RealView® Compilation Tools

Version 4.0

Linker User Guide

ARM®

ARM DUI 0206J (ID101213)

RealView Compilation Tools
Linker User Guide

Release Information

The following changes have been made to this book.

Change History

Date	Issue	Confidentiality	Change
August 2002	A	Non-Confidential	Release 1.2
January 2003	B	Non-Confidential	Release 2.0
September 2003	C	Non-Confidential	Release 2.0.1 for ARM® RealView® Developer Suite
January 2004	D	Non-Confidential	Release 2.1 for RealView Developer Suite
December 2004	E	Non-Confidential	Release 2.2 for RealView Developer Suite
May 2005	F	Non-Confidential	Release 2.2 SP1 for RealView Development Suite
March 2006	G	Non-Confidential	Release 3.0 for RealView Development Suite
March 2007	H	Non-Confidential	Release 3.1 for RealView Development Suite
September 2008	I	Non-Confidential	Release 4.0 for RealView Development Suite
23 January 2009	I	Non-Confidential	Update 1 for RealView Development Suite v4.0
10 December 2010	J	Non-Confidential	Update 2 for RealView Development Suite v4.0

Proprietary Notice

Product Status

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

Web Address

`http://www.arm.com`

Contents
RealView Compilation Tools Linker User Guide

Preface

This preface introduces the *RealView Compilation Tools Linker User Guide*. It contains the following sections:

- *About this book* on page viii
- *Feedback* on page xi.

About this book

This book provides user information for the ARM® linker, `armlink` provided with ARM RealView® Compilation Tools. It also gives an overview of the command-line options. For detailed information on command-line options, steering file commands, scatter-loading description files, the *Base Platform ABI* (BPABI) and *System V release 4* (SysV) shared libraries and executables, see the *Linker Reference Guide*.

Intended audience

This book is written for all developers who are producing applications using RealView Compilation Tools. It assumes that you are an experienced software developer and that you are familiar with the ARM development tools.

Using this book

This book is organized into the following chapters:

Chapter 1 ***Introduction***

Read this chapter for an introduction to the linker and its associated input and output files.

Chapter 2 ***Getting Started with the ARM Linker***

Read this chapter for an overview of the linking models, file naming conventions, and command-line options supported by the linker.

Chapter 3 ***Using the Basic Linker Functionality***

Read this chapter for an overview of image structures, memory maps, section placement, linker optimization, and use of library files.

Chapter 4 ***Accessing Image Symbols***

Read this chapter for an overview of accessing linker defined symbols, how to use steering files to manage symbol names in output files, and symbol versioning.

Chapter 5 ***Using Scatter-loading Description Files***

Read this chapter for an overview of the scatter-loading description files and how you can use them to specify the memory map of a simple or complex image.

This book assumes that the ARM software is installed in the default location. For example, on Windows this might be `volume:\Program Files\ARM`. This is assumed to be the location of *install_directory* when referring to path names, for example `install_directory\Documentation\...`. You might have to change this if you have installed your ARM software in a different location.

Typographical conventions

The following typographical conventions are used in this book:

`monospace`	Denotes text that can be entered at the keyboard, such as commands, file and program names, and source code.
`monospace`	Denotes a permitted abbreviation for a command or option. The underlined text can be entered instead of the full command or option name.
`monospace italic`	Denotes arguments to commands and functions where the argument is to be replaced by a specific value.
`monospace bold`	Denotes language keywords when used outside example code.
italic	Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.
bold	Highlights interface elements, such as menu names. Also used for emphasis in descriptive lists, where appropriate, and for ARM processor signal names.

Further reading

This section lists publications from both ARM Limited and third parties that provide additional information on developing code for the ARM family of processors.

ARM Limited periodically provides updates and corrections to its documentation. See `http://infocenter.arm.com/help/index.jsp` for current errata sheets and addenda, and the *ARM Frequently Asked Questions* (FAQs).

ARM publications

This book contains information on how to use the ARM linker supplied with RealView Compilation Tools. Other publications included in the suite are:

- *RealView Compilation Tools Essentials Guide* (ARM DUI 0202)
- *RealView Compilation Tools Compiler User Guide* (ARM DUI 0205)
- *RealView Compilation Tools Compiler Reference Guide* (ARM DUI 0348)
- *RealView Compilation Tools Libraries and Floating Point Support Guide* (ARM DUI 0349)
- *RealView Compilation Tools Linker Reference Guide* (ARM DUI 0381)
- *RealView Compilation Tools Utilities Guide* (ARM DUI 0382)
- *RealView Compilation Tools Assembler Guide* (ARM DUI 0204)
- *RealView Compilation Tools Developer Guide* (ARM DUI 0203).

For full information about the base standard, software interfaces, and standards supported by ARM, see the ARM website, `http://infocenter.arm.com/help/index.jsp`.

In addition, see the following documentation for specific information relating to ARM products:

- *ARM Architecture Reference Manual, ARMv7-A™ and ARMv7-R™ edition* (ARM DDI 0406)
- *ARM7-M™ Architecture Reference Manual* (ARM DDI 0403)
- *ARM6-M™ Architecture Reference Manual* (ARM DDI 0419)
- ARM datasheet or technical reference manual for your hardware device.

Feedback

ARM Limited welcomes feedback on both RealView Compilation Tools and the documentation.

Feedback on RealView Compilation Tools

If you have any problems with RealView Compilation Tools, contact your supplier. To help them provide a rapid and useful response, give:

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

Feedback on this book

If you notice any errors or omissions in this book, send an email to `errata@arm.com` giving:

- the document title
- the document number
- the page number(s) to which your comments apply
- a concise explanation of the problem.

General suggestions for additions and improvements are also welcome.

Chapter 1
Introduction

This chapter introduces the ARM® linker, `armlink`, provided with RealView® Compilation Tools. It contains the following sections:

- *About the ARM linker* on page 1-2
- *Compatibility with legacy objects and libraries* on page 1-5.

1.1 About the ARM linker

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

- an ARM ELF image
- a partially linked ELF object that can be used as input in a subsequent link step
- ELF files that can be demand-paged efficiently
- a shared object, compatible with the *Base Platform Application Binary Interface* (BPABI) or *System V release 4* (SysV) specification, or a BPABI or SysV executable file.

The linker can:

- link ARM code and Thumb® and Thumb-2 code
- generate interworking veneers to switch processor state when required
- generate inline veneers or long branch veneers, where required, to extend the range of branch instructions
- perform *link-time code generation* (LTCG)
- pass a profiling data file to the compiler to perform Profiler-guided optimizations
- automatically select the appropriate standard C or C++ library variants to link with, based on the build attributes of the objects it is linking
- enable you to specify the locations of code and data within the system memory map, using either a command-line option or a scatter-loading description file
- perform Read/Write data compression to minimize ROM size
- perform unused section elimination to reduce the size of your output image
- control the generation of debug information in the output file
- generate a static callgraph and list the stack usage
- control the contents of the symbol table in output images
- show the sizes of code and data in the output
- use linker feedback to remove individual unused functions.

See also:

- *Input to the ARM linker* on page 1-3

- *Output from the ARM linker*
- Chapter 2 *Getting Started with the ARM Linker*.

1.1.1 Input to the ARM linker

Input to `armlink` consists of one or more object files in 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-loading description file
- a steering file.

1.1.2 Output from the ARM linker

Output from a successful invocation of `armlink` is one of the following:

- an ELF executable image
- an ELF shared object
- a partially-linked ELF object
- a relocatable ELF object.

You can use `fromelf` to convert an ELF executable image to other file formats, or to display, process, and protect the content of and ELF executable image. See Chapter 2 *Using fromelf* in the *Utilities Guide* for more information.

Constructing an executable image

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

- resolves symbolic references between the input object files
- extracts object modules from libraries to satisfy otherwise unsatisfied symbolic references
- sorts input sections according to their attributes and names, and merges similarly attributed and named sections into contiguous chunks
- removes unused sections
- eliminates duplicate common groups and common code, data, and debug sections
- organizes object fragments into memory regions according to the grouping and placement information provided

- assigns addresses to relocatable values
- generates an executable image.

See *The image structure* on page 3-2 for more information.

Constructing a partially-linked object

When you use the linker to construct a partially-linked object, it:

- eliminates duplicate copies of debug sections
- merges the symbol tables into one
- leaves unresolved references unresolved
- merges Comdat groups
- generates an object that can be used as an input to a subsequent link step.

Note

If you use partial linking, you cannot refer to the component objects by name in a scatter-loading description file.

1.2 Compatibility with legacy objects and libraries

If you are upgrading to RealView Compilation Tools from a previous release, ensure that you read *RealView Compilation Tools Essentials Guide* for the latest information.

The Application Binary Interface (ABI) used in RealView Compilation Tools v4.0 is different to that in earlier tools pre-v2.0. Therefore, legacy objects and libraries are not compatible and must be rebuilt.

See *ABI for the ARM Architecture compliance* on page 1-4 in the *Libraries Guide*, and the section on compatibility with legacy objects and libraries in *RealView Compilation Tools Essentials Guide* for more information.

Chapter 2
Getting Started with the ARM Linker

This chapter outlines the command-line options accepted by the ARM® linker, `armlink`. It describes the ordering of command-line options, how to invoke the linker, how the environment variables must be configured, and ARM RealView® Compilation Tools file naming conventions.

The linker combines the contents of one or more object files with selected parts of one or more object libraries to produce an image or object file. It contains the following sections:

2.1 Linking models

This section describes the linking models supported by the ARM linker and outlines the specific behavior and restrictions that apply. Options that are common to all linking models are not described.

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

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

Partial linking

This model produces a platform-independent object suitable for input to the linker in a subsequent link step. It can be used as an intermediate step in the development process and performs limited processing of input objects to produce a single output object.

BPABI This model supports the DLL-like *Base Platform Application Binary Interface* (BPABI). It is intended to produce applications and DLLs that will run on a platform OS that varies in complexity. The memory model is restricted according to the BPABI specification.

Base Platform

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

SysV This model supports SysV models specified in the ELF used by ARM Linux. The memory model is restricted according to the ELF specification.

Within each model, related options can be combined to tighten control over the output. The following sections describe these combinations in more detail.

See Chapter 4 *BPABI and SysV Shared Libraries and Executables* in the *Linker Reference Guide* for more information.

See also:

- *Bare-metal linking model* on page 2-3
- *Partial linking model* on page 2-4
- *Concepts common to both BPABI and SysV linking models* on page 2-5
- *Base Platform Application Binary Interface (BPABI) linking model* on page 2-6
- *Base Platform linking model* on page 2-7

- *SysV linking model* on page 2-8.

2.1.1 Bare-metal linking model

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

By default, the linker attempts to resolve all the relocations statically. However, it is also possible to create a position-independent or relocatable image that can be executed from different addresses and have its relocations resolved at load or run-time. This can be achieved using a dynamic model.

This type of model has three parts:

- identify the regions that can be relocated or are position-independent using a scatter-loading description file or command-line options.
- identify the symbols that can be imported and exported using a steering file
- identify the shared libraries that are required by the ELF file using a steering file.

You can use the following options with this model:

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

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

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

See also

- *Specifying an image memory map* on page 3-7
- the following in the *Linker Reference Guide*:
  - *--edit=file_list* on page 2-31
  - *--reloc* on page 2-72
  - *--ro_base=address* on page 2-74

- *--ropi* on page 2-75
- *--rosplit* on page 2-75
- *--rw_base=address* on page 2-76
- *--rwpi* on page 2-77
- *--scatter=file* on page 2-78
- *--split* on page 2-83
- *Steering file commands in alphabetical order* on page 2-102.

2.1.2 Partial linking model

Partial linking:

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

A single output file is produced that can be used as input to a subsequent link step. If the linker finds multiple entry points in the input files it generates an error because the output file can have only one entry point.

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

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

Note

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

Command-line options

the following in the *Linker Reference Guide*:

- *--edit=file_list* on page 2-31
- *--exceptions_tables=action* on page 2-35
- *--partial* on page 2-64
- *Steering file commands in alphabetical order* on page 2-102.

2.1.3 Concepts common to both BPABI and SysV linking models

For both *Base Platform Application Binary Interface* (BPABI) and *System V* (SysV) linking models, images and shared objects usually run on an existing operating platform.

There are many similarities between the BPABI and the SysV models. For example, both produce a program header that maps the exception tables. The main differences are in the memory model, and in the *Procedure Linkage Table* (PLT) and *Global Offset Table* (GOT) structure, referred to collectively as PLTGOT. There are many options that are common to both models.

Restrictions

Both the BPABI and SysV models have the following restrictions:

- unused section elimination is turned off for shared libraries and DLLs
- virtual function elimination is turned off
- read write data compression is not permitted
- scatter-loading is not permitted
- `__AT` sections are not permitted.

Note

Scatter-loading is supported in the Base Platform linking model.

See also

- *Base Platform Application Binary Interface (BPABI) linking model* on page 2-6
- *Base Platform linking model* on page 2-7
- *SysV linking model* on page 2-8
- the following in the *Linker Reference Guide*:
  - *--base_platform* on page 2-14
  - *--bpabi* on page 2-17
  - *--dynamic_debug* on page 2-30
  - *--force_so_throw, --no_force_so_throw* on page 2-41
  - *--runpath=pathlist* on page 2-76
  - *--soname=name* on page 2-81
  - *--symver_script=file* on page 2-90
  - *--symver_soname* on page 2-90
  - *--sysv* on page 2-90.

2.1.4 Base Platform Application Binary Interface (BPABI) linking model

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

Be aware of the following:

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

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

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

See also

- *Concepts common to both BPABI and SysV linking models* on page 2-5
- *Base Platform linking model* on page 2-7
- *Symbol versioning* on page 4-19
- the following in the *Linker Reference Guide*:
  - *--bpabi* on page 2-17
  - *--dll* on page 2-30
  - *--force_so_throw, --no_force_so_throw* on page 2-41
  - *--pltgot=type* on page 2-65
  - *--ro_base=address* on page 2-74
  - *--rosplit* on page 2-75
  - *--rw_base=address* on page 2-76
  - *--rwpi* on page 2-77.

2.1.5 Base Platform linking model

Base Platform enables you to create dynamically linkable images that do not have the memory map enforced by the *System V* (SysV) or *Base Platform Application Binary Interface* (BPABI) linking models. It enables you to:

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

Note

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

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

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

- Scatter-loading is available with `--scatter`, in addition to the following options:
  - `--dll`
  - `--force_so_throw`, `--no_force_so_throw`
  - `--pltgot=`*`type`* is restricted to types `none` or `direct`
  - `--ro_base=`*`address`*
  - `--rosplit`
  - `--rw_base=`*`address`*
  - `--rwpi`.
- The default value of the `--pltgot` option is different to that for `--bpabi`:
  - for `--base_platform`, the default is `--pltgot=none`
  - for `--bpabi` the default is `--pltgot=direct`.
- If you do not use a scatter file, the linker can ensure that the *Procedure Linkage Table* (PLT) section is placed correctly, and contains entries for calls only to imported symbols. If you specify a scatter file, the linker might not be able to find a suitable location to place the PLT.

  Each load region containing code might require a PLT section to indirect calls from the load region to functions where the address is not known at static link time. The PLT section for a load region `LR` must be placed in `LR` and be accessible at all times to code within `LR`.

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

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

Be aware of the following:

- The model assumes that shared objects cannot throw a C++ exception.
- Symbol versioning must be used to ensure that all the required symbols are available at load time.
- There are restrictions on the type of scatter files you can use.

See also

- *Base Platform Application Binary Interface (BPABI) linking model* on page 2-6
- *Concepts common to both BPABI and SysV linking models* on page 2-5
- *Specifying an image memory map* on page 3-7
- *Symbol versioning* on page 4-19
- *Restrictions on the use of scatter files with the Base Platform model* on page 5-2
- *Example scatter file for the Base Platform linking model* on page 5-5
- *Linker Reference*:
  - *--base_platform* on page 2-14
  - *--dll* on page 2-30
  - *--force_so_throw, --no_force_so_throw* on page 2-41
  - *--pltgot=type* on page 2-65
  - *--pltgot_opts=mode* on page 2-66
  - *--ro_base=address* on page 2-74
  - *--rosplit* on page 2-75
  - *--rw_base=address* on page 2-76
  - *--rwpi* on page 2-77
  - *--scatter=file* on page 2-78.

2.1.6 SysV linking model

The *System V* (SysV) model produces SysV shared objects and executables. It can also be used to produce ARM Linux compatible shared objects and executables.

Be aware of the following:

- you cannot use scatter-loading

- the model assumes that shared objects can throw an exception
- thread local storage is supported.

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

- `--force_so_throw`, `--no_force_so_throw`
- `--fpic`
- `--linux_abitag=`*`version_id`*
- `--shared`.

See also

- *Concepts common to both BPABI and SysV linking models* on page 2-5
- the following in the *Linker Reference Guide*:
  - *--force_so_throw, --no_force_so_throw* on page 2-41
  - *--fpic* on page 2-42
  - *--linux_abitag=version_id* on page 2-56
  - *--shared* on page 2-80
  - *--sysv* on page 2-90.

2.2 Using command-line options

The following rules apply, depending on the type of option:

Single-letter options

All single-letter options, or single-letter options with arguments, are preceded by a single dash `-`. You can use a space between the option and the argument, or the argument can immediately follow the option.

Keyword options

All keyword options, or keyword options with arguments, are preceded by a double dash `--`. An = or space character is required between the option and the argument. For example:

`--scatter=`*`file`*

`--scatter` *`file`*.

Options that contain non-leading - or _ can use either of these characters, for example, `--force_so_throw` is the same as `--force-so-throw`.

For filenames starting with a dash, use the POSIX option `--` to specify that all subsequent arguments are treated as filenames, not as command switches. For example, to link a file named `-ifile_1`, use: `armlink -- -ifile_1`.

——— Note ———

You can use special characters to select multiple symbolic names in some `armlink` arguments:

- The wildcard character * can be used to match any name.
- The wildcard character ? can be used to match any single character.

If you are using a special character on Unix platforms, you must enclose the options in quotes to prevent the shell expanding the selection.

———————

See also:

- *Invoking the ARM linker* on page 2-11
- *Ordering command-line options* on page 2-17
- *Specifying command-line options with an environment variable* on page 2-17
- *Reading command-line options from a file* on page 2-17.

2.2.1 Invoking the ARM linker

The command for invoking the ARM linker is:

```
armlink [help-options] [license-option] [project-template-options]
[library-options] [linker-control-options] [output-options] [memory-map-options]
[debuq-options] [image-content-options] [veneer-generation-options]
[byte-addressing-mode] [image-info-options] [diagnostic-options]
[via-file-options] [miscellaneous-options] [arm-linux-options] [input-file]
```

See Chapter 2 *Linker Command-line Options* in the *Linker Reference Guide* for more information on each of the following options:

help-options

Shows the main command-line options, the version number of the compiler and how the compiler has processed the command line:

- *--help* on page 2-43
- *--show_cmdline* on page 2-80
- *--version_number* on page 2-98
- *--vsn* on page 2-100.

license-option

Use the following option to make additional attempts to get a floating license:

- *--licretry* on page 2-56.

project-template-options

Project templates are files containing project information such as command-line options for a particular configuration. These files are stores in the project template working directory.

Use the following options to control the use of project templates:

- *--project=filename, --no_project* on page 2-69
- *--reinitialize_workdir* on page 2-72
- *--workdir=directory* on page 2-100.

library-options

Use the following options to control the library file and paths:

- *--add_needed, --no_add_needed* on page 2-6
- *--add_shared_references, --no_add_shared_references* on page 2-7
- *--libpath=pathlist* on page 2-53
- *--library=name* on page 2-54

- *--library_type=lib* on page 2-55
- *--reduce_paths, --no_reduce_paths* on page 2-70
- *--runpath=pathlist* on page 2-76
- *--scanlib, --no_scanlib* on page 2-77
- *--search_dynamic_libraries, --no_search_dynamic_libraries* on page 2-78
- *--thumb2_library, --no_thumb2_library* on page 2-91
- *--userlibpath=pathlist* on page 2-95.

linker-control-options

Use the following options to control how objects are linked:

- *--match=crossmangled* on page 2-60
- *--strict* on page 2-84
- *--strict_ph, --no_strict_ph* on page 2-85
- *--strict_relocations, --no_strict_relocations* on page 2-86
- *--unresolved=symbol* on page 2-94.

output-options

The following options specify the format and name of the output file:

- *--base_platform* on page 2-14
- *--bpabi* on page 2-17
- *--combreloc, --no_combreloc* on page 2-22
- *--dll* on page 2-30
- *--output=file* on page 2-62
- *--partial* on page 2-64
- *--prelink_support, --no_prelink_support* on page 2-68
- *--reloc* on page 2-72
- *--shared* on page 2-80
- *--sysv* on page 2-90.

memory-map-options

Use the following options to specify memory maps:

- *--autoat, --no_autoat* on page 2-14
- *--fpic* on page 2-42
- *--predefine="string"* on page 2-66
- *--ro_base=address* on page 2-74
- *--ropi* on page 2-75
- *--rosplit* on page 2-75

- *--rw_base=address* on page 2-76
- *--rwpi* on page 2-77
- *--scatter=file* on page 2-78
- *--split* on page 2-83.

debug-options

Use the following options to control debug information in the image:

- *--bestdebug, --no_bestdebug* on page 2-16
- *--compress_debug, --no_compress_debug* on page 2-23
- *--debug, --no_debug* on page 2-26
- *--dynamic_debug* on page 2-30
- *--emit_debug_overlay_relocs* on page 2-32
- *--emit_debug_overlay_section* on page 2-32
- *--emit_non_debug_relocs* on page 2-33.

image-content-options

Use the following options to control miscellaneous factors affecting the image content:

- *--any_contingency* on page 2-8
- *--any_placement=algorithm* on page 2-8
- *--any_sort_order=order* on page 2-10
- *--arm_linux* on page 2-11
- *--arm_only* on page 2-13
- *--as_needed, --no_as_needed* on page 2-13
- *--branchnop, --no_branchnop* on page 2-17
- *--comment_section, --no_comment_section* on page 2-23
- *--cppinit, --no_cppinit* on page 2-23
- *--cpu=list* on page 2-24
- *--cpu=name* on page 2-24
- *--datacompressor=opt* on page 2-25
- *--device=list* on page 2-27
- *--device=name* on page 2-27
- *--dynamic_linker=name* on page 2-30
- *--edit=file_list* on page 2-31
- *--emit_relocs* on page 2-33
- *--entry=location* on page 2-33
- *--exceptions, --no_exceptions* on page 2-35
- *--exceptions_tables=action* on page 2-35

- *--execstack, --no_execstack* on page 2-36
- *--export_all, --no_export_all* on page 2-36
- *--export_dynamic, --no_export_dynamic* on page 2-37
- *--filtercomment, --no_filtercomment* on page 2-39
- *--fini=symbol* on page 2-40
- *--first=section_id* on page 2-40
- *--force_explicit_attr* on page 2-41
- *--force_so_throw, --no_force_so_throw* on page 2-41
- *--fpu=list* on page 2-42
- *--fpu=name* on page 2-42
- *--gnu_linker_defined_syms* on page 2-43
- *--init=symbol* on page 2-47
- *--inline, --no_inline* on page 2-47
- *--keep=section_id* on page 2-49
- *--keep_protected_symbols* on page 2-51
- *--largeregions, --no_largeregions* on page 2-51
- *--last=section_id* on page 2-52
- *--linux_abitag=version_id* on page 2-56
- *--locals, --no_locals* on page 2-58
- *--ltcg* on page 2-59
- *--max_visibility=type* on page 2-61
- *--merge, --no_merge* on page 2-61
- *--muldefweak, --no_muldefweak* on page 2-62
- *--override_visibility* on page 2-63
- *--pad=num* on page 2-63
- *--pltgot=type* on page 2-65
- *--pltgot_opts=mode* on page 2-66
- *--privacy* on page 2-68
- *--profile=filename* on page 2-69
- *--ref_cpp_init, --no_ref_cpp_init* on page 2-71
- *--remove, --no_remove* on page 2-74
- *--soname=name* on page 2-81
- *--sort=algorithm* on page 2-81
- *--startup=symbol, --no_startup* on page 2-83
- *--strict_flags, --no_strict_flags* on page 2-85
- *--symbolic* on page 2-89
- *--symver_script=file* on page 2-90

- *--symver_soname* on page 2-90
- *--tailreorder, --no_tailreorder* on page 2-91
- *--tiebreaker=option* on page 2-92
- *--undefined=symbol* on page 2-93
- *--undefined_and_export=symbol* on page 2-93
- *--use_definition_visibility* on page 2-95
- *--vfemode=mode* on page 2-98.

veneer-generation-options

Use the following options to control veneer generation:

- *--crosser_veneershare, --no_crosser_veneershare* on page 2-25
- *--inlineveneer, --no_inlineveneer* on page 2-48
- *--piveneer, --no_piveneer* on page 2-64
- *--veneer_inject_type=type* on page 2-96
- *--veneer_pool_size=size* on page 2-97
- *--veneershare, --no_veneershare* on page 2-97.

byte-addressing-mode

Use the following options to control byte addressing mode:

- *--be8* on page 2-15
- *--be32* on page 2-16.

image-info-options

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.

Use the following options to control how to extract and present information about the image:

- *--callgraph, --no_callgraph* on page 2-18
- *--callgraph_file=filename* on page 2-19
- *--callgraph_output=fmt* on page 2-20
- *--cgfile=type* on page 2-21
- *--cgsymbol=type* on page 2-21
- *--cgundefined=type* on page 2-22
- *--feedback=file* on page 2-38
- *--feedback_image=option* on page 2-38
- *--feedback_type=type* on page 2-39
- *--info=topic[,topic,...]* on page 2-44
- *--info_lib_prefix=opt* on page 2-46

- *--list_mapping_symbols, --no_list_mapping_symbols* on page 2-57
- *--load_addr_map_info, --no_load_addr_map_info* on page 2-57
- *--mangled, --unmangled* on page 2-59
- *--map, --no_map* on page 2-59
- *--section_index_display=type* on page 2-79
- *--symbols, --no_symbols* on page 2-89
- *--symdefs=file* on page 2-89
- *--xref, --no_xref* on page 2-101
- *--xrefdbg, --no_xrefdbg* on page 2-101
- *--xref{from|to}=object(section)* on page 2-101

diagnostic-options

Use the following options to control how the linker emits diagnostics:

- *--diag_error=tag[,tag,...]* on page 2-28
- *--diag_remark=tag[,tag,...]* on page 2-28
- *--diag_style=arm|ide|gnu* on page 2-28
- *--diag_suppress=tag[,tag,...]* on page 2-29
- *--diag_warning=tag[,tag,...]* on page 2-29
- *--errors=file* on page 2-34
- *--list=file* on page 2-56
- *--remarks* on page 2-73
- *--strict_enum_size, --no_strict_enum_size* on page 2-84
- *--strict_symbols, --no_strict_symbols* on page 2-87
- *--strict_visibility, --no_strict_visibility* on page 2-88
- *--strict_wchar_size, --no_strict_wchar_size* on page 2-88
- *--verbose* on page 2-98.

via-file-options

Use the following option to specify a via file containing additional command-line arguments to the linker:

- *--via=file* on page 2-100.

miscellaneous-options

- *--legacyalign, --no_legacyalign* on page 2-53.

arm-linux-options

Use the following options for ARM Linux applications:

- *--add_needed, --no_add_needed* on page 2-6
- *--arm_linux* on page 2-11

- *--dynamic_linker=name* on page 2-30
- *--library=name* on page 2-54
- *--linux_abitag=version_id* on page 2-56
- *--runpath=pathlist* on page 2-76
- *--search_dynamic_libraries, --no_search_dynamic_libraries* on page 2-78.

input-file This is a space-separated list of objects, libraries, or *symbol definitions* (symdefs) files. See *input_file_list* on page 2-48 for more information.

2.2.2 Ordering command-line options

In general, command-line options can appear in any order. However, the effects of some options depend on how they are combined with other related options.

Where options override previous options on the same command line, the last one found takes precedence. Where an option does not follow this rule, this is noted in the description. Use `--show_cmdline` to see how options are processed from the command line. The commands are shown normalized, and the contents of any via files are expanded.

See *input_file_list* on page 2-48 in the *Linker Reference Guide* for more information on the ordering priority of input files.

2.2.3 Specifying command-line options with an environment variable

You can specify command-line options by setting the value of the `RVCT`*`ver`*`_LINKOPT` environment variable. The syntax is identical to the command line syntax. The linker reads the value of `RVCT`*`ver`*`_LINKOPT` and inserts it at the front of the command string. This means that options specified in `RVCT`*`ver`*`_LINKOPT` can be overridden by arguments on the command-line.

See also *Using command-line options* on page 2-10.

2.2.4 Reading command-line options from a file

When the operating system restricts the command line length, you can include additional command-line options in a file with the compiler option:

```
--via filename
```

The compiler opens the specified file and reads additional command-line options from it.

See Appendix A *Via File Syntax* in the *Compiler Reference Guide* for more information.

Chapter 3
Using the Basic Linker Functionality

This chapter describes the basic functionality available in the ARM® linker, `armlink` provided with ARM RealView® Compilation Tools. It contains the following sections:

3.1 The image structure

The structure of an image is defined by the:

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

Each link stage has a different view of the image:

ELF object file view (linker input)

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

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

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

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

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

ELF image file view (linker output)

The ELF image file view comprises Program Segments and output sections:

- a load region corresponds to a Program Segment

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

One or more execution regions make up a load region.

——— **Note** ———

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

———————————

When describing a memory view:

- the term *root region* is used to describe a region that has the same load and execution addresses
- load regions are equivalent to ELF segments.

Figure 3-1 on page 3-4 shows the relationship between the views at each link stage.

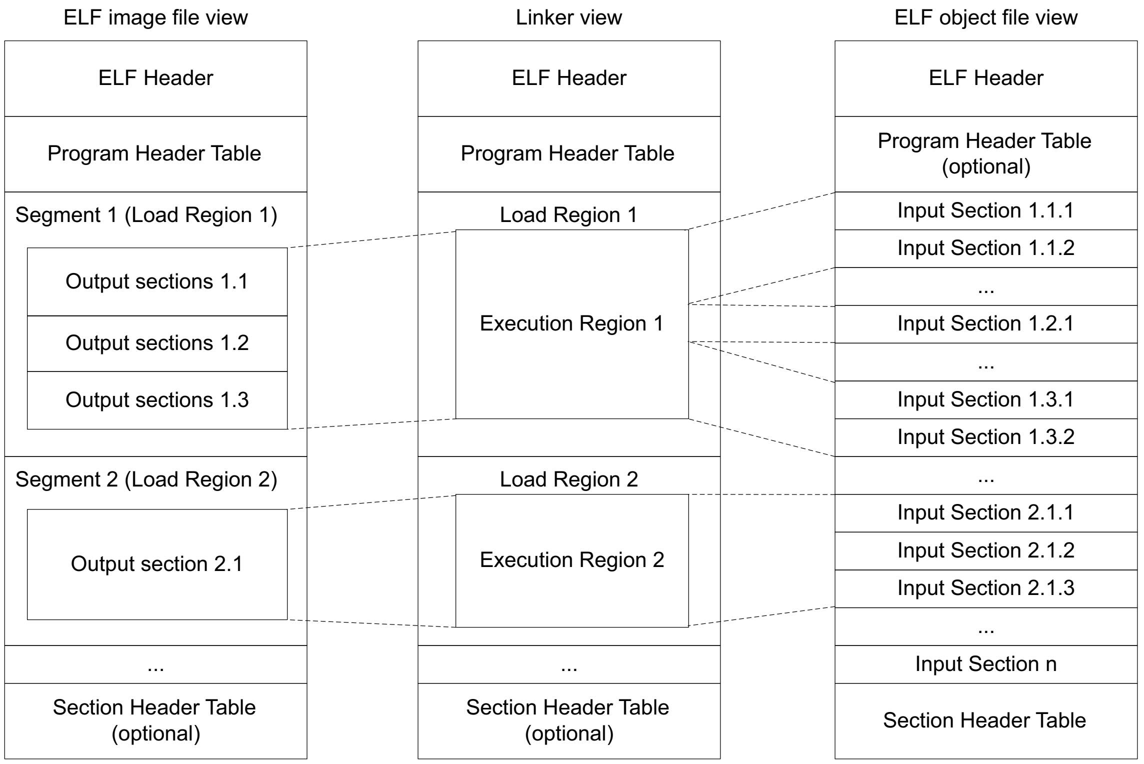


Figure 3-1 Relationship between sections, regions, and segments

See also:

- *Input sections, output sections, regions, and Program Segments*
- *Load view and execution view of an image* on page 3-5
- *Specifying an image memory map* on page 3-7
- *Image entry points* on page 3-8.

3.1.1 Input sections, output sections, regions, and Program Segments

An object or image file is constructed from a hierarchy of input sections, output sections, regions, and Program Segments:

Input section

An input section is an individual section from an input object file. It contains code, initialized data, or describes a fragment of memory that is not initialized or that must be set to zero before the image can execute.

These properties are represented by attributes such as RO, RW and ZI. These attributes are used by `armlink` to group input sections into bigger building blocks called output sections and regions.

Output section

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

Region A region is a contiguous sequence of one, two, or three output sections depending on the contents of the number of sections with different attributes. The output sections in a region are sorted according to their attributes. The RO output section is first, then the RW output section, and finally the ZI output section. A region typically maps onto a physical memory device, such as ROM, RAM, or peripheral.

Program Segment

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

Note

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

See also

- *The image structure* on page 3-2
- *Load view and execution view of an image*
- *Specifying an image memory map* on page 3-7
- *Image entry points* on page 3-8.

3.1.2 Load view and execution view of an image

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

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

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

Figure 3-2 shows these views.

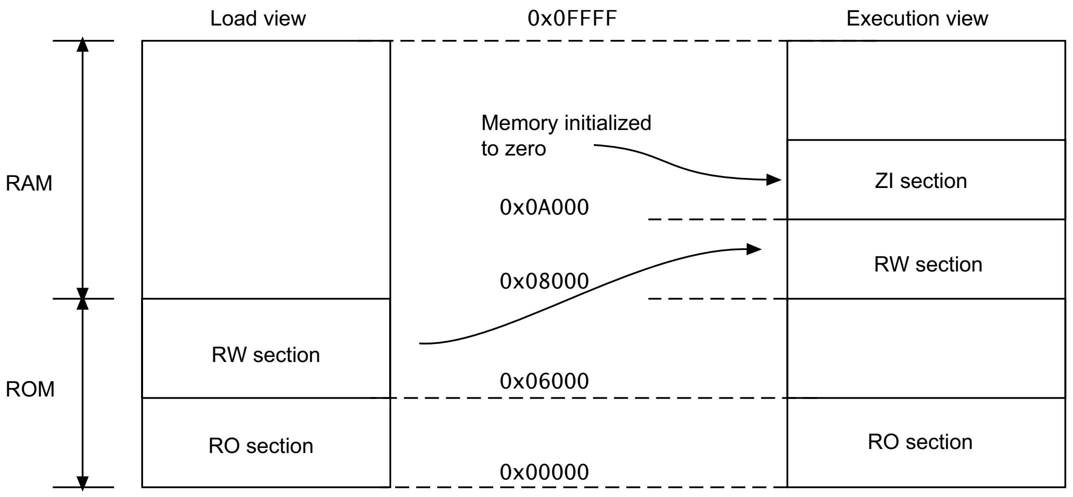


Figure 3-2 Load and execution memory maps

Table 3-1 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 region in the load address space	Execution region	A region in the execution address space

See also

- *The image structure* on page 3-2
- *Input sections, output sections, regions, and Program Segments* on page 3-4
- *Specifying an image memory map* on page 3-7
- *Image entry points* on page 3-8.

3.1.3 Specifying an image memory map

An image can consist of any number of regions and output sections. Any number of these regions can have different load and execution addresses. To construct the memory map of an image, `armlink` must have information about:

Grouping How input sections are grouped into output sections and regions.

Placement Where image regions are to be located in the memory maps.

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

Using command-line options

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

- `--ro-base`
- `--rw-base`
- `--ropi`
- `--rwpi`
- `--first`
- `--last`
- `--split`
- `--rosplit`.

The options listed above provide a simplified notation that gives the same settings as a scatter-loading description for a simple image. For more information, see *Using command-line options to create simple images* on page 3-30.

Using a scatter-loading description file

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

——— **Note** ———

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

———————————

For more information, see Chapter 5 *Using Scatter-loading Description Files*.

Table 3-2 shows the equivalent command-line options to use to give a similar memory map to that defined by a scatter-loading file.

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

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

3.1.4 Image entry points

An entry point in an image is a location where program execution can start. There are two distinct types of entry point:

Initial entry point

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

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

Entry points set by the ENTRY directive

These are entry points that are set in the assembly language sources with the `ENTRY` directive. In embedded systems, this directive is typically used to mark code that is entered through the processor exception vectors, such as RESET, IRQ, and FIQ.

You can specify multiple entry points in an image with the `ENTRY` directive. 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.

See the *Assembler Guide* for more information on the `ENTRY` directive.

If an embedded image is to be used by a loader, it must have a single initial entry point specified in the header. See *Specifying an initial entry point* for more information.

Specifying an initial entry point

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

For embedded applications with ROM at zero use `--entry 0x0` (or optionally `0xFFFF0000` for CPUs that have high vectors).

The initial entry point must meet the following conditions:

- the image entry point must always lie within an execution region
- the execution region must be non-overlay, and must be a root execution region (the load address is the same as the execution address).

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

- if the input objects contain only one entry point set by the `ENTRY` directive, the linker uses that entry point as the initial entry point for the image
- the linker generates an image that does not contain an initial entry point when either:
  - more than one entry point has been specified by using the `ENTRY` directive
  - no entry point has been specified by using the `ENTRY` directive.

For more information, see:

- *--entry=location* on page 2-33 in the *Linker Reference Guide*
- *ENTRY* on page 7-76 in the *Assembler Guide*.

3.2 Section placement

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

1. By attribute as follows:
   a. read-only code
   b. read-only data
   c. read-write code
   d. read-write data
   e. zero-initialized data.
2. By input section name if they have the same attributes. Names are considered to be case-sensitive and are compared in alphabetical order using the ASCII collation sequence for characters.
3. By their relative positions in the input file if they have the same attributes and section names, except where overridden by `FIRST` or `LAST`. See *Using FIRST and LAST to place sections* on page 3-11.

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

Note

This sorting order is unaffected by ordering within scatter-loading description files or object file names.

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

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

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

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

See also:

- *Handling unassigned sections*
- *Using FIRST and LAST to place sections*
- *Aligning sections* on page 3-12
- *Ordering execution regions containing Thumb code* on page 3-12.

3.2.1 Handling unassigned sections

The linker might not be able to place some input sections in any execution region. When this happens, the linker generates an error message. This might occur because your current scatter-loading description file does not allow for all possible module select patterns and input section selectors. How you fix this depends on how important it is that these sections are placed correctly:

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

When the linker creates an input section for each ELF section in the input objects, it increments a global counter. The value of this counter is stored in the section as the creation index. The creation index is used when placing any unassigned sections that have identical properties.

3.2.2 Using FIRST and LAST to place sections

You can make sure that a section is placed either first or last in its execution region. For example, you might want to make sure the section containing the vector table is placed first in the image. To do this, use one of the following methods:

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

  However, `FIRST` and `LAST` must not violate the basic attribute sorting order as described in *Section placement* on page 3-10. For example, `FIRST RW` is placed after any read-only code or read-only data.

3.2.3 Aligning sections

When input sections have been ordered and before the base address is 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 ARM 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_legacy_align` is used the region alignment is the maximum alignment of any input section contained by the region.

It is possible to use `ALIGN` to expand the alignment of a region, for example, changing something that is normally four-byte aligned to be eight-byte aligned. You cannot reduce the natural alignment, for example, forcing two-byte alignment on something that is normally four-byte aligned. See *Creating regions on page boundaries* on page 5-45 for more information.

You can also increase the alignment of an input section. See the description of `ALIGN` in the directives reference in the *Assembler Guide*.

3.2.4 Ordering execution regions containing Thumb code

The Thumb branch range is 4MB. When an execution region contains Thumb code that exceeds 4MB, `armlink` attempts to order sections that are at a similar average call depth and to place the most commonly called sections centrally. This helps to minimize the number of veneers generated. See *Veneers* on page 3-23 for more information.

The Thumb-2 branch range is 16MB. Section re-ordering is only required if that limit is exceeded.

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

3.3 Section elimination

This section describes linker optimizations for duplicate sections, unused sections, debug sections.

See also:

- *Common debug section elimination*
- *Common group or section elimination*
- *Unused section elimination* on page 3-14
- *Unused virtual function elimination* on page 3-15.

3.3.1 Common debug section elimination

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.

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

3.3.2 Common group or section elimination

If there are inline functions or templates used in the C++ source, the ARM compiler generates complete objects for linking such that each object contains the out-of-line copies of inline functions and 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. In order 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 different 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.

This optimization is controlled by linker options:

- Use the `--bestdebug` option to use the largest 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`.

  For more information, see:

  — *Function inlining* on page 5-18 in the *Compiler User Guide*

  — *--debug, --no_debug* on page 2-37 in the *Compiler Reference Guide.*

3.3.3 Unused section elimination

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

Unused section elimination is suppressed in cases that might result in the removal of all sections.

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

Unused section elimination requires and entry point. Therefore, if no entry point is 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 have been eliminated.

Unused section elimination is suppressed in those cases that might result in the removal of all sections.

An input section is retained in the final image under the following conditions:

- if it contains an entry point
- if it is referred to, directly or indirectly, by a non-weak reference from an input section containing an entry point

- if it is specified as the first or last input section by the `--first` or `--last` option (or a scatter-loading equivalent)
- if it is marked as unremovable by the `--keep` option.

Note

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

You can 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, and ensures that the function or variable is not removed by the linker.

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.

3.3.4 Unused virtual function elimination

Unused section elimination efficiently removes unused functions from C code. In C++ applications, virtual functions and RTTI objects are referenced by pointer tables, known as vtables. Without extra information, the linker cannot determine which vtable entries will be 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. This optimization can be used 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. In the text that follows, you can assume that functions are compiled in their own sections.

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

——— **Note** ———

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

———

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

See also:

- *--vfemode=mode* on page 2-98 in the *Linker Reference Guide*
- *--rtti, --no_rtti* on page 2-114 in the *Compiler Reference Guide*.

3.4 Feedback

This section describes how to use linker feedback for repeat compilations.

`armlink` provides feedback for the next time a file is compiled, to inform the compiler about unused functions. These are placed in their own sections for future elimination by the linker.

When `--inline` optimization is turned on (see *Inlining* on page 3-26), functions inlined by the linker are also emitted in the feedback file. These functions are also placed in their own sections.

The `--feedback` *`file`* option generates a feedback file containing each output filename, as a comment, and the unused symbols found in the file, for example:

```
;#<FEEDBACK># ARM Linker, RVCT ver [Build num]: Last Updated: Date
;VERSION 0.2
;FILE foo.o
unused_func1 <= USED 0
inlined_func <= LINKER_INLINED
;FILE bar.o
unused_func2 <= USED 0
```

When you next compile the source, use the compiler option `--feedback` *`file`* to specify the linker-generated feedback file to use. If no feedback file exists, the compiler issues a warning message.

See also:

- *Example of linker feedback*.

3.4.1 Example of linker feedback

To see how linker feedback works:

1. Create a file `fb.c` containing the code shown in Example 3-1.

Example 3-1 Feedback 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()
{
    int n = 3;
    printf("%d cubed = %d\n",n,cubed(n));
}
```

2. Compile the program (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, RVCT ver [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 main `.text` area. 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.

3.5 RW data compression

This section describes linker optimizations for data compression.

RW data areas typically contain a large number of repeated values, for example, zeros making them suitable for compression. RW data compression is enabled by default to minimize ROM size.

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. However, you can override the algorithm chosen by the linker.

See also:
- *Choosing a compressor*
- *How is compression applied?* on page 3-21
- *Working with RW data compression* on page 3-21.

3.5.1 Choosing a compressor

`armlink` gathers information about the content of data sections before choosing the most appropriate compression algorithm to generate the smallest image. If compression is appropriate, the linker can only use one data compressor for all the compressible data sections in the image and different compressions 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
```

Once 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.

You can override the compression used by the linker by either:
- using the `--datacompressor off` option to turn off compression
- specifying a compressor of your choosing.

Use the command-line option `--datacompressor list` to get a list of compressors available in the linker, for example:

```
Num     Compression algorithm
========================================================
0       Run-length encoding
1       Run-length encoding, with LZ77 on small-repeats
2       Complex LZ77 compression
```

3.5.2 How is compression applied?

Run-length compression encodes data as non-repeated bytes and repeated zero-bytes. Non-repeated bytes are output unchanged, followed by a count of zero-bytes. Limpel-Ziv 1977 (LZ77) compression keeps track of the last n bytes of data seen and, when a phrase is encountered that has already been seen, it outputs a pair of values corresponding to the position of the phrase in the previously-seen buffer of data, and the length of the phrase.

To specify a compressor, use the required ID on the linker command line, for example:

```
armlink --datacompressor 2 ...
```

When choosing a compressor 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 (see *Choosing a compressor* on page 3-20).

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.

3.5.3 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 does not apply compression if a `Load$$`*`region_name`*`$$Base` symbol is used, where *`region_name`* follows any execution region containing compressed data in the same load region.
- If you are using an ARM processor with on-chip cache, enable the cache after decompression to avoid code coherency problems.

See Chapter 3 *Embedded Software Development* in the *Developer Guide* for more information.

RW data compression uses library code, for example `__dc*.o` that must be placed in a root region. This is best done using `InRoot$$Sections` in a scatter-loading description file.

If you are using a scatter-loading description file, specify that a load or execution region must not be compressed by adding the `NOCOMPRESS` attribute. See Chapter 3 *Formal syntax of the scatter-loading description file* for more information.

3.6 Veneers

Veneers are small sections of code generated by the linker and inserted into your program. `armlink` must generate veneers when a branch involves a destination beyond the branching range of the current state.

The range of a `BL` instruction is 32MB for ARM, 16MB for Thumb-2, and 4MB for Thumb. A veneer can, therefore, extend the range of the branch by becoming the intermediate target of the instruction and then setting the PC to the destination address. If ARM and Thumb are mixed, the veneer also changes processor state.

`armlink` supports the following veneer types:

- ARM to ARM
- ARM to Thumb (interworking veneers)
- Thumb to ARM (interworking veneers)
- Thumb to Thumb.

`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. 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. In ARMv5 and above, the `BLX` instruction is used.

See also:

- *Veneer sharing*
- *Veneer variants* on page 3-24
- *PI to absolute veneers* on page 3-25
- *Reuse of veneers with overlay execution regions* on page 3-25.

3.6.1 Veneer sharing

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-loading description 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`, therefore, 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.

3.6.2 Veneer variants

Veneers have different variants depending on the branching range you require:

- Inline veneers:
  - used to perform 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 and so the function code must appear in the first 256 bytes immediately after the veneer code
  - a Thumb-ARM interworking veneer has a range of zero bytes and so the function code must appear immediately after the veneer code
  - an inline veneer is always position-independent.
- Short branch veneers:
  - 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.
- Long branch veneers:
  - 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-loading description 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 order of preference:

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

3.6.3 PI to absolute veneers

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

The generation of PI to absolute veneers can be controlled using the `--piveneer` option, which 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.

3.6.4 Reuse of veneers with overlay execution regions

A scatter-loading description file enables overlay regions to be created, that is, regions that share the same area of RAM. The linker reuses veneers whenever possible, and there are some limitations on the reuse of veneers in overlaid regions. However, both the following conditions are enforced on reuse:

- an overlay execution region cannot reuse a veneer placed in any other overlay execution region
- no other execution region can reuse a veneer placed in an overlay execution region.

If these conditions are not met, new veneers are created instead of reusing existing ones. Unless you have instructed the linker to place veneers somewhere specific using scatter-loading, a veneer is always placed in the execution region that contains the call requiring the veneer. This implies that:

- for an overlay execution region, all its veneers are included within the execution region
- an overlay execution region never requires a veneer from another execution region.

3.7 Inlining

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

Use the following command-line options to control branch inlining:

- *--branchnop, --no_branchnop* on page 2-17
- *--inline, --no_inline* on page 2-47
- *--tailreorder, --no_tailreorder* on page 2-91.

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 callee function.

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. See *Unused section elimination* on page 3-14 for more information.

Use the `--info=inline` command-line option to list all the inlined functions. See *--info=topic[,topic,...]* on page 2-44 in the *Linker Reference Guide* for more information.

See also:

- *Factors that influence inlining*
- *Handling tail calling sections* on page 3-27.

3.7.1 Factors that influence inlining

The following factors influence the way functions are inlined:

- The linker handles only the simplest cases and does not inline any instruction 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 other state must be built for interworking. The linker can inline functions containing up to two 16-bit Thumb instructions. However, an ARM caller can only inline functions containing a single 16-bit Thumb instruction or 32-bit Thumb-2 instruction.

- The action that the linker takes depends on the size of the callee function and the state of both the caller and the callee as shown in Table 3-3.

Table 3-3 Inlining small functions

Caller state	Callee state	Callee function size
ARM	ARM	4 to 8 bytes
ARM	Thumb	2 to 6 bytes
Thumb	Thumb	2 to 6 bytes

- In order to be inlined, the last instruction of a function must be either:

  `MOV pc, lr`

  or

  `BX lr`

  A function that consists of just 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 an IT block cannot inline a 16-bit 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.

3.7.2 Handling tail calling sections

As described in *Factors that influence inlining* on page 3-26, the linker replaces any branch that comprises a relocation that resolves to the next instruction with a `NOP`. This means that tail calling sections, that is, sections that finish with a branch instruction, might be optimized so that their target appears immediately after them in the execution region, and with the branch changed to a `NOP`.

You can take advantage of this behavior by using the command-line option `--tailreorder` to move tail calling sections immediately before their target. See *--tailreorder, --no_tailreorder* on page 2-91 in the *Linker Reference Guide* for more information.

Use the `--info=tailreorder` command-line option to display information about tail call optimization. See *--info=topic[,topic,...]* on page 2-44 in the *Linker Reference Guide* for more information.

3.8 About merging comment sections

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

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

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

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

Note

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

3.8.1 See also

the following in the *Linker Reference Guide*:

- *--comment_section, --no_comment_section* on page 2-23
- *--filtercomment, --no_filtercomment* on page 2-39.

3.9 Using command-line options to create simple images

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

Type 1 One region in load view, three contiguous regions in execution view. Use the `--ro-base` option to create this type of image.

See *Type 1, one load region and contiguous execution regions* for more information.

Type 2 One region in load view, three non-contiguous regions in execution view. Use the `--ro-base` and `--rw-base` options to create this type of image.

See *Type 2, one load region and non-contiguous execution regions* on page 3-32 for more information.

Type 3 Two regions in load view, three non-contiguous regions in execution view. Use the `--ro-base`, `--rw-base`, and `--split` options to create this type of image. You can also use the `--rosplit` option to split the default load region into two RO output sections, one for code and one for data.

See *Type 3, two load regions and non-contiguous execution regions* on page 3-34 for more information.

In all three simple image types, there are up to three execution regions where:

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

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

Simple images can also be created with scatter-loading description files. See *Equivalent scatter-loading descriptions for simple images* on page 5-48 for more information on how to do this.

3.9.1 Type 1, one load region and contiguous execution regions

An image of this type consists of a single load region in the load view and three execution regions 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 (see Figure 3-3 on page 3-31).

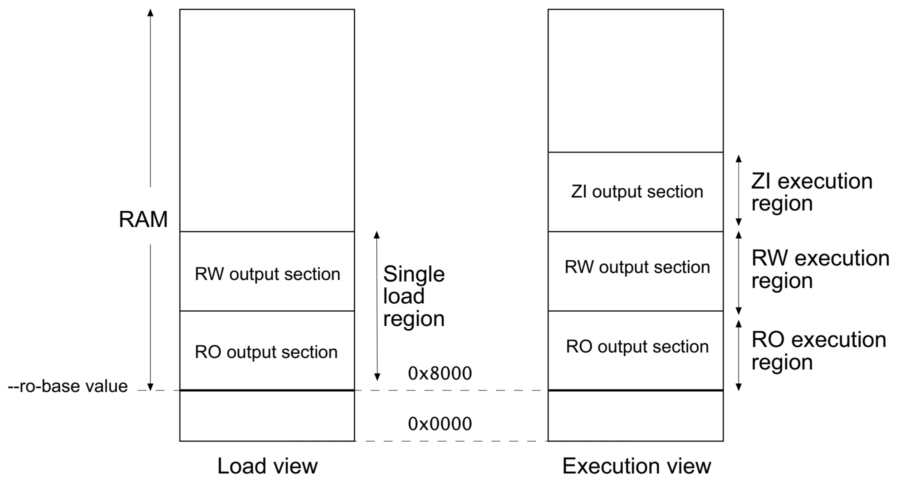


Figure 3-3 Simple type 1 image

Use the following command for images of this type:

```
armlink --ro-base 0x8000
```

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 execution regions are the same as their load addresses, so nothing has to be moved from its load address to its execution address. However, the ZI execution region that contains the ZI output section is created at run-time.

Use `armlink` option `--ro-base` *`address`* to specify the load and execution address of the region containing the RO output. The default address is `0x8000` as shown in Figure 3-3.

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

An image of this type 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 (see Figure 3-4), where RW data is copied from ROM to RAM at startup.

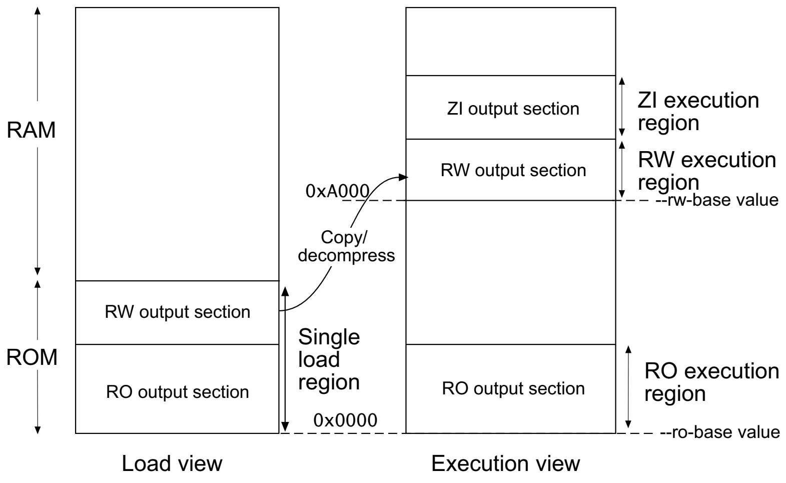


Figure 3-4 Simple type 2 image

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` *`exec_address`* to specify the execution address of the RW output section. If you do not use the `--ro-base` option to specify the address, the default value of `0x8000` is used by `armlink`. For an embedded system, `0x0` is typical for the `--ro-base` value. If you do not use the `--rw-base` option to specify the address, the default is to place RW directly above RO (as in *Type 1, one load region and contiguous execution regions* on page 3-30).

Note

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

3.9.3 Type 3, two load regions and non-contiguous execution regions

This type of image is similar to images of type 2 except that the single load region is now split into two root load regions (see Figure 3-5).

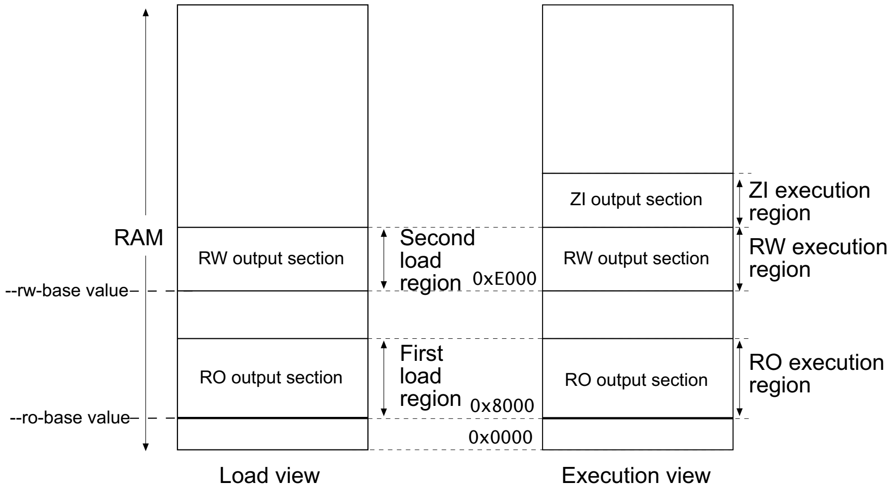


Figure 3-5 Simple type 3 image

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 and the second execution region contains the RW and ZI output sections.

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

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

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

`--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 and one containing the RW output section) so that they can be placed separately using `--ro-base` and `--rw-base`.

`--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.

3.10 Using command-line options to handle C++ exceptions

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 sections are present after unused sections have been eliminated.

You can use the `--no_exceptions` option if you want 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 --exception_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 cannot generate cleanup code for automatic variables in C++ code that is compiled without exceptions support, for example, any code compiled with RealView Compilation Tools prior to v2.1, or code compiled using the `--no_exceptions` option.

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

3.11 About weak references and definitions

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

Weak references

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

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

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

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

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

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

An unresolved weak function call is replaced with either:

- A no-operation instruction, `NOP`.

- A branch with link instruction, `BL`, to the following instruction. That is, the function call just does not happen.

Weak definitions

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

A weak definition can be used 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.

See also:

- *Example of a weak reference*
- *Example of a weak definition* on page 3-40
- *Library searching, selection, and scanning* on page 3-44
- the following in the *Linker Reference Guide*:
  - *--feedback=file* on page 2-38
  - *--keep=section_id* on page 2-49
  - *--remove, --no_remove* on page 2-74.
- the following in the *Compiler Reference Guide*:
  - *--split_sections* on page 2-118
  - *__weak* on page 4-21
  - *__attribute__((weak))* on page 4-41
  - *__attribute__((weakref("target")))* on page 4-41
  - *__attribute__((weak))* on page 4-55
  - *__attribute__((weakref("target")))* on page 4-56
  - *#pragma arm section [section_sort_list]* on page 4-59.
- the following in the *Assembler Guide*:
  - *NOP, SEV, WFE, WFI, and YIELD* on page 4-142
  - *B, BL, BX, BLX, and BXJ* on page 4-115
  - *ENTRY* on page 7-76
  - *EXPORT or GLOBAL* on page 7-78.

3.11.1 Example of a weak reference

A library contains a function `foo()`, that is called in some builds of an application but not in others. If it is used, `init_foo()` must be called first. Weak references can be used 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, also allowing this to 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 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
```

See also *About weak references and definitions* on page 3-37.

3.11.2 Example of a weak definition

A simple or dummy implementation of a function can be provided as a weak definition. This enables you to built 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.

See also *About weak references and definitions* on page 3-37.

3.12 Getting information about images

You can use the `--info` option to get information about how your image is generated by the linker, for example:

```
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`.

See *--info=topic[,topic,...]* on page 2-44 in the *Linker Reference Guide* for more information.

Example 3-2 shows the output in tabular format with the totals separated out for easy reading.

Example 3-2 Image information

```
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)
```

Code (inc. Data)

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

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

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

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

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

Object Totals

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

(incl. Generated)

`armlink` might generate image contents, for example, interworking veneers, and input sections such as region tables. If the `Object Totals` row includes this type of data, it is shown in this row. In Example 3-2 on page 3-41 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 (see *Aligning sections* on page 3-12). 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 Example 3-2 on page 3-41, 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 Example 3-2 on page 3-41 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 Example 3-2 on page 3-41, RW data compression is not enabled. If data is compressed, the RW value changes. See *RW data compression* on page 3-20 for more information.

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.

3.12.1 Using image-related information

You can use the `--map` option to create an image map. This includes the address and size of each load region, execution region, and input section in the image, and shows how RW data compression is applied. See *--map, --no_map* on page 2-59 in the *Linker Reference Guide* for more information.

You can use `--info inputs` to identify the source of some link errors. For example, you can search the output to locate undefined references from library objects or multiply defined symbols caused by retargeting some library functions and not others. Search backwards from the end of this output to find and resolve link errors. The `--verbose` option can also be used to output similar text with additional information on the linker operations.

3.13 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 are:

- 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, it 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.

Unused sections are subsequently eliminated unless `--no_remove` or `--keep` is used.

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

See also:
- *Searching for ARM libraries* on page 3-45
- *Searching for user libraries* on page 3-47
- *Scanning the libraries* on page 3-48.

3.13.1 Searching for ARM libraries

You can control how the linker searches for the ARM standard libraries with the `RVCT40LIB` environment variable or the `--libpath` command-line option.

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`*.

Using the `RVCT40LIB` environment variable

Use the environment variable `RVCT40LIB` to specify a library path. This is the default.

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 `RVCT40LIB`:

- \ on Windows
- / on Red Hat Linux.

For example, if `RVCT40LIB` is set to *`install_directory`*`\...\lib\`, the linker searches the directories:

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

Using the `--libpath` command-line option

Use the `--libpath` command-line option with a comma-separated list of parent directories. This list must end with the parent directory of the ARM library directories `armlib` and `cpplib`. The `RVCT40LIB` environment variable holds this 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 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, the paths specified by the `RVCT40LIB` 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.

Selecting ARM library variants

RVCT 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.

For more information on library variants, see the Chapter 2 *The C and C++ Libraries* in the *Libraries Guide*.

See also

- *Library searching, selection, and scanning* on page 3-44.

3.13.2 Searching for user libraries

You can specify user libraries by:

- including them explicitly in the input file list
- adding the `--userlibpath` option to the `armlink` command line with a comma-separated list of directories, and the names of the libraries as input files.

You can use the `--library=`*`name`* option to specify static libraries, `lib`*`name`*`.a`, or dynamic shared objects, `lib`*`name`*`.so`. Dynamic searching is controlled by the `--search_dynamic_libraries` option. For example, the following command searches for `libfoo.so` before `libfoo.a`:

```
armlink --arm_linux --shared --fpic --search_dynamic_libraries --library=foo
```

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

The search paths used for the ARM standard libraries specified by the `RVCT40LIB` environment variable or the linker command-line option `--libpath` are not searched for user libraries (see *Searching for ARM libraries* on page 3-45).

See also

- *Library searching, selection, and scanning* on page 3-44.

3.13.3 Scanning the libraries

When the linker has constructed the list of libraries, it repeatedly scans each library in the list to resolve references. There are two separate lists of files that are maintained. The lists are scanned in the following order to resolve all dependencies:

1. List of system libraries found in the `RVCT40LIB` (`--libpath`) directory. These might also be specified by the `-Jdir[,dir,...]` compiler option.
2. The list of all other files that have been loaded. These might be specified by the `-Idir[,dir,...]` compiler option.

Each list is scanned using the following process:

1. Search all specified directories to select the most compatible library variants.
2. Add the variants to the list of libraries.
3. Scan each of the libraries to load the required members:
    a. For each currently unsatisfied non-weak reference, search sequentially through the list of libraries for a matching definition. The first definition found is marked for step 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 or the behavior is unpredictable.
    b. Load the library members marked in stage 3a. 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.

See also

- *Library searching, selection, and scanning* on page 3-44.

Chapter 4
Accessing Image Symbols

This chapter describes how to reference symbols with the ARM® linker, `armlink`. It contains the following sections:

4.1 About mapping symbols

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

- code and data at literal pool boundaries
- ARM code and Thuimb 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 the modifying an executable image with the `fromelf --elf --strip=localsymbols` command removes all mapping symbols from the image.

See also

- *Symbol naming rules* on page 3-27 in the *Assembler Guide*
- *--list_mapping_symbols, --no_list_mapping_symbols* on page 2-57 in the *Linker Reference Guide*
- *--strict_symbols, --no_strict_symbols* on page 2-87 in the *Linker Reference Guide*
- *--strip=option[,option,...]* on page 2-53 in the *Utilities Guide*
- *--text* on page 2-55 in the *Utilities Guide*.
- *ELF for the ARM Architecture*.

4.2 ARM/Thumb synonyms (deprecated in RVCT v4.0)

The linker enables multiple definitions of a symbol to coexist in an image, only if each definition is associated with a different processor state. `armlink` applies the following rules when a reference is made to a symbol with ARM/Thumb synonyms:

- `B`, `BL`, or `BLX` instructions to a symbol from ARM state resolve to the ARM definition.
- `B`, `BL`, or `BLX` instructions to a symbol from Thumb state resolve to the Thumb definition.

Any other reference to the symbol resolves to the first definition encountered by the linker. In this case, `armlink` displays a warning that specifies the chosen symbol.

4.3 Accessing linker-defined symbols

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

You can import these symbolic addresses and use them as relocatable addresses by your assembly language programs, or refer to them as **`extern`** symbols from your C or C++ source code. See *Importing linker-defined symbols* on page 4-8 for more information.

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.

See also:
- *Region-related symbols*
- *Section-related symbols* on page 4-9.

4.3.1 Region-related symbols

The linker generates the following types of region-related symbols for each region in the image:
- `Image$$`
- `Load$$`
- `Load$$LR$$`.

If you are using a scatter-loading description 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.

Image$$ execution region symbols

Table 4-1 shows the symbols that the linker generates for every execution region present in the image. All symbols in Table 4-1 refer to execution addresses after the C library is initialized.

Table 4-1 Image$$ execution region symbols

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

Load$$ execution region symbols

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

Table 4-2 shows the symbols that the linker generates for every `Load$$` execution region present in the image. All symbols in this table refer to load addresses before the C library is initialized. Be aware of the following:

- The symbols are absolute because section-relative symbols can only have execution addresses.
- The symbols take into account RW compression.
- The symbols do not include ZI output section because it does not exist before the C library is initialized.
- All relocations from RW compressed execution regions must be performed before compression, because the linker cannot resolve a delayed relocation on compressed data.
- If the linker detects a relocation from a RW-compressed region to a linker-defined symbol that depends on RW compression, then the linker disables compression for that region.
- Any zero bytes written to the file are visible. Therefore, the Limit and Length values must take into account the zero bytes written into the file.

Table 4-2 Load$$ execution region symbols

Symbol	Description
`Load$$`*`region_name`*`$$Base`	Load address of the region.
`Load$$`*`region_name`*`$$Length`	Load 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.

Table 4-2 Load$$ execution region symbols (continued)

Symbol	Description
`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.
`Load$$`*`region_name`*`$$ZI$$Limit`	Load address of the byte beyond the end of the ZI output section in the execution region.

Load$$LR$$ load region symbols

Table 4-3 shows the symbols that the linker generates for every `Load$$LR$$` 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.

Table 4-3 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.

Region name values when not scatter-loading

If you are not using scatter-loading, the linker uses *`region_name`* values of:

- `ER_RO`, for read-only execution regions
- `ER_RW`, for read-write execution regions
- `ER_ZI`, for zero-initialized execution regions.

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.

Using scatter-loading description files

If you are using scatter-loading, the description file names all the execution regions in the image, and provides their load and execution addresses.

If the description file defines both stack and heap, the linker also generates special stack and heap symbols.

See Chapter 5 *Using Scatter-loading Description Files* for more information.

Importing linker-defined symbols

One common use of region-related symbols is to place a heap directly above the ZI region. Example 4-1 shows how to create a retargeted version of `__user_initial_stackheap()` in assembly language. The example assumes that you are using the default one region memory model from the ARM C libraries.

See *__user_initial_stackheap()* on page 2-72 in the *Libraries Guide*.

Example 4-1 Placing the stack and heap above the ZI region

```
    EXPORT __user_initial_stackheap
    IMPORT ||Image$$region_name$$ZI$$Limit||
__user_initial_stackheap
    LDR r0, =||Image$$region_name$$ZI$$Limit||
    MOV pc, lr
```

There are two ways to import linker-defined symbols into your C or C++ source code. Use either:

```
extern unsigned int symbol_name;
```

or:

```
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 Example 4-2 and Example 4-3 on page 4-9.

Example 4-2 Importing linker-defined symbols

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

Example 4-3 Importing linker-defined symbols

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

See also

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

4.3.2 Section-related symbols

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

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

- Image symbols, if you use command-line options to create a simple image. A simple image has three output sections (RO, RW, and ZI) that produce the three execution regions.
- Input section symbols, for every input section present in the image.

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

Image symbols

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

If you are using a scatter-loading description file, the image symbols in Table 4-4 on page 4-10 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-loading description file you must manually place the stack and heap. You can do this either:

- in a scatter-loading description 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.

See Chapter 5 *Using Scatter-loading Description Files* for more information.

Table 4-4 Image-related 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.

Input section symbols

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

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-loading description places these input sections non-contiguously, the linker issues an error. This is because the use of the base and limit symbols over non-contiguous memory usually produces unpredictable and undesirable effects.

Table 4-5 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`.

See also

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

4.4 Accessing symbols in another image

If you want one image to know the global symbol values of another image, you can use a *symbol definitions* (symdefs) file.

This can be used, 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.

See also:

- *Creating a symdefs file*
- *Reading a symdefs file* on page 4-13
- *Symdefs file format* on page 4-13.

4.4.1 Creating a symdefs file

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

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

Note

If *`filename`* does not exist, the file is created containing all the global symbols. If *`filename`* exists, the existing contents of *`filename`* are used to select the symbols that are output when the linker 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.

Outputting a subset of the global symbols

By default, all global symbols are written to the symdefs file.

When symdefs file exists, the linker uses its contents to restrict the output to a subset of the global symbols.

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

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

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

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

See also

- *Accessing symbols in another image* on page 4-12.

4.4.2 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.

———————————

See also

- *Accessing symbols in another image* on page 4-12.

4.4.3 Symdefs file format

The symdefs file contains symbols and their values.

The file consists of an identification line, optional comments, and symbol information as shown in Example 4-4 on page 4-14.

Example 4-4 Symdefs file format

```
#<SYMDEFS># ARM Linker, RVCTver [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
```

Identifying string

If the first 11 characters in the text file are `#<SYMDEFS>#`, the linker recognizes the file as a symdefs file.

The identifying string is followed by linker version information, and date and time of the most recent update of the symdefs file. The version and update information are not part of the identification 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 by the address, type, and name of the symbol on a single line:

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.

See also

- *Accessing symbols in another image* on page 4-12.

4.5 Hiding and renaming global symbols

This section describes how to use a steering file to manage symbol names in output files. For example, you can use steering files to protect intellectual property, or avoid namespace clashes. A steering file is a text file that contains a set of commands to edit the symbol tables of output objects.

Use the `armlink` command-line option `--edit` *`file-list`* to specify the steering file(s). When you are specifying more than one steering file, the syntax can be either of the following:

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

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

Do not include spaces between the comma and the filenames. See *--edit=file_list* on page 2-31 for more information.

See also:
- *Steering file format*.

4.5.1 Steering file format

A steering file is a plain text file of the following format:

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

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

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

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

For more information on steering file commands, see the following sections in the *Linker Reference Guide*:

- *IMPORT* on page 2-106
- *EXPORT* on page 2-103
- *RENAME* on page 2-108
- *RESOLVE* on page 2-111
- *REQUIRE* on page 2-110
- *HIDE* on page 2-105
- *SHOW* on page 2-113.

4.6 Using $Super$$ and $Sub$$ to override symbol definitions

There are situations where an existing symbol cannot be modified because, for example, it is located in an external library or in ROM code.

Use the `$Super$$` and `$Sub$$` patterns to patch an existing symbol.

For example, to patch the definition of a function `foo()`, use `$Super$$foo()` and `$Sub$$foo()` as follows:

`$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.

Example 4-5 shows the legacy function `foo()` modified to result in a call to `ExtraFunc()` and a call to `foo()`. See the *ELF for the ARM Architecture* for more information.

Example 4-5 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 */
}
```

4.7 Symbol versioning

Symbol versioning records extra information about symbols imported from, and exported by, a dynamic shared object. The dynamic loader uses this extra information to ensure that all the symbols required by an image are available at load time.

Symbol versioning enables shared object creators to produce new versions of symbols for use by all new clients, while maintaining compatibility with clients linked against old versions of the shared object.

See also:

- *Version*
- *Default version*
- *Creating versioned symbols*.

4.7.1 Version

Symbol versioning adds the concept of a *version* to the dynamic symbol table. A version is a name that symbols are associated with. When a dynamic loader tries to resolve a symbol reference associated with a version name, it can only match against a symbol definition with the same version name.

——— **Note** ———

A version might be associated with previous version names to show the revision history of the shared object.

———————————

See also

- *Symbol versioning*.

4.7.2 Default version

While a shared object might have multiple versions of the same symbol, a client of the shared object can only bind against the latest version.

This is called the *default version* of the symbol.

See also

- *Symbol versioning*.

4.7.3 Creating versioned symbols

By default, the linker does not create versioned symbols for a non-BPABI shared object.

Embedded symbols

You can add specially-named symbols to input objects that cause the linker to create symbol versions. These symbols are of the form:

- `name@version` for a non-default version of a symbol
- `name@@version` for a default version of a symbol.

You must define these symbols, at the address of the function or data, as that you want to export. The symbol name is divided into two parts, a symbol name *`name`* and a version definition *`version`*. The *`name`* is added to the dynamic symbol table and becomes part of the interface to the shared object. Version creates a version called *`ver`* if it does not already exist and associates *`name`* with the version called *`ver`*.

For more information on how to create version symbols, see:

- *Adding symbol versions* on page 2-29 in the *Compiler User Guide*
- Chapter 2 *Writing ARM Assembly Language* in the *Assembler Guide*.

Example 4-6 places the symbols `foo@ver1`, `foo@@ver2`, and `bar@@ver1` into the object symbol table:

Example 4-6 Creating versioned symbols, embedded symbols

```
int old_function(void) __asm__("foo@ver1");
int new_function(void) __asm__("foo@@ver2");
int other_function(void) __asm__("bar@@ver1");
```

The linker reads these symbols and creates version definitions `ver1` and `ver2`. The symbol `foo` is associated with a non-default version of `ver1`, and with a default version of `ver2`. The symbol `bar` is associated with a default version of `ver1`.

There is no way to create associations between versions with this method.

Steering file

You can embed the commands to produce symbol versions in a script file that is specified by the command-line option `--symver_script=`*`file`*. Using this option automatically enables symbol versioning.

The script file supports the same syntax as the GNU *ld* linker.

Using a script file enables you to associate a version with an earlier version.

A steering file can be provided in addition to the embedded symbol method. See *Steering file format* on page 4-16 for more information. If you choose to do this then your script file must match your embedded symbols and use the *Backus-Naur Form* (BNF) notation:

```
version_definition ::=

  version_name "{" symbol_association* "}" [depend_version] ";"
```

The *`version_name`* is a string containing the name of the version. *`depend_version`* is a string containing the name of a version that this *`version_name`* depends on. This version must have already been defined in the script file. Version names are not significant, but it helps to choose readable names, for example:

```
symbol_association ::=

  "local:" | "global:" | symbol_name ";"
```

where:

- `"local:"` indicates that all subsequent `symbol_names` in this version definition are local to the shared object and are not versioned.
- `"global:"` indicates that all subsequent `symbol_names` belong to this version definition.

  There is an implicit `"global:"` at the start of every version definition.
- `symbol_name` is the name of a global symbol in the static symbol table.

Example 4-7 shows a steering file that corresponds to the embedded symbols example (Example 4-6 on page 4-20) with the addition of dependency information so that `ver2` depends on `ver1`:

Example 4-7 Creating versioned symbols, steering file

```
ver1
{
    global:
        foo; bar;
    local:
        *;
};

ver2
{
    global:
        foo;
} ver1;
```

Errors & warnings

If you use a script file then the version definitions and symbols associated with them must match. The linker warns you if it detects any mismatch.

Filename

Use the command-line option `--symver_soname` to turn on implicit symbol versioning. Use this option if you need to version your symbols in order to force static binding, but where you do not care about the version number that they are given.

Where a symbol has no defined version, the linker uses the `SONAME` of the file being linked.

This option cannot be combined with embedded symbols or a script file.

See also

- *Symbol versioning* on page 4-19.

Chapter 5
Using Scatter-loading Description Files

This chapter describes how you use the ARM® linker, `armlink`, with scatter-loading description files to create complex images. It contains the following sections:

5.1 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 regions are scattered in the memory map at load and execution time.

An image is made up of regions and output sections. Every region in the image can have a different load and execution address. See *The image structure* on page 3-2 for more information.

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

- grouping information describing how input sections are grouped into output sections and regions
- placement information describing the addresses where regions are to be located in the memory maps.

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

See also:

- *When to use scatter-loading*
- *Symbols defined for scatter-loading* on page 5-3
- *Specifying stack and heap using the scatter-loading description file* on page 5-4
- *Scatter-loading command-line option* on page 5-5
- *Images with a simple memory map* on page 5-6
- *Images with a complex memory map* on page 5-8.

5.1.1 When to use scatter-loading

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

Situations where scatter-loading descriptions are either required or very useful are:

Complex memory maps

Code and data that must be placed into many distinct areas of memory require detailed instructions on which section goes into which memory space.

Different types of memory

Many systems contain a variety of physical memory devices such as flash, ROM, SDRAM, and fast SRAM. A scatter-loading description can match the code and data with the most appropriate type of memory. For example, interrupt code might be placed into fast SRAM to improve interrupt response time but infrequently-used configuration information might be placed into slower flash memory.

Memory-mapped peripherals

The scatter-loading description can place a data section at a precise address in the memory map so that memory mapped peripherals can be accessed.

Functions at a constant location

A function can be placed at the same location in memory even though the surrounding application has been modified and recompiled. This is useful for jump table implementation.

Using symbols to identify the heap and stack

Symbols can be defined for the heap and stack location when the application is linked.

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

See also

- *About scatter-loading* on page 5-2.

5.1.2 Symbols defined for scatter-loading

When the linker creates an image using a scatter-loading description file, it creates some region-related symbols. These are described in *Region-related symbols* on page 4-4. The linker creates these special symbols only if your code references them.

Undefined symbols

Be aware, the following symbols are undefined when a scatter-loading description file is used:

- `Image$$RW$$Base`
- `Image$$RW$$Limit`
- `Image$$RO$$Base`
- `Image$$RO$$Limit`
- `Image$$ZI$$Base`

- `Image$$ZI$$Limit`

See *Accessing linker-defined symbols* on page 4-4 for more information.

If you use a scatter-loading description file but do not specify any special region names for stack and heap, or do not re-implement `__user_setup_stackheap()`, the library generates an error message.

For more information see:

- *Tailoring the runtime memory model* on page 2-69 in the *Libraries and Floating Point Support Guide*
- *Placing the stack and heap* on page 3-13 in the *Developer Guide*.

See also

- *About scatter-loading* on page 5-2.

5.1.3 Specifying stack and heap using the scatter-loading description file

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

To select the two region memory model, define two special execution regions in your scatter-loading description 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:

```
ARM_LIB_HEAP 0x20100000 EMPTY 0x100000-0x8000  ; Heap starts at 1MB
                                               ; and grows upwards
ARM_LIB_STACK 0x20200000 EMPTY -0x8000         ; Stack space starts at the end
                                               ; of the 2MB of RAM
                                               ; And grows downwards for 32KB
```

Note

If you use the stack function above, you must also include an `IMPORT __use_two_region_memory` in your assembly source or a `#pragma import(__use_two_region_memory)` in your C/C++ source, because the two-region model is not automatically selected.

You can force `__user_setup_stackheap()` to use a combined stack/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.

See also

- *About scatter-loading* on page 5-2.

5.1.4 Scatter-loading command-line option

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

```
--scatter=description_file
```

This instructs the linker to construct the image memory map as described in *`description_file`*. The format of the description file is given in Chapter 3 *Formal syntax of the scatter-loading description file* in the *Linker Reference Guide*.

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

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

- `--bpabi`
- `--dll`
- `--partial`
- `--reloc`
- `--ro_base`
- `--ropi`
- `--rosplit`
- `--rw_base`

- `--rwpi`
- `--shared`
- `--split`
- `--startup`
- `--sysv`.

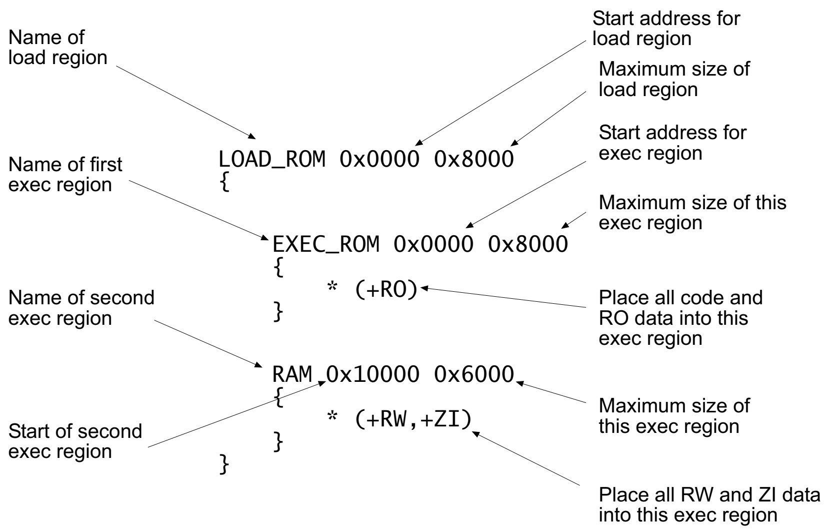
See also

- *Base Platform linking model* on page 2-7
- *Examples of specifying region and section addresses* on page 5-10
- *Equivalent scatter-loading descriptions for simple images* on page 5-48
- the following in the *Linker Reference Guide*:
  - *--base_platform* on page 2-14
  - *--bpabi* on page 2-17
  - *--dll* on page 2-30
  - *--partial* on page 2-64
  - *--reloc* on page 2-72
  - *--ro_base=address* on page 2-74
  - *--ropi* on page 2-75
  - *--rosplit* on page 2-75
  - *--rw_base=address* on page 2-76
  - *--rwpi* on page 2-77
  - *--shared* on page 2-80
  - *--startup=symbol, --no_startup* on page 2-83
  - *--sysv* on page 2-90

5.1.5 Images with a simple memory map

The scatter-loading description in Figure 5-1 on page 5-7 loads the segments from the object file into memory corresponding to the map shown in Figure 5-2 on page 5-7. The maximum size specifications for the regions are optional but, if they are included, these enable the linker to check that a region does not overflow its boundary.

In this example, the same result can be achieved by specifying `--ro-base 0x0` and `--rw-base 0x10000` as command-line options to the linker.



```
LOAD_ROM 0x0000 0x8000
{
    EXEC_ROM 0x0000 0x8000
    {
        * (+RO)
    }
    RAM 0x10000 0x6000
    {
        * (+RW,+ZI)
    }
}
```

Name of load region
Start address for load region
Maximum size of load region
Name of first exec region
Start address for exec region
Maximum size of this exec region
Name of second exec region
Place all code and RO data into this exec region
Maximum size of this exec region
Start of second exec region
Place all RW and ZI data into this exec region

Figure 5-1 Simple memory map in a scatter-loading description file

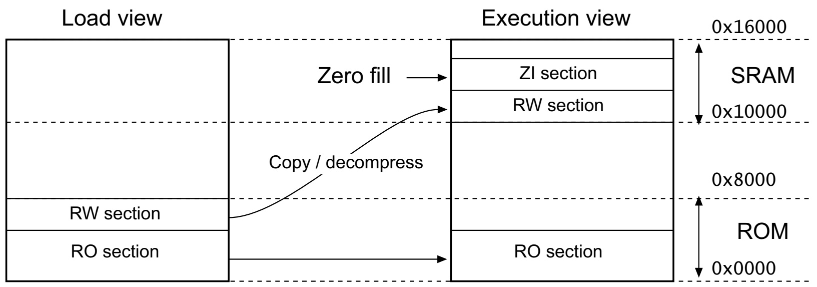


Figure 5-2 Simple scatter-loaded memory map

See also

- *About scatter-loading* on page 5-2.

5.1.6 Images with a complex memory map

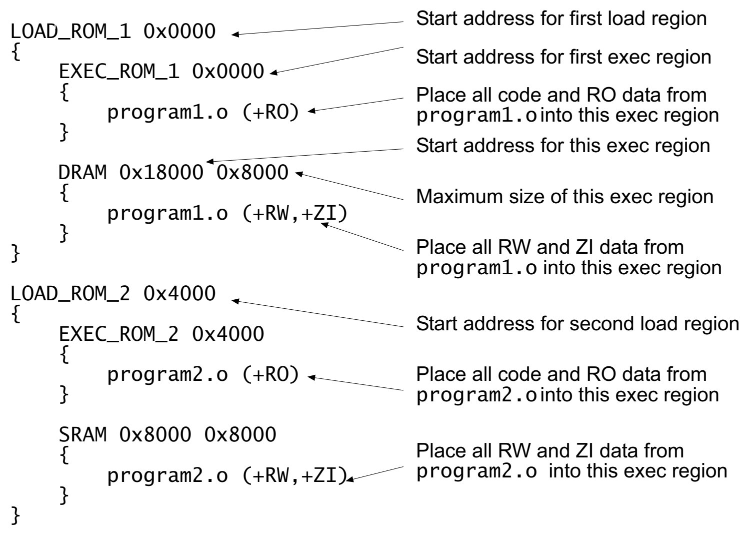
The scatter-loading description in Figure 5-3 loads the segments from the `program1.o` and `program2.o` files into memory corresponding to the map shown in Figure 5-4 on page 5-9.

Unlike the simple memory map shown in Figure 5-2 on page 5-7, this application cannot be specified to the linker using only the basic command-line options.

Caution

The scatter-loading description in Figure 5-3 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. See *Placing regions at fixed addresses* on page 5-14 for more information.



```
LOAD_ROM_1 0x0000
{
    EXEC_ROM_1 0x0000
    {
        program1.o (+RO)
    }
    DRAM 0x18000 0x8000
    {
        program1.o (+RW,+ZI)
    }
}
LOAD_ROM_2 0x4000
{
    EXEC_ROM_2 0x4000
    {
        program2.o (+RO)
    }
    SRAM 0x8000 0x8000
    {
        program2.o (+RW,+ZI)
    }
}
```

Start address for first load region
Start address for first exec region
Place all code and RO data from program1.o into this exec region
Start address for this exec region
Maximum size of this exec region
Place all RW and ZI data from program1.o into this exec region
Start address for second load region
Place all code and RO data from program2.o into this exec region
Place all RW and ZI data from program2.o into this exec region

Figure 5-3 Complex memory map in a scatter-loading description file

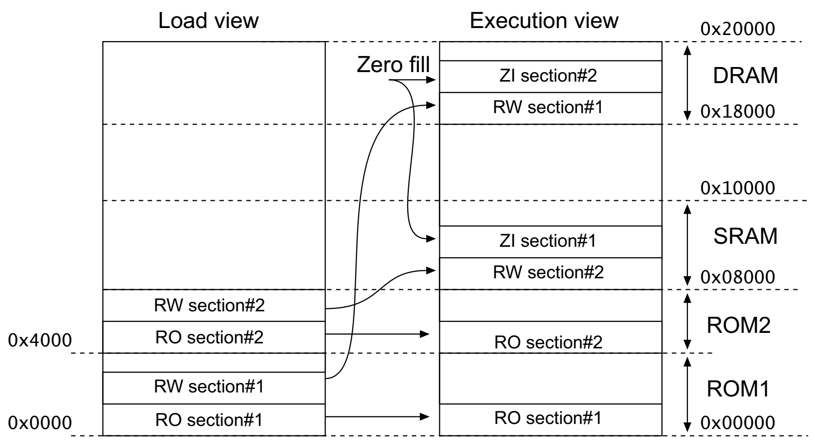


Figure 5-4 Complex scatter-loaded memory map

See also

- *About scatter-loading* on page 5-2
- *Scatter files containing relative base address load regions and a ZI execution region* on page 3-36 in the *Linker Reference Guide*.

5.2 Examples of specifying region and section addresses

This section describes input and executions sections, regions and preprocessing directives. For examples on accessing data and functions at fixed addresses, see Chapter 3 *Embedded Software Development* in the *Developer Guide*.

See also:

- *Selecting veneer input sections in scatter-loading descriptions*
- *Creating root execution regions* on page 5-11
- *Placing unassigned sections with the .ANY module selector* on page 5-26
- *Examples of using placement algorithms for .ANY sections* on page 5-30
- *Example of next_fit algorithm showing behavior of full regions, selectors, and priority* on page 5-33
- *Examples of using sorting algorithms for .ANY sections* on page 5-35
- *Using overlays to place sections* on page 5-38
- *Assigning sections to a root region* on page 5-40
- *Reserving an empty region* on page 5-41
- *Placing ARM C library code* on page 5-43
- *Creating regions on page boundaries* on page 5-45
- *Using preprocessing directives* on page 5-46.

5.2.1 Selecting veneer input sections in scatter-loading descriptions

Veneers are used to switch between ARM and Thumb code or to perform a longer program jump than can be specified in a single instruction. See *Veneers* on page 3-23. Use a scatter-loading description file to place linker-generated veneer input sections. At most, one execution region in the scatter-loading description 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$$Code)` in scatter-loading description files from earlier versions of ARM tools are automatically translated into `*(Veneer$$Code)`. Use `*(Veneer$$Code)` in new descriptions.

`*(Veneer$$Code)` is ignored when the amount of code in an execution region exceeds 4Mb of Thumb code, 16Mb of Thumb-2 code, and 32Mb of ARM code.

See also

- *Examples of specifying region and section addresses* on page 5-10.

5.2.2 Creating root execution regions

A root region is a region with the same load and execution address. The initial entry point for an 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.

To specify that a region is a root region in a scatter-loading description file you can either:

- 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.

  See Example 5-1.

Example 5-1 Specifying 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 description
}
```

- Use the `FIXED` execution region attribute to ensure that the load address and execution address of a specific region are the same. See Example 5-2 and Figure 5-5.

  You can use the `FIXED` attribute to place any execution region at a specific address in ROM. See *Placing regions at fixed addresses* on page 5-14 for more information.

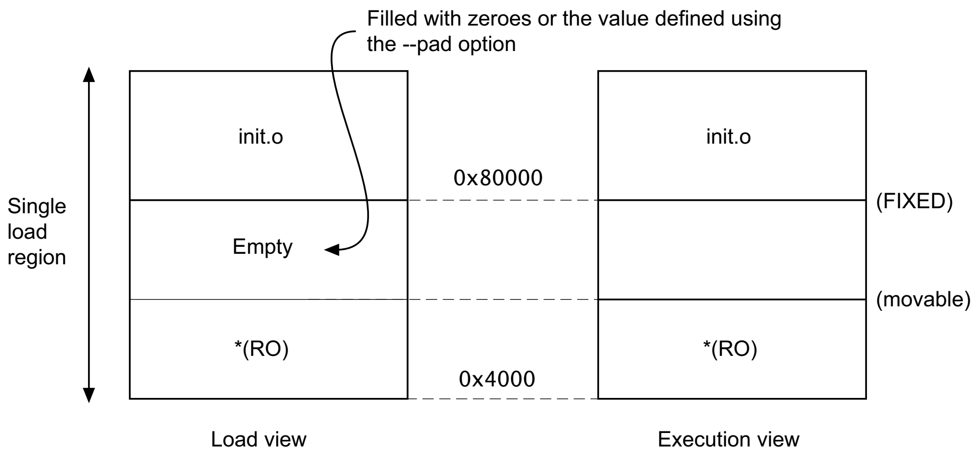


Figure 5-5 Memory Map for fixed execution regions

Example 5-2 Using the FIXED attribute

```
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 description
}
```

Examples of misusing the FIXED attribute

Example 5-3 shows common cases where the FIXED execution region attribute is misused:

Example 5-3 Misuse of the FIXED attribute

```
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
}
```

Placing regions at fixed addresses

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

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

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

To make your code easier to maintain and debug, use the minimum amount of placement specifications in scatter-loading description files and leave the detailed placement of functions and data to the linker.

You cannot specify component objects that have been partially linked. For example, if you partially link the objects obj1.o, obj2.o, and obj3.o together to produce obj_all.o, the resulting 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 only refer to the combined object obj_all.o.

Note

There are some situations where using FIXED and a single load region are not appropriate. Other techniques for specifying fixed locations are:

- If your loader can handle multiple load regions, place the RO code or data in its own load region.
- If you do not require the function or data to be at a fixed location in ROM, use ABSOLUTE instead of FIXED. The loader then copies the data from the load region to the specified address in RAM. ABSOLUTE is the default attribute.
- To place a data structure at the location of memory-mapped I/O, use two load regions and specify UNINIT. UNINIT ensures that the memory locations are not initialized to zero.

Placing functions and data at specific addresses

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

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

- An execution region can be created at the desired address with a section description that selects just 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. See *Using __at sections to place sections at a specific address* on page 5-22 for more information.

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. See *__attribute__((at(address)))* on page 4-48 in the *Compiler Reference Guide*.
- Use `__attribute__((section(”name”)))` to create multiple named sections. See *__attribute__((section("name")))* on page 4-52 in the *Compiler Reference Guide*.
- Use the `AREA` directive from assembly language. In assembly code, the smallest locatable unit is an `AREA`. See the *Assembler Guide* for more information.
- Use the `--split_sections` compiler option to generate one ELF section for each function in the source file. See *--split_sections* on page 2-118 in the *Compiler Reference Guide*.

  This option increases code size slightly 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 placing a variable at a specific address without scatter-loading

The following example shows how to place code and data at specific addresses, and does not require a scatter-loading 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

void 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 `0x20000`. `gSquared` is placed in the execution region `ER$$.ARM.__AT_0x00005000` and load region `LR$$.ARM.__AT_0x00005000`.

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           15    .ARM.__AT_0x00005000  main.o
```

Example of placing a variable in a named section with scatter-loading

This example shows how to modify your source code to place code and data in a specific section using a scatter-loading 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-loading 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
    }
    RAM 0x200000 (0x1FF00-0x2000)   ; RW & ZI data to be placed at
0x200000
    {
        *(+RW, +ZI)
    }
    ARM_LIB_STACK 0x800000 EMPTY -0x10000
    {
    }
    ARM_LIB_HEAP  +0 EMPTY 0x10000
    {
    }
}
```

The `ARM_LIB_STACK` and `ARM_LIB_HEAP` regions are required because the program is being linked with the semihosting libraries.

4. Compile and link the sources:

   ```
   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-loading file specifies that the section `foo` is to be placed in the `ER3` execution region.

The memory map shows:

```
  Load Region LR1 (Base: 0x00000000, Size: 0x00001778, Max: 0x00020000, ABSOLUTE)
...
    Execution Region ER3 (Base: 0x00010000, Size: 0x00000004, Max: 0x00002000, ABSOLUTE)

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x00010000   0x00000004   Data   RW           15    foo                 main.o
...
```

Note

If you omit `*(foo)` from the scatter-loading file, the section is placed in the region of the same type. That is `RAM` in this example.

Example of placing a variable at a specific address with scatter-loading

This example shows how to modify your source code to place code and data at a specific address using a scatter-loading 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-loading 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
       }
       RAM 0x200000 (0x1FF00-0x2000)   ; RW & ZI data to be placed at
   0x200000
       {
           *(+RW, +ZI)
       }
       ARM_LIB_STACK 0x800000 EMPTY -0x10000
       {
       }
       ARM_LIB_HEAP  +0 EMPTY 0x10000
       {
       }
   }
   ```

   The `ARM_LIB_STACK` and `ARM_LIB_HEAP` regions are required because the program is being linked with the semihosting libraries.

4. Compile and link the sources:

   ```
   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 `0x11000`:

```
...
    Execution Region ER2 (Base: 0x00001598, Size: 0x0000ea6c, Max: 0xffffffff, ABSOLUTE)

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object
```

```
    0x00001598   0x0000000c   Code   RO           3    .text              function.o
    0x000015a4   0x0000ea5c   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$$.ARM.__AT_0x00010000 that contains the execution region ER$$.ARM.__AT_0x00020000.

Placing a named section explicitly using scatter-loading

The scatter-loading description file in Example 5-4 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.

Example 5-4 Section placement

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

```
        data.o(+RO-DATA)            ; place RO data between 0x1FF00 and 0x1FFFF
    }
}
```

Using __attribute__((section("name")))

Placing a code or data object in its own source file and then placing the object file sections uses standard coding techniques. However, you can also use `__attribute__((section("name")))` and a scatter-loading description file to place named sections. Create a module, for example, `adder.c` and name a section explicitly as shown in Example 5-5.

Example 5-5 Naming a section

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

Use a scatter-loading description file to specify where the named section is placed, see Example 5-6. If both code and data sections have the same name, the code section is placed first.

Example 5-6 Placing a section

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

    ADDER 0x08000000
    {
        adder.o (foo)                  ; select section foo from adder.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-loading 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;
}
```

Using __at sections to place sections at a specific address

A section can be given a special name that encodes the address at which it must be placed. You can specify the name as follows:

`.ARM.__at_address`

Where:

address	is the required address of the section. This can be specified in hexadecimal or decimal. Sections in the form of `.ARM.__at_address` are referred to by the abbreviation `__at`.

In the compiler, variables can be assigned to `__at` sections by either explicitly naming the section using the `__attribute__((section("name")))` or by using the attribute `__at` which sets up the name of the section for you. See Example 5-7.

Note

When using `__attribute__((at(address)))`, the part of the `__at` section name representing *address* is normalized to an 8 digit hexadecimal number. The name of the section is only significant if you are trying to match the section by name in a scatter-loading description file. The linker automatically assigns `__at` sections when you use the `--autoat` command-line option. This option is the default.

Example 5-7 Assigning variables to `__at` sections

```
; 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;
```

See *__attribute__((at(address)))* on page 4-48 and *__attribute__((section("name")))* on page 4-52 in the *Compiler Reference Guide* for more information.

Restrictions

- `__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 and BPABI executables and BPABI DLLs
- `__at` sections must have an address that is a multiple of their alignment
- `__at` sections ignore any `+FIRST` or `+LAST` ordering constraints.

Automatic placement

The automatic placement of `__at` sections is enabled by default. This feature is controlled by the linker command-line option, `--autoat`. See *--autoat, --no_autoat* on page 2-14 in the *Linker Reference Guide* for more information.

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 limit is assumed to be infinite.
- 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.

Example 5-8 show the sections `.ARM.__at_0x0` type RO, `.ARM.__at_0x2000` type RW, `.ARM.__at_0x4000` type ZI and `.ARM.__at_0x8000` type ZI.

Example 5-8 Automatic placement of `__at` sections

```
LR1 0x0
{
    ER_RO 0x0 0x2000
    {
        *(+RO)      ; .ARM.__at_0x0 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.
```

Manual placement

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.

Example 5-9 on page 5-25 shows the placement of read-only sections `.ARM.__at_0x2000` and the read-write section `.ARM.__at_0x4000`. Load and execution regions are not created automatically in manual mode. An error is produced if an `__at` section cannot be placed in an execution region.

Example 5-9 Manual placement of `__at` sections

```
LR1 0x0
{
    ER_RO 0x0 0x2000
    {
        *(+RO)              ; .ARM.__at_0x0 is selected by +RO
    }
    ER_RO2 0x2000
    {
        *(.ARM.__at_0x2000)  ; .ARM.__at_0x2000 is selected by .ARM.__at_0x2000
    }
    ER2 0x4000
    {
        *(+RW +ZI)           ; .ARM.__at_0x4000 is selected by +RW
    }
}
```

Placing a key in flash memory

Some flash devices require a key to be written to an address to activate certain features. An `__at` section provides a simple method of writing a value to a specific address.

Assuming a device has flash memory from `0x8000` to `0x10000` and a key is required in address `0x8000`. To do this with an `__at` section, you must declare a variable so that the compiler can generate a section called `.ARM.__at_0x8000`. See Example 5-7 on page 5-22.

Example 5-10 shows a scatter-loading description file with manual placement of the flash execution region.

Example 5-10 Manual placement of flash execution regions

```
ER_FLASH 0x8000 0x2000
{
    *(+RO)                  ; other code, read-only data, and padding if reqd
    *(.ARM.__at_0x8000)     ; key
}
```

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

Example 5-11 on page 5-26 shows a scatter-loading description file with automatic placement of the flash execution region. Use the linker command-line option `--autoat` to enable automatic placement.

Example 5-11 Automatic placement of flash execution regions

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

Mapping a structure over a peripheral register

To place an uninitialized variable over a peripheral register, a ZI `__at` section can be used. 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));
```

Example 5-12 shows the a scatter-loading description file with the manual placement of the ZI `__at` section.

Example 5-12 Manual placement of ZI `__at` sections

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

Using automatic placement, 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.

See also

- *Examples of specifying region and section addresses* on page 5-10.

5.2.3 Placing 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 the placement of those sections is not important, you can use the `.ANY` module selector in the scatter-loading 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.

Default rules for placing unassigned sections

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

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

Placement rules when using multiple `.ANY` selectors

If more than one `.ANY` selector is present in a scatter-loading file, the linker takes the unassigned section with the largest size and assigns the section to the most specific `.ANY` execution region that has enough free space. For example, `.ANY(.text)` is judged to be more specific than `.ANY(+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.

Prioritizing `.ANY` sections

You can give a priority ordering if you have multiple `.ANY` sections with the `.ANY`*`num`* selector, where *`num`* is a positive integer from zero upwards. The highest priority is given to the selector with the highest integer.

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

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

Controlling the placement of input sections for multiple `.ANY` selectors

You can modify how the linker places unassigned input sections when using multiple `.ANY` selectors by using a different placement algorithm or a different sort order. The following command-line options are available:

- `--any_placement=`*`algorithm`*, where *`algorithm`* is one of `first_fit`, `worst_fit`, `best_fit`, or `next_fit`
- `--any_sort_order=`*`order`*, where *`order`* is one of `cmdline` or `descending_size`.

Use `first_fit` when you want to fill regions in order.

Use `best_fit` when you want to fill regions to their maximum.

Use `worst_fit` when you want to fill regions evenly. With equal sized regions and sections `worst_fit` fills regions cyclically.

Use `next_fit` when you need a more deterministic fill pattern.

If the linker attempts to fill a region to its limit, as it does with `first_fit` and `best_fit`, it might 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.

Specifying the maximum size permitted for placing unassigned sections

The execution region attribute `ANY_SIZE` *`max_size`* enables you to specify the maximum size in that region that `armlink` can fill with unassigned sections.

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`.

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. 98% of the region is filled, with a two percent contingency for veneers. This is the same as omitting the `ANY_SIZE` attribute and parameter.

See also

- *Examples of using placement algorithms for .ANY sections*
- *Example of next_fit algorithm showing behavior of full regions, selectors, and priority* on page 5-33
- *Examples of using sorting algorithms for .ANY sections* on page 5-35.
- The following in the *Linker Reference Guide*:
  - *--any_contingency* on page 2-8
  - *--any_placement=algorithm* on page 2-8
  - *--any_sort_order=order* on page 2-10
  - *--info=topic[,topic,...]* on page 2-44
  - *--map, --no_map* on page 2-59
  - *--section_index_display=type* on page 2-79
  - *--tiebreaker=option* on page 2-92
  - *Syntax of an execution region description* on page 3-8
  - *Syntax of an input section description* on page 3-18
  - *Resolving multiple matches* on page 3-24
  - *Behavior when .ANY sections overflow because of linker-generated content* on page 3-27.

5.2.4 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 5-1 Input section properties

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

The scatter-loading 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 specificness, 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.

See also

- *Placing unassigned sections with the .ANY module selector* on page 5-26
- *Example of next_fit algorithm showing behavior of full regions, selectors, and priority*
- the following in the *Linker Reference Guide*:
  - *--any_placement=algorithm* on page 2-8
  - *--scatter=file* on page 2-78.

5.2.5 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 5-2 Input section properties

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

The scatter-loading 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 specificness 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.

See also

- *Placing unassigned sections with the .ANY module selector* on page 5-26
- *Examples of using placement algorithms for .ANY sections* on page 5-30
- the following in the *Linker Reference Guide*:
  - *--any_placement=algorithm* on page 2-8
  - *--scatter=file* on page 2-78.
  - *Resolving multiple matches* on page 3-24.

5.2.6 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 5-3 Input section properties for `sections_a.o`

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

Table 5-4 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 5-5 Sort order for `descending_size` algorithm

Name	Size
`seca_4`	`0x14`
`secb_4`	`0x14`
`seca_3`	`0x10`
`secb_3`	`0x10`

Table 5-5 Sort order for `descending_size` algorithm (continued)

Name	Siz e
`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 5-6 Sort order for `cmdline` algorithm

Name	Siz e
`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`.

See also

- *Placing unassigned sections with the .ANY module selector* on page 5-26

- the following in the *Linker Reference Guide*:
  - — *--any_sort_order=order* on page 2-10
  - — *--scatter=file* on page 2-78
  - — *--tiebreaker=option* on page 2-92.

5.2.7 Using overlays to place sections

You can use the `OVERLAY` attribute in a scatter-loading description file to place multiple execution regions at the same address. An overlay manager is required to make sure that only one execution region is instantiated at a time. ARM RealView Compilation Tools does not provide an overlay manager.

Example 5-13 defines a static section in RAM followed by a series of overlays. Here, only one of these sections is instantiated at a time.

Example 5-13 Specifying a root region

```
EMB_APP 0x8000
{
    .
    .
    STATIC_RAM 0x0                  ; contains most of the RW and ZI code/data
    {
            * (+RW,+ZI)
    }
    OVERLAY_A_RAM 0x1000 OVERLAY    ; start address of overlay...
    {
            module1.o (+RW,+ZI)
    }
    OVERLAY_B_RAM 0x1000 OVERLAY
    {
            module2.o (+RW,+ZI)
    }
    ...                             ; rest of scatter 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 the overlay manager. Therefore, your overlay manager must copy any Code and Data, and initialize any ZI when it instantiates a region. If the region contains initialized data you also need to prevent RW data compression using `NOCOMPRESS`.

The linker defined symbols can be used to obtain the addresses required to copy the code and data, see *Accessing linker-defined symbols* on page 4-4 for more information.

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 will depend on the regions that precede it and the value of +offset, with +0 having a special meaning.

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*

Table 5-7 shows the effect of +offset when used with the OVERLAY attribute. REGION1 appears immediately before REGION2 in the scatter-loading description file.

Table 5-7 Using relative offset in overlays

REGION1 is set with OVERLAY	***+offset***	**REGION2 Base Address**
NO	*<offset>*	REGION1 Limit + *<offset>*
YES	+0	REGION1 Base Address
YES	*<non-0 offset>*	REGION1 Limit + *<non-0 offset>*

Example 5-14 shows the use of relative offsets with overlays and the effect on execution region addresses.

Example 5-14 Example of relative offset in overlays

```
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, a zero relative offset can be used 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 allow an overlay-aware debugger to track which overlay is currently active.

See also

- *Examples of specifying region and section addresses* on page 5-10.

5.2.8 Assigning sections to a root region

There are a number of ARM library sections that 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`.

Use a scatter-loading description file to specify a root section in the same way as a named section. Example 5-15 on page 5-41 uses the section selector `InRoot$$Sections` to place all sections that must be in a root region.

Example 5-15 Specifying a root region

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

See also

- *Examples of specifying region and section addresses* on page 5-10.

5.2.9 Reserving an empty region

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

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

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

If the length is given as a negative value, the address is taken to be the end address of the region. This must be an absolute address and not a relative one. For example, the execution region definition `STACK 0x800000 EMPTY -0x10000` shown in Example 5-16 defines a region called `STACK` that starts at address `0x7F0000` and ends at address `0x800000`.

Example 5-16 Reserving a region for the stack

```
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 used to place stack
    }
    HEAP +0 EMPTY 0x10000                 ; region starts at the end of previous
                                          ; region. End of region calculated using
                                          ; positive length
    {
                                          ; Empty region used to place heap
    }
    ...                                   ; rest of scatter 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.

Figure 5-6 is a diagrammatic representation of this example.

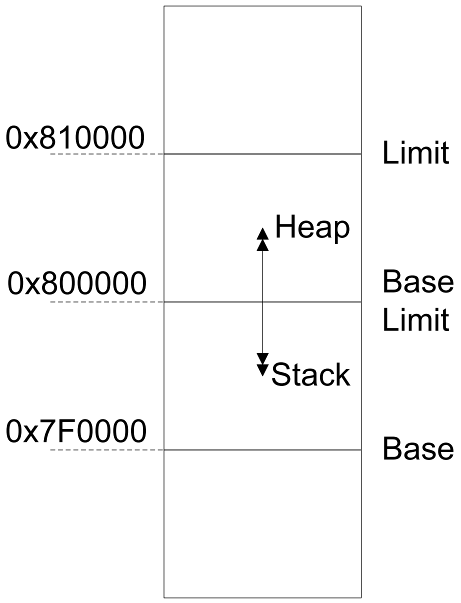


Figure 5-6 Reserving a region for the stack

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.

See also

- *Examples of specifying region and section addresses* on page 5-10.

5.2.10 Placing ARM C library code

You can place code from the ARM standard C and C++ libraries in a scatter-loading description file. Use `*armlib` or `*armlib*` so that the linker can resolve library naming in your scatter-loading file. For example:

```
ER 0x2000
{
    *armlib/c_* (+RO)                   ; all ARM-supplied C libraries
    ...                                 ; rest of scatter description...
}
```

Example 5-17 shows how to place library code.

Example 5-17 Placing ARM 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
                                            ; e.g. floating-point libraries
}
RAM2 0x4000
{
    * (+RW, +ZI)
}
```

The name `armlib` is used to indicate the ARM C library files that are located in the `armlib` directory. The location of `armlib` is defined by the `RVCT40LIB` environment variable.

See also

- *Examples of specifying region and section addresses* on page 5-10
- *Assigning sections to a root region* on page 5-40
- *Placing ARM C++ library code*.

5.2.11 Placing ARM C++ library code

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

```
#include <iostream>

using namespace std;

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

To place the C++ library code, define the scatter-loading 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 `armlib` is used to indicate the ARM C library files that are located in the `armlib` directory.

The name `cpplib` is used to indicate the ARM C++ library files that are located in the `cpplib` directory.

The location of `armlib` and `cpplib` is defined by the `RVCT40LIB` environment variable.

See also

- *Examples of specifying region and section addresses* on page 5-10
- *Assigning sections to a root region* on page 5-40
- *Placing ARM C library code* on page 5-43.

5.2.12 Creating regions on page boundaries

The `ALIGN` directive produces an ELF file that can be loaded directly to a target with each execution region starting at a page boundary.

Note

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

Example 5-18 on page 5-46 assumes a page size of 65536 and produces an ELF file with each region starting on a new page.

Example 5-18 Creating regions on page boundaries

```
LR1 +4 ALIGN 65536              ; load region at 65536
{
    ER1 +0 ALIGN 65536          ; first region at first page boundary
    {
        *(+RO)                  ; all RO sections are placed consecutively here
    }
    ER2 +0 ALIGN 65536          ; second region at next available page boundary
    {
        *(+RW)                  ; all RW sections are placed consecutively here
    }
    ER3 +0 ALIGN 65536          ; third region at next available page boundary
   {
        *(+ZI)                  ; all ZI sections are placed consecutively here
    }
    ...                         ; rest of scatter description...
}
```

See also

- *Examples of specifying region and section addresses* on page 5-10.

5.2.13 Using preprocessing directives

You can pass a scatter-loading file through a C preprocessor. This allows access to all the features of the C preprocessor.

Use the first line in the scatter-loading description file to specify a preprocessor that the linker invokes to process the file. This command is of the form:

```
#! <preprocessor> [pre_processor_flags]
```

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

You can:

- add preprocessing directives to the top of the scatter-loading description file
- use simple expression evaluation in the scatter-loading file.

For example, a scatter-loading file, `file.scat`, might contain:

```
#! armcc -E

#define ADDRESS 0x20000000
#include "include_file_1.h"
```

```
lr1 ADDRESS
{
    ...
}
```

The linker parses the preprocessed scatter-loading description file and treats the directives as comments.

You can also use preprocessing of a scatter-loading 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
   ```

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

For example, 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-loading description 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-loading file to produce the load region:

```
LOAD_FLASH 0x8808 ; start address
```

See *--predefine="string"* on page 2-66 in the *Linker Reference Guide* for more information.

See also

- *Examples of specifying region and section addresses* on page 5-10.

5.3 Equivalent scatter-loading descriptions for simple images

The command-line options `--reloc`, `--ro-base`, `--rw-base`, `--ropi`, `--rwpi`, and `--split` create the simple image types described in *Using command-line options to create simple images* on page 3-30. 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.

See also:

- *Type 1, one load region and contiguous execution regions*
- *Type 2, one load region and non-contiguous execution regions* on page 5-50
- *Type 3, two load regions and non-contiguous execution regions* on page 5-52.

5.3.1 Type 1, one load region and contiguous execution regions

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

`--ro-base` *`address`* specifies the load and execution address of the region containing the RO output section. Example 5-19 shows the scatter-loading description equivalent to using `--ro-base 0x040000`.

Example 5-19 Single load region and contiguous execution regions

```
LR_1 0x040000     ; Define the load region name as LR_1, the region starts at 0x040000.
{
    ER_RO +0      ; First execution region is called ER_RO, region starts at end of previous region.
                  ; However, since there is no previous region, the address is 0x040000.
    {
        * (+RO)   ; All RO sections go into this region, they are placed consecutively.
    }
    ER_RW +0      ; Second execution region is called ER_RW, the region starts at the end of the
                  ; previous region. The address is 0x040000 + size of ER_RO region.
    {
        * (+RW)   ; All RW sections go into this region, they are placed consecutively.
    }
    ER_ZI +0      ; Last execution region is called ER_ZI, the region starts at the end of the
                  ; previous region at 0x040000 + the size of the ER_RO regions + the size of
                  ; the ER_RW regions.
    {
        * (+ZI)   ; All ZI sections are placed consecutively here.
    }
}
```

The description shown in Example 5-19 on page 5-48 creates an image with one load region called LR_1, whose load address is 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.

The --reloc option is used 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.

Example 5-20 shows the scatter-loading description equivalent to using --ro-base 0x010000 --ropi.

Example 5-20 Position-independent code

```
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.
    }
}
```

Shown in Example 5-20 on page 5-49, `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.

See also

- *Equivalent scatter-loading descriptions for simple images* on page 5-48.

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

An image of this type consists of a single load region in the load view and three execution regions in the execution view. It is similar to images of type 1 except that the RW execution region is not contiguous with the RO execution region.

`--ro-base=address1` specifies the load and execution address of the region containing the RO output section. `--rw-base=address2` specifies the execution address for the RW execution region.

Example 5-21 shows the scatter-loading description equivalent to using `--ro-base=0x010000 --rw-base=0x040000`.

Example 5-21 Single load region and multiple execution regions

```
LR_1 0x010000       ; Defines the load region name as LR_1
{
    ER_RO +0        ; The first execution region is called ER_RO and starts at end of previous region.
                    ; Since 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.
    }
}
```

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.

Example 5-22 shows the scatter-loading description equivalent to using --ro-base=0x010000 --rw-base=0x018000 --rwpi.

Example 5-22 Position-independent data

```
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.

See also

- *Equivalent scatter-loading descriptions for simple images* on page 5-48.

5.3.3 Type 3, two load regions and non-contiguous execution regions

Type 3 images consist of two load regions in load view and three execution regions in execution view. They are similar to images of type 2 except that the single load region in type 2 is now split into two load regions.

Relocate and split load regions using the following linker options:

`--reloc`
: The combination `--reloc --split` makes an image similar to simple type 3, but the two load regions now have the `RELOC` attribute.

`--ro-base=`*`address1`*
: Specifies the load and execution address of the region containing the RO output section.

`--rw-base=`*`address2`*
: Specifies the load and execution address for the region containing the RW output section.

`--split`
: Splits the default single load region (that contains the RO and RW output sections) into two load regions. One load region contains the RO output section and one contains the RW output section.

Example 5-23 shows the scatter-loading description equivalent to using `--ro-base=0x010000 --rw-base=0x040000 --split`.

Example 5-23 Multiple load regions

```
LR_1 0x010000     ; The first load region is at 0x010000.
{
    ER_RO +0      ; The address is 0x010000.
    {
        * (+RO)
    }
}
LR_2 0x040000     ; The second load region is at 0x040000.
{
    ER_RW +0      ; The address is 0x040000.
    {
        * (+RW)   ; All RW sections are placed consecutively into this region.
    }
    ER_ZI +0      ; The address is 0x040000 + size of ER_RW region.
    {
        * (+ZI)   ; All ZI sections are placed consecutively into this region.
    }
}
```

In this example:

- The scatter-loading 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`. Its execution address is `0x040000`.
- The `ER_ZI` execution region is placed immediately following the end of the preceding execution region, `ER_RW`.

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` is used to specify that the two load regions now have the `RELOC` attribute.

Example 5-24 shows the scatter-loading description equivalent to using `--ro-base 0x010000 --rw-base 0x040000 --reloc --split`.

Example 5-24 Relocatable load regions

```
LR_1 0x010000 RELOC
{
     ER_RO + 0
     {
         * (+RO)
     }
}

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

```
    ER_ZI +0
    {
        * (+ZI)
    }
}
```

See also

- *Equivalent scatter-loading descriptions for simple images* on page 5-48.