 Syntax of a load region description on page 8-204.

8.7 Syntax of an execution region description on page 8-208.

8.24 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

```
#! armcc -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 concepts

8.17 Expression evaluation in scatter files on page 8-225.

Related references

8.5 Load region attributes on page 8-205.
8.8 Execution region attributes on page 8-210.
8.26 GetPageSize() function on page 8-236.
8.27 SizeOfHeaders() function on page 8-237.
8.4 Syntax of a load region description on page 8-204.
8.7 Syntax of an execution region description on page 8-208.
8.25 AlignExpr(expr, align) function on page 8-235.

8.25 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 it is:

`0` mod `align`

Examples

This example aligns the address of `ER2` on an 8-byte boundary:

```
ER +0
{
    …
}
ER2 AlignExpr(+0x8000,8)
{
    …
}
```

Relationship with the `ALIGN` keyword

The following relationship exists between `ALIGN` and `AlignExpr`:

`ALIGN` keyword
: Load and execution regions already have an `ALIGN` keyword:

  - For load regions the `ALIGN` keyword aligns the base of the load region in load space and in the file to the specified alignment.
  - For execution regions the `ALIGN` keyword aligns the base of the execution region in execution and load space to the specified alignment.

`AlignExpr`
: Aligns the expression it operates on, but has no effect on the properties of the load or execution region.

Related references

8.8 Execution region attributes on page 8-210.

8.26 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 either the `--paged` or `--sysv` command-line option, returns the value of the internal page size that `armlink` uses in its alignment calculations. Otherwise, it return zero.

By default the internal page size is set to `8000`, but you can change it with the `--pagesize` command-line option.

Examples

This example aligns the base address of `ER` to a Page Boundary:

```
ER AlignExpr(+0, GetPageSize())
{
    …
}
```

Related concepts

8.24 Example of aligning a base address in execution space but still tightly packed in load space on page 8-234.

Related references

9.85 --pagesize=pagesize on page 9-333.

8.25 AlignExpr(expr, align) function on page 8-235.

8.27 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.

Examples

This example sets the base of `LR1` to start immediately after the ELF header and Program Headers:

```
LR1 SizeOfHeaders()
{
    …
}
```

Related concepts

8.24 Example of aligning a base address in execution space but still tightly packed in load space on page 8-234.

3.14 Linker support for creating demand-paged files on page 3-54.

7.33 Creation of regions on page boundaries on page 7-179.

Chapter 9
Linker Command-line Options

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

It contains the following sections:

9.1 --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

7.43 Behavior when .ANY sections overflow because of linker-generated content on page 7-195.

7.14 Placement of unassigned sections with the .ANY module selector on page 7-151.

Related references

9.2 --any_placement=algorithm on page 9-243.

9.53 --info=topic[,topic,...] on page 9-299.

9.3 --any_sort_order=order on page 9-245.

9.100 --scatter=filename on page 9-349.

8.16 Syntax of an input section description on page 8-221.

9.2 --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

7.15 Examples of using placement algorithms for .ANY sections on page 7-154.

7.16 Example of next_fit algorithm showing behavior of full regions, selectors, and priority on page 7-156.

7.14 Placement of unassigned sections with the .ANY module selector on page 7-151.

7.43 Behavior when .ANY sections overflow because of linker-generated content on page 7-195.

Related references

9.1 --any_contingency on page 9-242.

9.3 --any_sort_order=order on page 9-245.

9.53 --info=topic[,topic,...] on page 9-299.

9.100 --scatter=filename on page 9-349.

8.16 Syntax of an input section description on page 8-221.

9.3 --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`
: Sort input sections by command-line index.

By default, sections that have the same properties are resolved using the creation index. You can use the `--tiebreaker` command-line option to resolve sections by the order they appear on the linker command-line.

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 `descending_size`.

Related concepts

7.14 Placement of unassigned sections with the .ANY module selector on page 7-151.
7.17 Examples of using sorting algorithms for .ANY sections on page 7-158.

Related references

9.1 --any_contingency on page 9-242.
9.2 --any_placement=algorithm on page 9-243.
9.53 --info=topic[,topic,...] on page 9-299.
9.100 --scatter=filename on page 9-349.

9.4 --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 minimises 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:

32Mb
Execution region contains only ARM.

16Mb
Execution region contains Thumb code and the processor supports Thumb-2 technology.

4Mb
Execution region contains Thumb code and the processor does not support Thumb-2 technology.

Related concepts

3.16 Overview of veneers on page 3-57.

Related references

9.62 --largeregions, --no_largeregions on page 9-310.

9.5 --arm_only

Enables the linker to target the ARM instruction set only.

Usage

If the linker detects any objects requiring Thumb state, an error is generated.

Related references

9.108 --startup=symbol, --no_startup on page 9-358.

Related information

--arm compiler option.
--arm_only compiler option.
--thumb compiler option.
--arm assembler option.
--arm_only assembler option.
--thumb assembler option.

9.6 --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 concepts

7.20 Placement of __at sections at a specific address on page 7-163.
7.22 Automatic placement of __at sections on page 7-165.
7.23 Manual placement of __at sections on page 7-167.

Related references

8.2 Syntax of a scatter file on page 8-202.

9.7 --be8

Specifies ARMv6 Byte Invariant Addressing big-endian mode.

Usage

This is the default byte addressing mode for ARMv6 and later big-endian images. It means that the linker reverses the endianness of the instructions to give little-endian code and big-endian data for input objects that have been compiled or assembled as big-endian.

Byte Invariant Addressing mode is only available on ARM processors that support ARMv6 and above.

Related information

ARM Architecture Reference Manuals.

9.8 --be32

Specifies legacy Word Invariant Addressing big-endian mode. That is, identical to big-endian images prior to ARMv6.

Usage

This option produces big-endian code and data.

Word Invariant Addressing mode is the default mode for all pre-ARMv6 big-endian images.

Related information

ARM Architecture Reference Manuals.

9.9 --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`. This ensures that the code and data of the final image are the same regardless of whether you compile for debug or not. The smallest COMDAT groups are selected when linking, at the expense of a possibly slightly poorer debug illusion.

Examples

For two objects compiled with different optimization levels:

```
armcc -c -O2 file1.c
armcc -c -O0 file2.c
armlink --bestdebug fil1.o file2.o -o image.axf
```

Related concepts

4.1 Elimination of common debug sections on page 4-74.
4.2 Elimination of common groups or sections on page 4-75.
4.3 Elimination of unused sections on page 4-76.
4.4 Elimination of unused virtual functions on page 4-78.

Related references

9.81 --output=filename on page 9-329.

Related information

-c compiler option.
-Onum compiler option.

9.10 --blx_arm_thumb, --no_blx_arm_thumb

Enables the linker to use the `BLX` instruction for ARM to Thumb state changes.

Usage

If the linker cannot use `BLX` it must use an ARM to Thumb 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`.

Related concepts

3.18 Veneer types on page 3-59.

Related references

9.11 --blx_thumb_arm, --no_blx_thumb_arm on page 9-253.

9.11 --blx_thumb_arm, --no_blx_thumb_arm

Enables the linker to use the `BLX` instruction for Thumb to ARM state changes.

Usage

If the linker cannot use `BLX` it must use a Thumb to ARM 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`.

Note

Using `--no_blx_thumb_arm` prevents the possible issue with using a `BLX` (immediate) instruction on an ARM1176JZ-S or ARM1176JZF-S. See the *ARM1176JZ-S™ and ARM1176JZF-S™ Programmers Advice Notice Use of BLX (immediate)* for more details.

Related concepts

3.18 Veneer types on page 3-59.

Related references

9.10 --blx_arm_thumb, --no_blx_arm_thumb on page 9-252.

Related information

ARM1176JZ-S and ARM1176JZF-S Programmers Advice Notice Use of BLX (immediate) (ARM UAN 0002).

9.12 --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

4.14 About branches that optimize to a NOP on page 4-91.

Related references

9.56 --inline, --no_inline on page 9-304.

9.119 --tailreorder, --no_tailreorder on page 9-369.

9.13 --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 same name as the linked image. 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:

- `PROC` and `ENDP` directives.
- `FRAME PUSH` and `FRAME POP` directives.

The linker lists the following for each function `func`:

- Instruction set state for which the function is compiled (ARM or Thumb).
- Set of functions that call `func`.
- Set of functions that are called by `func`.
- Number of times the address of `func` is used in the image.

In addition, the callgraph identifies functions that are:

- Called through interworking veneers.
- Defined outside the image.
- Permitted to remain undefined (weak references).
- Called through a *Procedure Linkage Table* (PLT).
- Not called but still exist in the image.

The static callgraph also gives information about stack usage. It lists the:

- Size of the stack frame used by each function.
- Maximum size of the stack used by the function over any call sequence, that is, over any acyclic chain of function calls.

If there is a cycle, or if the linker detects a function with no stack size information in the call chain, `+ Unknown` is added to the stack usage. A reason is added to indicate why stack usage is unknown.

The linker reports missing stack frame information if there is no debug frame information for the function.

For indirect functions, the linker cannot reliably determine which function made the indirect call. This might affect how the maximum stack usage is calculated for a call chain. The linker lists all function pointers used in the image.

Use frame directives in assembly language code to describe how your code uses the stack. These directives ensure that debug frame information is present for debuggers to perform stack unwinding or profiling.

Default

The default is `--no_callgraph`.

Related references

9.14 --callgraph_file=filename on page 9-257.
9.15 --callgraph_output=fmt on page 9-258.
9.16 --cgfile=type on page 9-259.
9.17 --cgsymbol=type on page 9-260.
9.18 --cgundefined=type on page 9-261.
8.2 Syntax of a scatter file on page 8-202.

Related information

FRAME POP.
FRAME PUSH.
FUNCTION or PROC.
ENDFUNC or ENDP.

9.14 --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 same as the linked image.

Related references

9.13 --callgraph, --no_callgraph on page 9-255.
9.15 --callgraph_output=fmt on page 9-258.
9.16 --cgfile=type on page 9-259.
9.17 --cgsymbol=type on page 9-260.
9.18 --cgundefined=type on page 9-261.
9.81 --output=filename on page 9-329.

9.15 --callgraph_output=fmt

Controls the output format of the callgraph.

Syntax

`--callgraph_output=`*`fmt`*

Where *`fmt`* can be one of the following:

`html`
Outputs the callgraph in HTML format.

`text`
Outputs the callgraph in plain text format.

Default

The default is `--callgraph_output=html`.

Related references

9.13 --callgraph, --no_callgraph on page 9-255.
9.14 --callgraph_file=filename on page 9-257.
9.16 --cgfile=type on page 9-259.
9.17 --cgsymbol=type on page 9-260.
9.18 --cgundefined=type on page 9-261.

9.16 --cgfile=type

Controls the type of files to use for obtaining the symbols to be included in the callgraph.

Syntax

`--cgfile=`*`type`*

where *`type`* can be one of the following:

`all`
Includes symbols from all files.

`user`
Includes only symbols from user defined objects and libraries.

`system`
Includes only symbols from system libraries.

Default

The default is `--cgfile=all`.

Related references

9.13 --callgraph, --no_callgraph on page 9-255.
9.14 --callgraph_file=filename on page 9-257.
9.15 --callgraph_output=fmt on page 9-258.
9.17 --cgsymbol=type on page 9-260.
9.18 --cgundefined=type on page 9-261.

9.17 --cgsymbol=type

Controls what symbols are included in the callgraph.

Syntax

`--cgsymbol=`*`type`*

Where *`type`* can be one of the following:

`all`
: Includes both local and global symbols.

`locals`
: Includes only local symbols.

`globals`
: Includes only global symbols.

Default

The default is `--cgsymbol=all`.

Related references

9.13 --callgraph, --no_callgraph on page 9-255.
9.14 --callgraph_file=filename on page 9-257.
9.15 --callgraph_output=fmt on page 9-258.
9.16 --cgfile=type on page 9-259.
9.18 --cgundefined=type on page 9-261.

9.18 --cgundefined=type

Controls what undefined references are included in the callgraph.

Syntax

`--cgundefined=`*`type`*

Where *`type`* can be one of the following:

`all`
: Includes both function entries and calls to undefined weak references.

`entries`
: Includes function entries for undefined weak references.

`calls`
: Includes calls to undefined weak references.

`none`
: Omits all undefined weak references from the output.

Default

The default is `--cgundefined=all`.

Related references

9.13 --callgraph, --no_callgraph on page 9-255.
9.14 --callgraph_file=filename on page 9-257.
9.15 --callgraph_output=fmt on page 9-258.
9.16 --cgfile=type on page 9-259.
9.17 --cgsymbol=type on page 9-260.

9.19 --combreloc, --no_combreloc

Enables or disables the linker reordering of dynamic relocations so that a dynamic loader can process them more efficiently.

`--combreloc` is the more efficient option.

Default

The default is `--combreloc`.

9.20 --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

4.17 Linker merging of comment sections on page 4-94.

Related references

9.46 --filtercomment, --no_filtercomment on page 9-291.

9.21 --compress_debug, --no_compress_debug

Causes the linker to compress `.debug_*` sections, if it is sensible to do so.

Usage

This removes some redundancy and reduces debug table size. Using `--compress_debug` can significantly increase the time required to link an image. Debug compression can only be performed on DWARF3 debug data, not DWARF2.

Default

The default is `--no_compress_debug`.

Related information

The DWARF Debugging Standard.

9.22 --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_`, 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.

`--no_cppinit` means do not add a reference.

Related references

9.90 --ref_cpp_init, --no_ref_cpp_init on page 9-339.

Related information

Initialization of the execution environment and execution of the application.

C++ initialization, construction and destruction.

9.23 --cpu=list

Lists the architecture and processor names that are supported by the `--cpu=name` option.

Syntax

```
--cpu=list
```

Related references

9.24 --cpu=name on page 9-267.
9.50 --fpu=list on page 9-295.
9.51 --fpu=name on page 9-296.

9.24 --cpu=name

Enables code generation for the selected ARM processor or architecture.

Syntax

`--cpu=`*name*

Where *name* is the name of a processor or architecture:

- If *name* is the name of a processor, enter it as shown on ARM data sheets, for example, `ARM7TDMI`, `ARM1176JZ-S`, `MPCore`.
- If *name* is the name of an architecture, it must belong to the list of architectures shown in the following table.

Processor and architecture names are not case-sensitive.

Wildcard characters are not accepted.

Table 9-1 Supported ARM architectures

Architecture	Description	Example processors
`4`	ARMv4 without Thumb	SA-1100
`4T`	ARMv4 with Thumb	ARM7TDMI, ARM9TDMI, ARM720T, ARM740T, ARM920T, ARM922T, ARM940T, SC100
`5T`	ARMv5 with Thumb and interworking	-
`5TE`	ARMv5 with Thumb, interworking, DSP multiply, and double-word instructions	ARM9E, ARM946E-S, ARM966E-S
`5TEJ`	ARMv5 with Thumb, interworking, DSP multiply, double-word instructions, and Jazelle® extensions **Note** `armlink` cannot generate Java bytecodes.	ARM926EJ-S, ARM1026EJ-S, SC200
`6`	ARMv6 with Thumb, interworking, DSP multiply, double-word instructions, unaligned and mixed-endian support, Jazelle, and media extensions	ARM1136J-S, ARM1136JF-S
`6-M`	ARMv6 micro-controller profile with Thumb only, plus processor state instructions	Cortex-M1 without OS extensions, Cortex-M0, SC000, Cortex-M0plus
`6S-M`	ARMv6 micro-controller profile with Thumb only, plus processor state instructions and OS extensions	Cortex-M1 with OS extensions
`6K`	ARMv6 with SMP extensions	MPCore
`6T2`	ARMv6 with Thumb (Thumb-2 technology)	ARM1156T2-S, ARM1156T2F-S
`6Z`	ARMv6 with Security Extensions	ARM1176JZF-S, ARM1176JZ-S
`7`	ARMv7 with Thumb (Thumb-2 technology) only, and without hardware divide	-
`7-R`	ARMv7 real-time profile with ARM, Thumb (Thumb-2 technology), DSP support, and 32-bit SIMD support	Cortex-R4, Cortex-R4F, Cortex-R7

Table 9-1 Supported ARM architectures (continued)

Architecture	Description	Example processors
`7-M`	ARMv7 micro-controller profile with Thumb (Thumb-2 technology) only and hardware divide	Cortex-M3, SC300
`7E-M`	ARMv7-M enhanced with DSP (saturating and 32-bit SIMD) instructions	Cortex-M4

Note

- ARMv7 is not an actual ARM architecture. `--cpu=7` denotes the features that are common to the ARMv7-A, ARMv7-R, and ARMv7-M architectures. By definition, any given feature used with `--cpu=7` exists on the ARMv7-A, ARMv7-R, and ARMv7-M architectures.
- `7-A.security` is not an actual ARM architecture, but rather, refers to `7-A` plus Security Extensions.

Usage

The following general points apply to processor and architecture options:

Processors

- Selecting the processor selects the appropriate architecture, *Floating-Point Unit* (FPU), and memory organization.
- The supported `--cpu` values include all current ARM product names or architecture versions.

  Other ARM architecture-based processors, such as the Marvell Feroceon and the Marvell XScale, are also supported.
- If you specify a processor for the `--cpu` option, the generated code is optimized for that processor.

Architectures

- If you specify an architecture name for the `--cpu` option, the generated code can run on any processor supporting that architecture. For example, `--cpu=5TE` produces code that can be used by the ARM926EJ-S® processor.

FPU

- Some specifications of `--cpu` imply an `--fpu` selection.

  Note

  Any explicit FPU, set with `--fpu` on the command line, overrides an *implicit* FPU.
- If no `--fpu` option is specified and no `--cpu` option is specified, `--fpu=softvfp` is used.

Default

`armlink` assumes `--cpu=ARM7TDMI` if you do not specify a `--cpu` option.

To obtain a full list of architectures and processors, use the `--cpu=list` option.

Restrictions

You cannot specify both a processor and an architecture on the same command-line.

Related references

9.23 --cpu=list on page 9-266.

9.50 --fpu=list on page 9-295.
9.51 --fpu=name on page 9-296.

Related information

Types of floating-point linkage.
Compiler options for floating-point linkage and computations.
Floating-point linkage and computational requirements of compiler options.
Processors and their implicit Floating-Point Units (FPUs).

9.25 --crosser_veneershare, --no_crosser_veneershare

Enables or disables veneer sharing across execution regions.

Usage

The default is `--crosser_veneershare`, and enables veneer sharing across execution regions.

`--no_crosser_veneershare` prohibits veneer sharing across execution regions.

Related references

9.130 --veneershare, --no_veneershare on page 9-380.

9.26 --datacompressor=opt

Enables you to specify one of the supplied algorithms for RW data compression.

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 9-2 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

4.9 Options available to override the compression algorithm used by the linker on page 4-85.
4.10 How compression is applied on page 4-86.
4.11 Considerations when working with RW data compression on page 4-87.
4.8 How the linker chooses a compressor on page 4-84.

9.27 --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.

9.28 --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.
- `warning`, to treat all warnings as errors.

Related references

9.29 --diag_remark=tag[,tag,...] on page 9-274.
9.30 --diag_style=arm|ide|gnu on page 9-275.
9.31 --diag_suppress=tag[,tag,...] on page 9-276.
9.32 --diag_warning=tag[,tag,...] on page 9-277.
9.40 --errors=filename on page 9-285.
9.109 --strict on page 9-359.

9.29 --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.

Related references

9.28 --diag_error=tag[,tag,...] on page 9-273.
9.30 --diag_style=arm|ide|gnu on page 9-275.
9.31 --diag_suppress=tag[,tag,...] on page 9-276.
9.32 --diag_warning=tag[,tag,...] on page 9-277.
9.40 --errors=filename on page 9-285.
9.92 --remarks on page 9-341.
9.109 --strict on page 9-359.

9.30 --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 ARM compiler style.

`ide`
: Include the line number and character count for any line that is in error. These values are displayed in parentheses.

`gnu`
: Display messages in the format used by `gcc`.

Default

The default is `--diag_style=arm`.

Usage

`--diag_style=gnu` matches the format reported by the GNU Compiler, `gcc`.

`--diag_style=ide` matches the format reported by Microsoft Visual Studio.

Related references

9.28 --diag_error=tag[,tag,...] on page 9-273.
9.29 --diag_remark=tag[,tag,...] on page 9-274.
9.31 --diag_suppress=tag[,tag,...] on page 9-276.
9.32 --diag_warning=tag[,tag,...] on page 9-277.
9.40 --errors=filename on page 9-285.
9.92 --remarks on page 9-341.
9.109 --strict on page 9-359.

9.31 --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.
- `error`, to suppress all errors that can be downgraded.
- `warning`, to suppress all warnings.

Examples

To suppress the warning messages that have numbers `L6314W` and `L6305W`, use the following command:

```
armlink --diag_suppress=L6314,L6305 …
```

Related references

9.28 --diag_error=tag[,tag,...] on page 9-273.
9.29 --diag_remark=tag[,tag,...] on page 9-274.
9.30 --diag_style=arm|ide|gnu on page 9-275.
9.32 --diag_warning=tag[,tag,...] on page 9-277.
9.40 --errors=filename on page 9-285.
9.109 --strict on page 9-359.
9.92 --remarks on page 9-341.

9.32 --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.
- `error`, to set all errors that can be downgraded to warnings.

Related references

9.28 --diag_error=tag[,tag,...] on page 9-273.
9.29 --diag_remark=tag[,tag,...] on page 9-274.
9.30 --diag_style=arm|ide|gnu on page 9-275.
9.31 --diag_suppress=tag[,tag,...] on page 9-276.
9.40 --errors=filename on page 9-285.
9.109 --strict on page 9-359.
9.92 --remarks on page 9-341.

9.33 --eager_load_debug, --no_eager_load_debug

Keeps or removes debug section data.

Usage

The `--no_eager_load_debug` option causes the linker to remove debug section data from memory after object loading. This lowers the peak memory usage of the linker at the expense of some linker performance, because much of the debug data has to be loaded again when the final image is written.

Using `--no_eager_load_debug` option does not affect the debug data that is written into the ELF file.

The default is `--eager_load_debug`.

Note

If you use some command-line options, such as `--map`, the resulting image or object built without debug information might differ by a small number of bytes. This is because the `.comment` section contains the linker command line used, where the options have differed from the default. Therefore `--no_eager_load_debug` images are a little larger and contain Program Header and possibly a section header a small number of bytes later. Use `--no_comment_section` to eliminate this difference.

Related references

9.20 --comment_section, --no_comment_section on page 9-263.

9.34 --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

6.19 What is a steering file? on page 6-124.
6.23 Hide and rename global symbols with a steering file on page 6-128.

Related references

10 Linker Steering File Command Reference on page 10-391.

9.35 --emit_debug_overlay_relocs

Outputs only relocations of debug sections with respect to overlaid program sections to aid an overlay-aware debugger.

Related references

9.36 --emit_debug_overlay_section on page 9-281.

9.38 --emit_relocs on page 9-283.

9.37 --emit_non_debug_relocs on page 9-282.

Related information

ABI for the ARM Architecture: Support for Debugging Overlaid Programs.

9.36 --emit_debug_overlay_section

Resolves debug overlay sections during static linking.

Usage

In a relocatable file, a debug section refers to a location in a program section by way of a relocated location. A reference from a debug section to a location in a program section has the following format:

```
<debug_section_index, debug_section_offset>, <program_section_index,
program_section_offset>
```

During static linking the pair of *program* values is reduced to single value, the execution address. This is ambiguous in the presence of overlaid sections.

To resolve this ambiguity, use this option to output a `.ARM.debug_overlay` section of type `SHT_ARM_DEBUG_OVERLAY = SHT_LOUSER + 4` containing a table of entries as follows:

debug_section_offset, debug_section_index, program_section_index

Related references

9.35 --emit_debug_overlay_relocs on page 9-280.
9.38 --emit_relocs on page 9-283.

Related information

ABI for the ARM Architecture: Support for Debugging Overlaid Programs.

9.37 --emit_non_debug_relocs

Retains only relocations from non-debug sections in an executable file.

Related references

9.38 --emit_relocs on page 9-283.

9.38 --emit_relocs

Retains all relocations in the executable file.

Usage

This results in larger executable files.

This is equivalent to the GNU ld `--emit-relocs` option.

Related references

9.35 --emit_debug_overlay_relocs on page 9-280.

9.37 --emit_non_debug_relocs on page 9-282.

Related information

ABI for the ARM Architecture: Support for Debugging Overlaid Programs.

9.39 --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 Thumb 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 Thumb 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 ases, a unique initial entry point must be specified for an image, otherwise the error L6305E is generated. The initial entry point specified with the `--entry` option is stored in the executable file header for use by the loader. There can be only one occurrence of this option on the command line. A debugger typically uses this entry address to initialize the *Program Counter* (PC) when an image is loaded. The initial entry point must meet the following conditions:

- The image entry point must lie within an execution region.
- The execution region must be non-overlay, and must be a root execution region (load address == execution address).

Related references

9.108 --startup=symbol, --no_startup on page 9-358.

Related information

ENTRY directive.

9.40 --errors=filename

Redirects the diagnostics from the standard error stream to a specified file.

Syntax

`--errors=filename`

Usage

The specified file is created at the start of the link stage. If a file of the same name already exists, it is overwritten.

If *filename* is specified without path information, the file is created in the current directory.

Related references

9.28 --diag_error=tag[,tag,...] on page 9-273.
9.29 --diag_remark=tag[,tag,...] on page 9-274.
9.30 --diag_style=arm|ide|gnu on page 9-275.
9.31 --diag_suppress=tag[,tag,...] on page 9-276.
9.32 --diag_warning=tag[,tag,...] on page 9-277.
9.92 --remarks on page 9-341.

9.41 --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`.

Related concepts

3.21 Command-line options used to control the generation of C++ exception tables on page 3-62.

9.42 --exceptions_tables=action

Specifies how exception tables are generated for objects that do not already contain exception unwinding tables.

Syntax

`--exceptions_tables=`*`action`*

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

`nocreate`
The linker does not create missing exception tables.

`unwind`
The linker creates an unwinding table for each section in your image that does not already have an exception table.

`cantunwind`
The linker creates a nounwind table for each section in your image that does not already have an exception table.

Default

The default is `--exceptions_tables=nocreate`.

Related concepts

3.21 Command-line options used to control the generation of C++ exception tables on page 3-62.

9.43 --feedback=filename

Generates a feedback file for input to the compiler. This file informs the compiler about unused functions.

Syntax

`--feedback=`*`filename`*

Usage

During your next compilation, use the compiler option `--feedback=`*`filename`* to specify the feedback file to use. Unused functions are then placed in their own sections for possible future elimination by the linker.

Related concepts

4.5 About linker feedback on page 4-79.

Related references

9.44 --feedback_image=option on page 9-289.

9.45 --feedback_type=type on page 9-290.

Related information

--feedback=filename compiler option.

9.44 --feedback_image=option

Changes the behavior of the linker when writing a feedback file with scatter-loading.

Syntax

`--feedback_image=option`

Where *option* is one of the following:

none
Uses the scatter file to determine region size limits. Disables region overlap and region size overflow messages. Does not write an ELF image. Error messages are still produced if a region overflows the 32-bit address space.

noerrors
Uses the scatter file to determine region size limits. Warns on region overlap and region size overflow messages. Writes an ELF image, which might not be executable. Error messages are still produced if a region overflows the 32-bit address space.

simple
Ignores the scatter file. Disables ROPI/RWPI errors and warnings. Writes an ELF image, which might not be executable.

full
Enables all error and warning messages and writes a valid ELF image.

Usage

Use this option to produce a feedback file where an executable ELF image cannot be created. That is, when your code does not fit into the region limits described in your scatter file before unused functions are removed using linker feedback.

Default

The default option is `--feedback_image=full`.

Related concepts

4.5 About linker feedback on page 4-79.

Related references

9.43 --feedback=filename on page 9-288.
9.45 --feedback_type=type on page 9-290.
9.100 --scatter=filename on page 9-349.

Related information

--feedback=filename compiler option.

9.45 --feedback_type=type

Controls the information that the linker puts into the feedback file.

Syntax

`--feedback_type=`*`type`*

Where *`type`* is a comma-separated list from the following topic keywords:

`[no]iw`
Controls functions that require interworking support.

`[no]unused`
Controls unused functions in the image.

Default

The default option is `--feedback_type=unused,noiw`.

Related concepts

4.5 About linker feedback on page 4-79.

Related references

9.43 --feedback=filename on page 9-288.
9.44 --feedback_image=option on page 9-289.

Related information

--apcs compiler option.
--feedback=filename compiler option.

9.46 --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.

Default

The default is `--filtercomment`.

Related concepts

4.17 Linker merging of comment sections on page 4-94.

Related references

9.20 --comment_section, --no_comment_section on page 9-263.

9.47 --fini=symbol

Specifies the symbol name to use to define the entry point for finalization code.

Syntax

`--fini=`*`symbol`*

Where *`symbol`* is the symbol name to use for the entry point to the finalization code.

Usage

The dynamic linker executes this code when it unloads the executable file or shared object.

Related references

9.55 --init=symbol on page 9-303.

9.48 --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`*. You must not specify a symbol that has more than one definition, because only one section can be placed first. For example: `--first=reset`.

`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`*. If you use this short form and there is more than one input section, the linker generates an error message. For example: `--first=init.o`.

Usage

The `--first` option cannot be used with `--scatter`. Instead, use the `+FIRST` attribute in a scatter file.

Related concepts

3.12 Section placement with the FIRST and LAST attributes on page 3-52.

3.11 Section placement with the linker on page 3-50.

Related references

9.63 --last=section_id on page 9-311.

9.100 --scatter=filename on page 9-349.

9.49 --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 one of two situations:

- The ELF file contains instructions from architecture *`archtype`* yet the build attributes claim that *`archtype`* is not supported.
- The build attributes are inconsistent enough that the linker cannot map them to an existing CPU.

If setting the `--cpu` option still fails, use `--force_explicit_attr` to cause the linker to retry the CPU mapping using build attributes constructed from `--cpu=`*`archtype`*. This might help if the error is being given solely because of inconsistent build attributes.

Related references

9.24 --cpu=name on page 9-267.
9.51 --fpu=name on page 9-296.

9.50 --fpu=list

Lists the FPU architectures that are supported by the --fpu=name option.

Deprecated options are not listed.

Related references

9.23 --cpu=list on page 9-266.
9.24 --cpu=name on page 9-267.
9.51 --fpu=name on page 9-296.

9.51 --fpu=name

Specifies the target FPU architecture.

To obtain a full list of FPU architectures use the `--fpu=list` option.

Syntax

`--fpu=`*name*

Where *name* is one of:

`vfpv3`
: Selects a hardware vector floating-point unit conforming to architecture VFPv3. VFPv3 is backwards compatible with VFPv2 except that VFPv3 cannot trap floating-point exceptions.

`vfpv3_fp16`
: Selects a hardware vector floating-point unit conforming to architecture VFPv3 that also provides the half-precision extensions.

`vfpv3_d16`
: Selects a hardware vector floating-point unit conforming to VFPv3-D16 architecture.

`vfpv3_d16_fp16`
: Selects a hardware vector floating-point unit conforming to VFPv3-D16 architecture, that also provides the half-precision extensions.

`vfpv4`
: Selects a hardware floating-point unit conforming to the VFPv4 architecture.

`vfpv4_d16`
: Selects a hardware floating-point unit conforming to the VFPv4-D16 architecture.

`fpv4-sp`
: Selects a hardware floating-point unit conforming to the single precision variant of the FPv4 architecture.

`softvfp`
: Selects software floating-point support where floating-point operations are performed by a floating-point library, `fplib`. This is the default if you do not specify a `--fpu` option, or if you select a CPU that does not have an FPU.

`softvfp+vfpv3`
: Selects a hardware vector floating-point unit conforming to VFPv3, with software floating-point linkage. Select this option if you are interworking Thumb code with ARM code on a system that implements a VFPv3 unit.

`softvfp+vfpv3_fp16`
: Selects a hardware vector floating-point unit conforming to VFPv3-fp16, with software floating-point linkage.

`softvfp+vfpv3_d16`
: Selects a hardware vector floating-point unit conforming to VFPv3-D16, with software floating-point linkage.

`softvfp+vfpv3_d16_fp16`
: Selects a hardware vector floating-point unit conforming to VFPv3-D16-fp16, with software floating-point linkage.

`softvfp+vfpv4`
: Selects a hardware floating-point unit conforming to FPv4, with software floating-point linkage.

`softvfp+vfpv4_d16`
: Selects a hardware floating-point unit conforming to VFPv4-D16, with software floating-point linkage.

`softvfp+fpv4-sp`
: Selects a hardware floating-point unit conforming to FPv4-SP, with software floating-point linkage.

Related references

9.23 --cpu=list on page 9-266.
9.24 --cpu=name on page 9-267.
9.50 --fpu=list on page 9-295.

Related information

Types of floating-point linkage.
Compiler options for floating-point linkage and computations.
Floating-point linkage and computational requirements of compiler options.
Processors and their implicit Floating-Point Units (FPUs).

9.52 --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 references

9.132 --version_number on page 9-382.
9.135 --vsn on page 9-385.
9.102 --show_cmdline on page 9-351.

9.53 --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 sections 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 they are assigned by the placement algorithm.
- Information about the contingency space and policy 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 CPU, 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
Lists all functions that are inlined by the linker, and the total number of inlines if `--inline` is used.

inputs
Lists the input symbols, objects and libraries.

libraries
Lists the full path name of every library automatically selected for the link stage.

You can use this option with `--info_lib_prefix` to display information about a specific library.

merge
Lists the **const** strings that are merged by the linker. Each item lists the merged result, the strings being merged, and the associated object files.

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 have been removed by unused section elimination.

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 or not 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

7.14 Placement of unassigned sections with the .ANY module selector on page 7-151.
4.3 Elimination of unused sections on page 4-76.
4.11 Considerations when working with RW data compression on page 4-87.
4.7 Optimization with RW data compression on page 4-83.
4.8 How the linker chooses a compressor on page 4-84.
4.10 How compression is applied on page 4-86.

Related references

9.1 --any_contingency on page 9-242.
9.2 --any_placement=algorithm on page 9-243.
9.3 --any_sort_order=order on page 9-245.
9.54 --info_lib_prefix=opt on page 9-302.
9.79 --merge, --no_merge on page 9-327.
9.128 --veneer_inject_type=type on page 9-378.
5.1 Options for getting information about linker-generated files on page 5-96.
9.26 --datacompressor=opt on page 9-271.

9.56 --inline, --no_inline on page 9-304.
9.93 --remove, --no_remove on page 9-342.
9.119 --tailreorder, --no_tailreorder on page 9-369.
8.8 Execution region attributes on page 8-210.

9.54 --info_lib_prefix=opt

Specifies a filter for the `--info=libraries` option. The linker only displays the libraries that have the same prefix as the filter.

Syntax

`--info=libraries --info_lib_prefix=opt`

Where *opt* is the prefix of the required library.

Examples

- Displaying a list of libraries without the filter:

```
armlink --info=libraries test.o
```

  Produces a list of libraries, for example:

```
install_directory\lib\armlib\c_4.l
install_directory\lib\armlib\fz_4s.l install_directory\lib\armlib\h_4.l
install_directory\lib\armlib\m_4s.l
install_directory\lib\armlib\vfpsupport.l
```

- Displaying a list of libraries with the filter:

```
armlink --info=libraries --info_lib_prefix=c test.o
```

  Produces a list of libraries with the specified prefix, for example:

```
install_directory\lib\armlib\c_4.l
```

Related references

9.53 --info=topic[,topic,...] on page 9-299.

9.55 --init=symbol

Specifies a symbol name to use for the initialization code. A dynamic linker executes this code when it loads the executable file or shared object.

Syntax

`--init=`*`symbol`*

Where *`symbol`* is the symbol name you want to use to define the location of the initialization code.

Related references

9.47 --fini=symbol on page 9-292.

9.56 --inline, --no_inline

Enables or disables branch inlining to optimize small function calls in your image.

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 C Library by default.

Related concepts

4.12 Function inlining with the linker on page 4-88.

Related references

9.12 --branchnop, --no_branchnop on page 9-254.
9.57 --inline_type=type on page 9-305.
9.119 --tailreorder, --no_tailreorder on page 9-369.

9.57 --inline_type=type

Inlines functions from all objects, ARM C Library only, or turns of 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 C Library.

none
The linker is not permitted to inline functions.

This option takes precedence over `--inline` if both options are present on the command line. The mapping between the options is:

- `--inline` maps to `--inline_type=all`
- `--no_inline` maps to `--inline_type=library`

Note

To disable linker inlining completely you must use `--inline_type=none`.

Related references

9.56 --inline, --no_inline on page 9-304.
9.119 --tailreorder, --no_tailreorder on page 9-369.

9.58 --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

3.18 Veneer types on page 3-59.

3.16 Overview of veneers on page 3-57.

3.17 Veneer sharing on page 3-58.

3.19 Generation of position independent to absolute veneers on page 3-60.

3.20 Reuse of veneers when scatter-loading on page 3-61.

Related references

9.87 --piveneer, --no_piveneer on page 9-335.

9.130 --veneershare, --no_veneershare on page 9-380.

9.59 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 in order 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

6.14 Access symbols in another image on page 6-118.

3.23 How the linker performs library searching, selection, and scanning on page 3-66.

Related references

9.99 --scanlib, --no_scanlib on page 9-348.

9.60 --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-sensitive match for the string Premier.

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

3.1 The structure of an ARM ELF image on page 3-34.

9.61 --keep_protected_symbols

Keeps STV_PROTECTED symbols even if you are not using dynamic linking.

Examples

For example, your application might export functions provided by an API to shared objects that are loaded using a custom loader. However, the linker unused section elimination optimization causes the sections to be removed, even if those sections include STV_PROTECTED symbols. To prevent section containing STV_PROTECTED symbols from being removed, specify `--keep_protected_symbols`.

Related concepts

3.23 How the linker performs library searching, selection, and scanning on page 3-66.

4.3 Elimination of unused sections on page 4-76.

Related references

9.78 --max_visibility=type on page 9-326.

9.82 --override_visibility on page 9-330.

9.62 --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 place 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:

32Mb
Execution region contains only ARM instructions.

16Mb
Execution region contains Thumb instructions and the processor supports Thumb-2 technology.

4Mb
Execution region contains Thumb instructions and the processor does not support Thumb-2 technology.

Related concepts

3.16 Overview of veneers on page 3-57.
3.17 Veneer sharing on page 3-58.
3.18 Veneer types on page 3-59.
3.19 Generation of position independent to absolute veneers on page 3-60.

Related references

9.4 --api, --no_api on page 9-246.
9.106 --sort=algorithm on page 9-355.
9.128 --veneer_inject_type=type on page 9-378.
9.127 --veneerinject,--no_veneerinject on page 9-377.

9.63 --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*. If 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.

Examples

This option can force an input section that contains a checksum to be placed last in the RW section.

Related concepts

3.12 Section placement with the FIRST and LAST attributes on page 3-52.

3.11 Section placement with the linker on page 3-50.

Related references

9.48 --first=section_id on page 9-293.

9.100 --scatter=filename on page 9-349.

9.64 --ldpartial

Enables you to link a partial object and combine sections in the output object.

Usage

You can control how the sections are combined with a scatter file.

`-r` is a synonym for `--ldpartial`.

——— **Note** ———

This option contrasts with the `--partial` option that does not combine sections.

———————————

9.65 --legacyalign, --no_legacyalign

Controls how padding is inserted into the image.

Usage

By default, the linker assumes execution regions and load regions to be four-byte aligned. `--legacyalign` 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.

Related concepts

3.11 Section placement with the linker on page 3-50.
7.37 Example of using expression evaluation in a scatter file to avoid padding on page 7-184.

Related references

8.5 Load region attributes on page 8-205.
8.8 Execution region attributes on page 8-210.

9.66 --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`*.

Usage

You can also use the `ARMCC5LIB` environment variable to specify the path for the parent directory containing the ARM libraries. Any paths specified with `--libpath` override the path specified by the environment variable.

Note

This option does not affect searches for user libraries. Use `--userlibpath` instead for user libraries.

Related concepts

3.23 How the linker performs library searching, selection, and scanning on page 3-66.

Related references

9.126 --userlibpath=pathlist on page 9-376.

Related information

Toolchain environment variables.

9.67 --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 runtime libraries are selected at link time. This is the default.

`microlib`
Specifies that the *C micro-library* (microlib) is selected at link time.

Usage

Use this option:

- When use of the libraries require more specialized optimizations.
- With the linker to override all other `--library_type` options.

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.

9.68 --licretry

If you are using floating licenses, this option makes up to 10 attempts to obtain a license when you invoke `armlink`.

Usage

Use this option if your builds are failing to obtain a license from your license server, and only after you have ruled out any other problems with the network or the license server setup.

It is recommended that you place this option in the `ARMCC5_LINKOPT` environment variable. In this way, you do not have to modify your build files.

Related information

Toolchain environment variables.

ARM DS-5 License Management Guide.

9.69 --list=filename

Redirects diagnostic output to a file.

Syntax

`--list=`*`filename`*

Where *`filename`* is the file to use to save the diagnostic output. *`filename`* can include a path

Usage

Redirects the diagnostics output by the `--info`, `--map`, `--symbols`, `--verbose`, `--xref`, `--xreffrom`, and `--xrefto` options to *`file`*.

The specified file is created when diagnostics are output. If a file of the same name already exists, it is overwritten. However, if diagnostics are not output, a file is not created. In this case, the contents of any existing file with the same name remain unchanged.

If *`filename`* is specified without a path, it is created in the output directory, that is, the directory where the output image is being written.

Related references

9.74 --map, --no_map on page 9-322.
9.131 --verbose on page 9-381.
9.137 --xref, --no_xref on page 9-387.
9.138 --xrefdbg, --no_xrefdbg on page 9-388.
9.139 --xref{from|to}=object(section) on page 9-389.
9.53 --info=topic[,topic,...] on page 9-299.
9.117 --symbols, --no_symbols on page 9-367.

9.70 --list_mapping_symbols, --no_list_mapping_symbols

Enables or disables the addition of mapping symbols in the output produced by `--symbols`.

The mapping symbols `$a`, `$d`, `$t`, and `$t.x` flag transitions between ARM code, Thumb code, and data.

Default

The default is `--no_list_mapping_symbols`.

Related concepts

6.1 About mapping symbols on page 6-104.

Related references

9.117 --symbols, --no_symbols on page 9-367.

Related information

ELF for the ARM Architecture.

9.71 --load_addr_map_info, --no_load_addr_map_info

Includes load addresses for execution regions 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 the `--map` with this option.

Examples

The following example shows the format of the map file output:

```
    Base Addr    Load Addr    Size         Type   Attr      Idx    E Section Name
Object
    0x00008000   0x00008000   0x00000008   Code   RO           25  * !!!main
__main.o(c_4.l)
    0x00010000   COMPRESSED   0x00001000   Data   RW            2    dataA
data.o
    0x00003000         -      0x00000004   Zero   RW            2    .bss
test.o
```

Related references

9.74 --map, --no_map on page 9-322.

9.72 --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`.

9.73 --mangled, --unmangled

Instructs the linker to display mangled or unmangled C++ symbol names in diagnostic messages, and in listings produced by the `--xref`, `--xreffrom`, `--xrefto`, and `--symbols` options.

Usage

If `--unmangled` is selected, C++ symbol names are displayed as they appear in your source code.

If `--mangled` is selected, C++ symbol names are displayed as they appear in the object symbol tables.

Default

The default is `--unmangled`.

Related references

9.75 --match=crossmangled on page 9-323.
9.117 --symbols, --no_symbols on page 9-367.
9.137 --xref, --no_xref on page 9-387.
9.138 --xrefdbg, --no_xrefdbg on page 9-388.
9.139 --xref{from|to}=object(section) on page 9-389.

9.74 --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 tasks

5.5 How to find the location of a symbol within the map file on page 5-101.

Related references

9.69 --list=filename on page 9-317.

9.71 --load_addr_map_info, --no_load_addr_map_info on page 9-319.

9.101 --section_index_display=type on page 9-350.

9.75 --match=crossmangled

Instructs the linker to match the combinations of mangled and unmangled symbol references and definitions.

Usage

Matches:

- A reference to an unmangled symbol with the mangled definition.
- A reference to a mangled symbol with the unmangled definition.

Libraries and matching combinations operate as follows:

- If the library members define a mangled definition, and there is an unresolved unmangled reference, the member is loaded to satisfy it.
- If the library members define an unmangled definition, and there is an unresolved mangled reference, the member is loaded to satisfy it.

Note

This option has no effect if used with partial linking. The partial object contains all the unresolved references to unmangled symbols, even if the mangled definition exists. Matching is done only in the final link step.

Related references

9.73 --mangled, --unmangled on page 9-321.

9.76 --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 concepts

7.22 Automatic placement of __at sections on page 7-165.

9.77 --max_veneer_passes=value

Specifies a limit to the number of veneer generation passes the linker attempts to make when certain conditions are met.

Syntax

`--max_veneer_passes=`*`value`*

Where *`value`* is the maximum number of veneer passes the linker is to attempt. The minimum value you can specify is one.

Usage

The linker applies this limit when both the following conditions are met:

- A section that is sufficiently large has a relocation that requires a veneer.
- The linker cannot place the veneer close enough to the call site.

The linker attempts to diagnose the failure if the maximum number of veneer generation passes you specify is exceeded, and displays a warning message. You can downgrade this warning message using `--diag_remark`.

Default

The default number of passes is 10.

Related references

9.29 --diag_remark=tag[,tag,...] on page 9-274.
9.32 --diag_warning=tag[,tag,...] on page 9-277.

9.78 --max_visibility=type

Controls the visibility of all symbol definitions.

Syntax

`--max_visibility=`*`type`*

Where *`type`* can be one of:

`default`
Default visibility.

`protected`
Protected visibility.

Usage

Use`--max_visibility=protected` to limit the visibility of all symbol definitions. Global symbol definitions that normally have default visibility, are given protected visibility when this option is specified.

Default

The default is `--max_visibility=default`.

Related references

9.61 --keep_protected_symbols on page 9-309.
9.82 --override_visibility on page 9-330.

9.79 --merge, --no_merge

Enables or disables the merging of **const** strings that are placed in shareable sections by the compiler.

Usage

Using `--merge` can reduce the size of the image if there are similarities between **const** strings.

Use `--info=merge` to see a listing of the merged **const** strings.

By default, merging happens between different load and execution regions. Therefore, code from one execution or load region might use a string stored in different region. If you do not want this behavior, then do one of the following:

- Use the `PROTECTED` load region attribute if you are using scatter-loading.
- Globally disable merging with `--no_merge`.

Default

The default is `--merge`.

Related references

9.53 --info=topic[,topic,...] on page 9-299.
8.5 Load region attributes on page 8-205.

9.80 --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 `--no_muldefweak`.

9.81 --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`*

Where *`filename`* is the name of the output file, and can include a path.

Usage

If `--output=`*`filename`* is not specified, the linker uses the following default filenames:

`__image.axf`
: If the output is an executable image.

`__object.o`
: If the output is a partially-linked object.

If *`filename`* is specified without path information, it is created in the current working directory. If path information is specified, then that directory becomes the default output directory.

Related references

9.14 --callgraph_file=filename on page 9-257.
9.86 --partial on page 9-334.

9.82 --override_visibility

Enables `EXPORT` and `IMPORT` directives in a steering file to override the visibility of a symbol.

Usage

By default:

- Only symbol definitions with `STV_DEFAULT` or `STV_PROTECTED` visibility can be exported.
- Only symbol references with `STV_DEFAULT` visibility can be imported.

When you specify `--override_visibility`, any global symbol definition can be exported and any global symbol reference can be imported.

Related references

9.61 --keep_protected_symbols on page 9-309.

9.123 --undefined_and_export=symbol on page 9-373.

10.1 EXPORT steering file command on page 10-392.

10.3 IMPORT steering file command on page 10-394.

9.83 --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

3.2 Input sections, output sections, regions, and program segments on page 3-36.
3.3 Load view and execution view of an image on page 3-37.

9.84 --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.

Related concepts

3.14 Linker support for creating demand-paged files on page 3-54.

7.33 Creation of regions on page boundaries on page 7-179.

Related references

9.85 --pagesize=pagesize on page 9-333.

9.85 --pagesize=pagesize

Allows you to change the page size used when demand paging.

Syntax

`--pagesize=`*`pagesize`*

Where *`pagesize`* is the page size in bytes. The default value is `0x8000`.

Related concepts

3.14 Linker support for creating demand-paged files on page 3-54.

7.33 Creation of regions on page boundaries on page 7-179.

Related references

9.84 --paged on page 9-332.

9.86 --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 2-31.

9.87 --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.

Default

The default is `--piveneer`.

Related concepts

3.19 Generation of position independent to absolute veneers on page 3-60.

3.16 Overview of veneers on page 3-57.

3.17 Veneer sharing on page 3-58.

3.18 Veneer types on page 3-59.

3.20 Reuse of veneers when scatter-loading on page 3-61.

Related references

9.58 --inlineveneer, --no_inlineveneer on page 9-306.

9.130 --veneershare, --no_veneershare on page 9-380.

9.88 --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`.

Scatter file before preprocessing

The following example shows the scatter file contents before preprocessing.

```
#! armcc -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.

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 concepts

7.35 Preprocessing of a scatter file on page 7-181.

Related references

9.100 --scatter=filename on page 9-349.

9.89 --reduce_paths, --no_reduce_paths

Enables or disables the elimination of redundant path name information in file paths.

Mode

Effective on Windows systems only.

Usage

Windows systems impose a 260 character limit on file paths. Where path names exist whose absolute names expand to longer than 260 characters, you can use the `--reduce_paths` option to reduce absolute path name length by matching up directories with corresponding instances of `..` and eliminating the directory/`..` sequences in pairs.

Note

It is recommended that you avoid using long and deeply nested file paths, in preference to minimizing path lengths using the `--reduce_paths` option.

Default

The default is `--no_reduce_paths`.

Examples

A file to be linked might be at the location:

```
..\..\..\xyzzy\xyzzy\objects\file.c
```

Your current working directory might be at the location:

```
\foo\bar\baz\gazonk\quux\bop
```

The combination of these paths results in the path:

```
\foo\bar\baz\gazonk\quux\bop\..\..\..\xyzzy\xyzzy\objects\file.o
```

By using the option `--reduce_paths` the path becomes:

```
\foo\bar\baz\xyzzy\xyzzy\objects\file.c
```

9.90 --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. For example, if you are building an ARM Linux application.

Default

The default is `--ref_cpp_init`.

Related references

9.22 --cppinit, --no_cppinit on page 9-265.

Related information

C++ initialization, construction and destruction.

9.91 --reloc

Creates a single relocatable load region with contiguous execution regions.

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 this option if an object file contains execute-only sections.

You cannot use this option with `--xo_base`.

Related concepts

7.39 Type 1 image, one load region and contiguous execution regions on page 7-186.
3.8 Type 3 image structure, multiple load regions and non-contiguous execution regions on page 3-46.

Related information

Base Platform ABI for the ARM Architecture.
ELF for the ARM Architecture.

9.92 --remarks

Enables the display of remark messages, including any messages redesignated to remark severity using `--diag_remark`.

Note

The linker does not issue remarks by default.

Related references

9.29 --diag_remark=tag[,tag,...] on page 9-274.
9.40 --errors=filename on page 9-285.

9.93 --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.

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`.

Related concepts

4.3 Elimination of unused sections on page 4-76.
3.23 How the linker performs library searching, selection, and scanning on page 3-66.
4.1 Elimination of common debug sections on page 4-74.
4.2 Elimination of common groups or sections on page 4-75.
4.4 Elimination of unused virtual functions on page 4-78.

Related references

9.60 --keep=section_id on page 9-308.

9.94 --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 references

9.95 --ropi on page 9-344.
9.96 --rosplit on page 9-345.
9.97 --rw_base=address on page 9-346.
9.136 --xo_base=address on page 9-386.
9.140 --zi_base=address on page 9-390.
9.100 --scatter=filename on page 9-349.

9.95 --ropi

Makes the load and execution region containing the RO output section position-independent.

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 `--rwpi`:

- With `--scatter` or `--xo_base`.
- If an object file contains execute-only sections.

Related references

9.94 --ro_base=address on page 9-343.
9.96 --rosplit on page 9-345.
9.97 --rw_base=address on page 9-346.
9.136 --xo_base=address on page 9-386.
9.140 --zi_base=address on page 9-390.
9.100 --scatter=filename on page 9-349.

9.96 --rosplit

Splits the default RO load region into two RO output sections.

The RO load region is split into the RO output sections:

- `RO-CODE`.
- `RO-DATA`.

Restrictions

You cannot use `--rosplit` with `--scatter`.

Related references

9.94 --ro_base=address on page 9-343.
9.95 --ropi on page 9-344.
9.97 --rw_base=address on page 9-346.
9.136 --xo_base=address on page 9-386.
9.140 --zi_base=address on page 9-390.
9.100 --scatter=filename on page 9-349.

9.97 --rw_base=address

Sets the execution addresses of the region containing the RW output section at a specified address.

Syntax

`--rw_base=address`

Where *address* must be word-aligned.

Note

This option does not affect the placement of execute-only sections.

Restrictions

You cannot use `--rw_base` with `--scatter`.

Related references

9.94 --ro_base=address on page 9-343.
9.95 --ropi on page 9-344.
9.96 --rosplit on page 9-345.
9.136 --xo_base=address on page 9-386.
9.140 --zi_base=address on page 9-390.
9.100 --scatter=filename on page 9-349.

9.98 --rwpi

Makes the load and execution region containing the RW and ZI output section position-independent.

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 `--scatter`.
- If an object file contains execute-only sections.

Related references

9.107 --split on page 9-357.
9.100 --scatter=filename on page 9-349.

9.99 --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`.

9.100 --scatter=filename

Creates an image memory map using the scatter-loading description 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`.
- `--tiebreaker`.

The `--scatter` option cannot be used with `--first`, `--last`, `--partial`, `--reloc`, `--ro_base`, `--ropi`, `--rosplit`, `--rw_base`, `--rwpi`, `--split`, `--startup`, `--xo_base`, and `--zi_base`.

Related concepts

7.15 Examples of using placement algorithms for .ANY sections on page 7-154.

7.43 Behavior when .ANY sections overflow because of linker-generated content on page 7-195.

Related references

9.1 --any_contingency on page 9-242.

9.2 --any_placement=algorithm on page 9-243.

9.3 --any_sort_order=order on page 9-245.

9.48 --first=section_id on page 9-293.

9.63 --last=section_id on page 9-311.

9.94 --ro_base=address on page 9-343.

9.95 --ropi on page 9-344.

9.96 --rosplit on page 9-345.

9.97 --rw_base=address on page 9-346.

9.98 --rwpi on page 9-347.

9.107 --split on page 9-357.

9.121 --tiebreaker=option on page 9-371.

9.136 --xo_base=address on page 9-386.

9.140 --zi_base=address on page 9-390.

9.107 --split on page 9-357.

9.86 --partial on page 9-334.

9.91 --reloc on page 9-340.

7 Scatter-loading Features on page 7-130.

9.101 --section_index_display=type

Changes the display of the index column when printing memory map output.

Syntax

`--section_index_display=`*`type`*

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

`cmdline`
Alters the display of the map file to show the order that a section appears on the command-line. The command-line order is defined as `File.Object.Section` where:

- `Section` is the section index, `sh_idx`, of the `Section` in the `Object`.
- `Object` is the order that `Object` appears in the `File`.
- `File` is the order the `File` appears on the command line.

The order the `Object` appears in the `File` is only significant if the file is an `ar` archive.

`internal`
The index value represents the order in which the linker creates the section.

`input`
The index value represents the section index of the section in the original input file. This is useful when you want to find the exact section in an input object.

Usage

Use this option with `--map`.

Default

The default is `--section_index_display=internal`.

Related references

9.74 --map, --no_map on page 9-322.
9.121 --tiebreaker=option on page 9-371.

9.102 --show_cmdline

Outputs the command line used by the .

Usage

Shows the command line after processing by the , and can be useful to check:

- The command line a build system is using.
- How the is interpreting the supplied command line, for example, the ordering of command-line options.

The commands are shown normalized, and the contents of any via files are expanded.

The output is sent to the standard error stream (`stderr`).

Related references

9.52 --help on page 9-298.
9.134 --via=filename on page 9-384.

9.103 --show_full_path

Displays the full path name of an object in any diagnostic messages.

Usage

If the file representing object `obj` has full path name `path/to/obj` then the linker displays `path/to/obj` instead of `obj` in any diagnostic message.

Related references

9.104 --show_parent_lib on page 9-353.
9.105 --show_sec_idx on page 9-354.

9.104 --show_parent_lib

Displays the library name containing an object in any diagnostic messages.

Usage

If an object `obj` comes from library `lib`, then this option displays `lib(obj)` instead of `obj` in any diagnostic messages.

Related references

9.103 --show_full_path on page 9-352.
9.105 --show_sec_idx on page 9-354.

9.105 --show_sec_idx

Displays the section index, `sh_idx`, of section in the originating object.

Examples

If section `sec` has section index 3 then it is displayed as `sec:3` in all diagnostic messages.

Related references

9.103 --show_full_path on page 9-352.
9.104 --show_parent_lib on page 9-353.

9.106 --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 Thumb code before ARM 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 Thumb code before ARM 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 Thumb code before ARM 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.

Default

The default algorithm is `--sort=Lexical`. In large region mode, the default algorithm is `--sort=AvgCallDepth`.

Related concepts

Handling unassigned sections.

3.11 Section placement with the linker on page 3-50.

8.6 Execution region descriptions on page 8-207.

Related references

9.62 --largeregions, --no_largeregions on page 9-310.

9.121 --tiebreaker=option on page 9-371.

8.8 Execution region attributes on page 8-210.

9.107 --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 be 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 be 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 3-34.

Related references

9.100 --scatter=filename on page 9-349.
9.100 --scatter=filename on page 9-349.

9.108 --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 reference to (or definition of) *`symbol`*, then it generates an error.
- If the linker finds a definition of `main()` and a reference to (or definition of) *`symbol`*, and no entry point is specified, then the linker generates a warning.

`--no_startup` does not add a reference.

Default

The default is `--startup=__main`.

Related references

9.39 --entry=location on page 9-284.

9.109 --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 stack checked location from a location that uses the reserved stack checking register R10. This is for *ARM Developer Suite* (ADS) compatibility only.
- A location that uses the reserved stack checking register r10 from a stack checked location. This is for ADS compatibility only.

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

3.27 The strict family of linker options on page 3-71.

Related references

9.29 --diag_remark=tag[,tag,...] on page 9-274.
9.31 --diag_suppress=tag[,tag,...] on page 9-276.
9.32 --diag_warning=tag[,tag,...] on page 9-277.
9.110 --strict_enum_size, --no_strict_enum_size on page 9-360.
9.111 --strict_flags, --no_strict_flags on page 9-361.
9.112 --strict_ph, --no_strict_ph on page 9-362.
9.113 --strict_relocations, --no_strict_relocations on page 9-363.
9.114 --strict_symbols, --no_strict_symbols on page 9-364.
9.115 --strict_visibility, --no_strict_visibility on page 9-365.
9.116 --strict_wchar_size, --no_strict_wchar_size on page 9-366.
9.28 --diag_error=tag[,tag,...] on page 9-273.
9.40 --errors=filename on page 9-285.

9.110 --strict_enum_size, --no_strict_enum_size

Checks whether or not the enum size is consistent across all inputs.

Usage

Use `--strict_enum_size` to force the linker to display an error message if the enum size is not consistent across all inputs. This is the default.

Use `--no_strict_enum_size` for compatibility with objects built using RVCT v3.1 and earlier.

Related concepts

3.27 The strict family of linker options on page 3-71.

Related references

9.109 --strict on page 9-359.
9.111 --strict_flags, --no_strict_flags on page 9-361.
9.112 --strict_ph, --no_strict_ph on page 9-362.
9.113 --strict_relocations, --no_strict_relocations on page 9-363.
9.114 --strict_symbols, --no_strict_symbols on page 9-364.
9.115 --strict_visibility, --no_strict_visibility on page 9-365.
9.116 --strict_wchar_size, --no_strict_wchar_size on page 9-366.

Related information

--enum_is_init compiler option.

9.111 --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

3.27 The strict family of linker options on page 3-71.

Related references

9.109 --strict on page 9-359.
9.110 --strict_enum_size, --no_strict_enum_size on page 9-360.
9.112 --strict_ph, --no_strict_ph on page 9-362.
9.113 --strict_relocations, --no_strict_relocations on page 9-363.
9.114 --strict_symbols, --no_strict_symbols on page 9-364.
9.115 --strict_visibility, --no_strict_visibility on page 9-365.
9.116 --strict_wchar_size, --no_strict_wchar_size on page 9-366.

Related information

ARM ELF Specification (SWS ESPC 0003 B-02).

9.112 --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.

Related concepts

3.27 The strict family of linker options on page 3-71.

Related references

9.109 --strict on page 9-359.
9.110 --strict_enum_size, --no_strict_enum_size on page 9-360.
9.111 --strict_flags, --no_strict_flags on page 9-361.
9.113 --strict_relocations, --no_strict_relocations on page 9-363.
9.114 --strict_symbols, --no_strict_symbols on page 9-364.
9.115 --strict_visibility, --no_strict_visibility on page 9-365.
9.116 --strict_wchar_size, --no_strict_wchar_size on page 9-366.

9.113 --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

3.27 The strict family of linker options on page 3-71.

Related references

9.109 --strict on page 9-359.
9.110 --strict_enum_size, --no_strict_enum_size on page 9-360.
9.111 --strict_flags, --no_strict_flags on page 9-361.
9.112 --strict_ph, --no_strict_ph on page 9-362.
9.114 --strict_symbols, --no_strict_symbols on page 9-364.
9.115 --strict_visibility, --no_strict_visibility on page 9-365.
9.116 --strict_wchar_size, --no_strict_wchar_size on page 9-366.

9.114 --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`.

Examples

In the following assembler code the symbol `sym` has type `STT_FUNC` and is ARM:

```
        area code, readonly
        DCD sym + 4
        ARM
sym  PROC
        NOP
        THUMB
        NOP
        ENDP
        END
```

The difference in behavior is the meaning of `DCD sym + 4`:

- In pre-ABI linkers the state of the symbol is the state of the only of the mapping symbol at that location. In this example, the state is Thumb.
- In ABI linkers the type of the symbol is the state of the location of symbol plus the offset.

Related concepts

3.27 The strict family of linker options on page 3-71.
6.1 About mapping symbols on page 6-104.

Related references

9.109 --strict on page 9-359.
9.110 --strict_enum_size, --no_strict_enum_size on page 9-360.
9.111 --strict_flags, --no_strict_flags on page 9-361.
9.112 --strict_ph, --no_strict_ph on page 9-362.
9.113 --strict_relocations, --no_strict_relocations on page 9-363.
9.115 --strict_visibility, --no_strict_visibility on page 9-365.
9.116 --strict_wchar_size, --no_strict_wchar_size on page 9-366.

9.115 --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

3.27 The strict family of linker options on page 3-71.

Related references

9.109 --strict on page 9-359.
9.110 --strict_enum_size, --no_strict_enum_size on page 9-360.
9.111 --strict_flags, --no_strict_flags on page 9-361.
9.112 --strict_ph, --no_strict_ph on page 9-362.
9.113 --strict_relocations, --no_strict_relocations on page 9-363.
9.114 --strict_symbols, --no_strict_symbols on page 9-364.
9.116 --strict_wchar_size, --no_strict_wchar_size on page 9-366.

9.116 --strict_wchar_size, --no_strict_wchar_size

Checks whether or not the wide character size is consistent across all inputs.

Usage

The option `--strict_wchar_size` causes the linker to display an error message if the wide character size is not consistent across all inputs. This is the default.

Use `--no_strict_wchar_size` for compatibility with objects built using RVCT v3.1 and earlier.

Related concepts

3.27 The strict family of linker options on page 3-71.

Related references

9.109 --strict on page 9-359.
9.110 --strict_enum_size, --no_strict_enum_size on page 9-360.
9.111 --strict_flags, --no_strict_flags on page 9-361.
9.112 --strict_ph, --no_strict_ph on page 9-362.
9.113 --strict_relocations, --no_strict_relocations on page 9-363.
9.114 --strict_symbols, --no_strict_symbols on page 9-364.
9.115 --strict_visibility, --no_strict_visibility on page 9-365.

Related information

--wchar16 compiler option.
--wchar32 compiler option.

9.117 --symbols, --no_symbols

Enables or disables the listing of each local and global symbol used in the link step, and its value.

Note

This does not include mapping symbols output to `stdout`. Use `--list_mapping_symbols` to include mapping symbols in the output.

Default

The default is `--no_symbols`.

Related references

9.70 --list_mapping_symbols, --no_list_mapping_symbols on page 9-318.

9.118 --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

6.14 Access symbols in another image on page 6-118.

9.119 --tailreorder, --no_tailreorder

Moves tail calling sections immediately before their target, if possible, to optimize the branch instruction at the end of a section.

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

4.15 Linker reordering of tail calling sections on page 4-92.
4.14 About branches that optimize to a NOP on page 4-91.

Related references

9.12 --branchnop, --no_branchnop on page 9-254.

9.120 --thumb2_library, --no_thumb2_library

Enables you to link against the combined ARM and Thumb library.

Usage

`--thumb2_library` only applies when the processor supports ARM and Thumb-2 technology, such as the Cortex-A and Cortex-R series processors.

Use the `--no_thumb2_library` option to revert to the ARMv5T and later libraries.

Note

The linker ignores `--thumb2_library` if the target does not support Thumb-2 technology.

Default

The default is `--thumb2_library`.

Related information

C and C++ library naming conventions.

9.121 --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 tiebreak. The command-line index is more stable across builds.

Use this option with the `--scatter` option.

Default

The default option is `creation`.

Related concepts

7.17 Examples of using sorting algorithms for .ANY sections on page 7-158.

Related references

9.100 --scatter=filename on page 9-349.
9.101 --section_index_display=type on page 9-350.
9.106 --sort=algorithm on page 9-355.
9.74 --map, --no_map on page 9-322.
9.3 --any_sort_order=order on page 9-245.

9.122 --undefined=symbol

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

Syntax

`--undefined=`*`symbol`*

Usage

Causes the linker to:

1. Create a symbol reference to the specified symbol name.
2. Issue an implicit `--keep(`*`symbol`*`)` to prevent any sections brought in to define that symbol from being removed.

Related references

9.123 --undefined_and_export=symbol on page 9-373.

9.60 --keep=section_id on page 9-308.

9.123 --undefined_and_export=symbol

Prevents the removal of a specified symbol if it is undefined, and pushes the symbol into the dynamic symbol table.

Syntax

`--undefined_and_export=`*`symbol`*

Usage

Causes the linker to:

1. Create a symbol reference to the specified symbol name.
2. Issue an implicit `--keep(`*`symbol`*`)` to prevent any sections brought in to define that symbol from being removed.
3. Add an implicit `EXPORT` *`symbol`* to push the specified symbol into the dynamic symbol table.

Considerations

Be aware of the following when using this option:

- It does not change the visibility of a symbol unless you specify the `--override_visibility` option.
- A warning is issued if the visibility of the specified symbol is not high enough.
- A warning is issued if the visibility of the specified symbol is overridden because you also specified the `--override_visibility` option.
- Hidden symbols are not exported unless you specify the `--override_visibility` option.

Related references

9.82 --override_visibility on page 9-330.
9.122 --undefined=symbol on page 9-372.
9.60 --keep=section_id on page 9-308.
10.1 EXPORT steering file command on page 10-392.

9.124 --unresolved=symbol

Takes each reference to an undefined symbol and matches it to the global definition of the specified symbol.

Syntax

`--unresolved=`*`symbol`*

`symbol` must be both defined and global, otherwise it appears in the list of undefined symbols and the link step fails.

Usage

This option is particularly useful during top-down development, because it enables you to test a partially-implemented system by matching each reference to a missing function to a dummy function.

Related references

9.122 --undefined=symbol on page 9-372.
9.123 --undefined_and_export=symbol on page 9-373.

9.125 --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.

———————————

9.126 --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

3.23 How the linker performs library searching, selection, and scanning on page 3-66.

Related references

9.66 --libpath=pathlist on page 9-314.

9.127 --veneerinject,--no_veneerinject

Enables or disables the placement of veneers outside of the sorting order for the Execution Region.

Usage

Use `--veneerinject` to allow the linker to place veneers outside of the sorting order for the Execution Region. This option is a subset of the `--largeregions` command. Use `--veneerinject` if you want to allow the veneer placement behavior described, but do not want to implicitly set the `--api` and `--sort=AvgCallDepth`.

Use `--no_veneerinject` to allow the linker use the sorting order for the Execution Region.

Use `--veneer_inject_type` to control the strategy the linker uses to place injected veneers.

The following command-line options allow stable veneer placement with large Execution Regions:

```
--veneerinject --veneer_inject_type=pool --sort=lexical
```

Default

The default is `--no_veneerinject`. The linker automatically switches to large region mode if it is required to successfully link the image. If large region mode is turned off with `--no_largeregions` then only `--veneerinject` is turned on if it is required to successfully link the image.

Note

`--veneerinject` is the default for large region mode.

Related references

9.62 --largeregions, --no_largeregions on page 9-310.
9.128 --veneer_inject_type=type on page 9-378.
9.4 --api, --no_api on page 9-246.
9.106 --sort=algorithm on page 9-355.

9.128 --veneer_inject_type=type

Controls the veneer layout when `--largeregions` mode is on.

Syntax

`--veneer_inject_type=`*`type`*

Where *`type`* is one of:

`individual`
The linker places veneers to ensure they can be reached by the largest amount of sections that use the veneer. Veneer reuse between execution regions is permitted. This type minimizes the number of veneers that are required but disrupts the structure of the image the most.

`pool`
The linker:

1. Collects veneers from a contiguous range of the execution region
2. Places all the veneers generated from that range into a pool.
3. Places that pool at the end of the range.

A large execution region might have more than one range and therefore more than one pool. Although this type has much less impact on the structure of image, it has fewer opportunities for reuse. This is because a range of code cannot reuse a veneer in another pool. The linker calculates the range based on the presence of branch instructions that the linker predicts might require veneers. A branch is predicted to require a veneer when either:

- A state change is required.
- The distance from source to target plus a contingency greater than the branch range.

You can set the size of the contingency with the `--veneer_pool_size=`*`size`* option. By default the contingency size is set to 102400 bytes. The `--info=veneerpools` option provides information on how the linker has placed veneer pools.

Restrictions

You must use `--largeregions` with this option.

Related references

9.53 --info=topic[,topic,...] on page 9-299.
9.127 --veneerinject,--no_veneerinject on page 9-377.
9.129 --veneer_pool_size=size on page 9-379.
9.62 --largeregions, --no_largeregions on page 9-310.

9.129 --veneer_pool_size=size

Sets the contingency size for the veneer pool in an execution region.

Syntax

`--veneer_pool_size=`*`pool`*

where *`pool`* is the size in bytes.

Default

The default size is 102400 bytes.

Related references

9.128 --veneer_inject_type=type on page 9-378.

9.130 --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

3.17 Veneer sharing on page 3-58.

3.16 Overview of veneers on page 3-57.

3.18 Veneer types on page 3-59.

3.19 Generation of position independent to absolute veneers on page 3-60.

Related references

9.58 --inlineveneer, --no_inlineveneer on page 9-306.

9.87 --piveneer, --no_piveneer on page 9-335.

9.25 --crosser_veneershare, --no_crosser_veneershare on page 9-270.

9.131 --verbose

Prints detailed information about the link operation, including the objects that are included and the libraries from which they are taken.

Usage

This output is particular useful for tracing undefined symbols reference or multiply defined symbols. Because this output is typically quite long, you might want to use this command with the `--list=*filename*` command to redirect the information to *`filename`*.

Use `--verbose` to output diagnostics to `stdout`.

Related references

9.69 --list=filename on page 9-317.
9.80 --muldefweak, --no_muldefweak on page 9-328.
9.124 --unresolved=symbol on page 9-374.

9.132 --version_number

Displays the version of `armlink` you are using.

Usage

The linker displays the version number in the format `nnnbbbb`, where:

- `nnn` is the version number.
- `bbbb` is the build number.

Examples

Version 5.01 build 0019 is displayed as `5010019`.

Related references

9.52 --help on page 9-298.
9.135 --vsn on page 9-385.

9.133 --vfemode=mode

Specifies how *Virtual Function Elimination* (VFE), and *RunTime Type Information* (RTTI) objects, are eliminated. VFE is a technique that enables the linker to identify more unused sections.

Syntax

`--vfemode=`*`mode`*

where *`mode`* is one of the following:

`on`

Use the command-line option `--vfemode=on` to make the linker VFE aware.

In this mode the linker chooses `force` or `off` mode based on the content of object files:

- Where every object file contains VFE information or does not refer to a symbol with a mangled C++ name, the linker assumes `force` mode and continues with the elimination.
- If any object file is missing VFE information and refers to a symbol with a mangled C++ name, for example, where code has been compiled with a previous release of the ARM tools, the linker assumes `off` mode, and VFE is disabled silently. Choosing `off` mode to disable VFE in this situation ensures that the linker does not remove a virtual function that is used by an object with no VFE information.

`off`

Use the command-line option `--vfemode=off` to make `armlink` ignore any extra information supplied by the compiler. In this mode, the final image is the same as that produced by compiling and linking without VFE awareness.

`force`

Use the command-line option `--vfemode=force` to make the linker VFE aware and force the VFE algorithm to be applied. If some of the object files do not contain VFE information, for example, where they have been compiled with a previous release of the ARM tools, the linker continues with the elimination but displays a warning to alert you to possible errors.

`force_no_rtti`

Use the command-line option `--vfemode=force_no_rtti` to make the linker VFE aware and force the removal of all RTTI objects. In this mode all virtual functions are retained.

Default

The default is `--vfemode=on`.

Related concepts

4.4 Elimination of unused virtual functions on page 4-78.
4.1 Elimination of common debug sections on page 4-74.
4.2 Elimination of common groups or sections on page 4-75.
4.3 Elimination of unused sections on page 4-76.

9.134 --via=filename

Reads an additional list of input filenames and 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 command line. The `--via` options can also be included within a via file.

Related references

11.2 Via file syntax rules on page 11-402.

9.135 --vsn

Displays the version information and the license details.

Examples

Example output:

```
> armlink --vsn
Product: ARM Compiler N.nn
Component: ARM Compiler N.nn
Tool: armlink [build_number]
license_type
Software supplied by: ARM Limited
```

Related references

9.52 --help on page 9-298.
9.132 --version_number on page 9-382.

9.136 --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.

You cannot use `--xo_base` with `--reloc`, `--scatter`, `--ropi`, or `--rwpi`.

Related concepts

2.2 Bare-metal linking model on page 2-29.

Related references

9.94 --ro_base=address on page 9-343.
9.95 --ropi on page 9-344.
9.96 --rosplit on page 9-345.
9.97 --rw_base=address on page 9-346.
9.140 --zi_base=address on page 9-390.
9.100 --scatter=filename on page 9-349.

9.137 --xref, --no_xref

Lists to stdout all cross-references between input sections.

Default

The default is `--no_xref`.

Related references

9.69 --list=filename on page 9-317.
9.138 --xrefdbg, --no_xrefdbg on page 9-388.
9.139 --xref{from|to}=object(section) on page 9-389.

9.138 --xrefdbg, --no_xrefdbg

Lists to stdout all cross-references between input debug sections.

Default

The default is `--no_xrefdbg`.

Related references

9.69 --list=filename on page 9-317.
9.137 --xref, --no_xref on page 9-387.
9.139 --xref{from|to}=object(section) on page 9-389.

9.139 --xref{from|to}=object(section)

Lists to stdout cross-references from and to input sections.

Syntax

`--xref{from|to}=object(section)`

Usage

This option lists to stdout cross-references:

- From input *section* in *object* to other input sections.
- To input *section* in *object* from other input sections.

This is a useful subset of the listing produced by the `--xref` linker option if you are interested in references from or to a specific input section. You can have multiple occurrences of this option to list references from or to more than one input section.

Related references

9.69 --list=filename on page 9-317.
9.137 --xref, --no_xref on page 9-387.
9.138 --xrefdbg, --no_xrefdbg on page 9-388.

9.140 --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:

- `--reloc`.
- `--rwpi`.
- `--split`.

You cannot use `--zi_base` with `--scatter`.

Related references

9.94 --ro_base=address on page 9-343.
9.95 --ropi on page 9-344.
9.96 --rosplit on page 9-345.
9.97 --rw_base=address on page 9-346.
9.136 --xo_base=address on page 9-386.
9.100 --scatter=filename on page 9-349.

Chapter 10
Linker Steering File Command Reference

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

It contains the following sections:

10.1 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 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

`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

6.19 What is a steering file? on page 6-124.

Related references

9.82 --override_visibility on page 9-330.
10.3 IMPORT steering file command on page 10-394.

10.2 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

6.19 What is a steering file? on page 6-124.

Related references

10.7 SHOW steering file command on page 10-399.
9.34 --edit=file_list on page 9-279.
9.86 --partial on page 9-334.

10.3 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

6.19 What is a steering file? on page 6-124.

Related references

9.82 --override_visibility on page 9-330.
10.1 EXPORT steering file command on page 10-392.

10.4 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. 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, 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 bar
RENAME foo2 bar
```

Renames only take effect at the end of the link step. Therefore, renaming a symbol does not remove its original name. This means that you cannot do the following:

```
RENAME func1 func2
RENAME func2 func3
```

The linker gives an error that func1 cannot be renamed to func2 as a symbol already exists with that name.

Only one wildcard character (either `*` or `?`) is permitted in `RENAME`.

Examples

Given an image containing the symbols `func1`, `func2`, and `func3`, you might have a steering file containing the following commands:

```
;invalid, func2 already exists EXPORT func1 AS func2
; valid RENAME func3 AS b2
;invalid, func3 still exists because the link step is not yet complete EXPORT func1 AS func3
```

Related concepts

6.19 What is a steering file? on page 6-124.

10.5 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

6.19 What is a steering file? on page 6-124.

10.6 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.

Examples

You might have two files `file1.c` and `file2.c`, as shown in the following example:

Using the RESOLVE command

```
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

6.19 What is a steering file? on page 6-124.

Related references

9.34 --edit=file_list on page 9-279.
9.124 --unresolved=symbol on page 9-374.

10.7 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

6.19 What is a steering file? on page 6-124.

Related references

10.2 HIDE steering file command on page 10-393.

Chapter 11
Via File Syntax

Describes the syntax of via files accepted by the `armlink`.

It contains the following sections:

11.1 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 references

11.2 Via file syntax rules on page 11-402.
9.134 --via=filename on page 9-384.

11.2 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:

  `--dll --base_platform` (two words)

  `--dll--base_platform` (one word)
- The end of a line is treated as whitespace, for example:

  ```
  --dll
  --base_platform
  ```

  This is equivalent to:

  `--dll --base_platform`
- 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

11.1 Overview of via files on page 11-401.

Related references

9.134 --via=filename on page 9-384.