RealView® Compilation Tools

Version 4.0

Linker Reference Guide

ARM®

ARM DUI 0381B (ID101213)

RealView Compilation Tools
Linker Reference Guide

Release Information

The following changes have been made to this book.

Change History

Date	Issue	Confidentiality	Change
September 2008	A	Non-Confidential	Release 4.0 for ARM® RealView® Development Suite
23 January 2009	A	Non-Confidential	Update 1 for RealView Development Suite v4.0
10 December 2010	B	Non-Confidential	Update 2 for RealView Development Suite v4.0

Proprietary Notice

Confidentiality Status

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 Reference Guide

Preface

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

About this book

This book provides reference information for the ARM® linker, `armlink` provided with ARM RealView® Compilation Tools.

It gives detailed desriptions on command-line options, steering file commands, scatter-loading description files, the *Base Platform Application Binary Interface* (BPABI) and *System V release 4* (SysV) shared libraries and executables. For general information on using and controlling the tools, see the *Linker User 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.

Chapter 2 ***Linker Command-line Options***

Read this chapter for a list of all the command-line options and steering file commands supported by the linker.

Chapter 3 ***Formal syntax of the scatter-loading description file***

Read this chapter for information on the format of scatter-loading description files.

Chapter 4 ***BPABI and SysV Shared Libraries and Executables***

Read this chapter for information on the BPABI and SysV shared libraries and executables.

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.

`mono`space 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 reference information that is specific to 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 Linker User Guide* (ARM DUI 0206)

- *RealView Compilation Tools Utilities Guide* (ARM DUI 0382)
- *RealView Compilation Tools Libraries and Floating Point Support Guide* (ARM DUI 0349)
- *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.

Other publications

For more information on the Linux Standard Base specification, see `http://www.linux-foundation.org`.

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 ARM RealView® Compilation Tools. It contains the following sections:

- *The ARM linker* on page 1-2
- *Compatibility with legacy objects and libraries* on page 1-3.

1.1 The ARM linker

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

- an ARM *Executable and Linking Format* (ELF) image
- a partially linked ELF object that can be used as input in a subsequent link step
- a shared object, compatible with the *Base Platform Application Binary Interface for the ARM Architecture* (BPABI) or *System V release 4* (SysV) specification, or a BPABI or SysV executable 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.

Chapter 2
Linker Command-line Options

This chapter lists the command-line options and steering file commands supported by the ARM linker, `armlink`. It includes the following section:

- *Command-line options listed alphabetically* on page 2-2
- *Steering file commands in alphabetical order* on page 2-102.

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.

2.1 Command-line options listed alphabetically

This section lists the command-line options supported by the linker in alphabetical order:

2.1.1 --add_needed, --no_add_needed

This option controls shared object dependencies of libraries that are not specified on the command-line.

Usage

The `--add_needed` setting applies to any following shared objects until a `--no_add_needed` option appears on the command line. The linker adds all shared objects that the shared object depends on and recursively all of the dependent shared objects to the link.

Default

If you are using the `--arm_linux` option then the default is `--add_needed` otherwise the default is `--no_add_needed`.

Example

Example 2-1 on page 2-7 shows how to use this option.

Example 2-1 `--[no_]add_needed` **options**

Assuming that the following dependencies exist:

- `cl1.so` depends on `dep1.so`
- `cl2.so` depends on `dep2.so`
- `cl3.so` depends on `dep3.so`
- `dep2.so` depends on `depofdep2.so`.

Use the following command-line options:

```
armlink --no_add_needed cl1.so --add_needed cl2.so --no_add_needed cl3.so
```

This results in the addition of the following shared objects to the link:

- `cl1.so`
- `cl2.so`
- `dep2.so`
- `depofdep2.so`
- `cl3.so`.

See also

- *--arm_linux* on page 2-11.

2.1.2 `--add_shared_references`, `--no_add_shared_references`

This option affects the behavior of the `--sysv` mode. If you specify `--add_shared_references` when linking an application the linker adds references from shared libraries. The linker gives an undefined symbol error message if these references are not defined by the application or by some other shared library. These references can be satisfied by static archive format libraries.

Note

A reference from a shared library can only be satisfied by a symbol definition with protected or default visibility, because these are the only symbols that can be exported into dynamic symbol tables. The linker gives an error message if the symbol reference is resolved by a symbol with hidden or internal visibility.

Default

The default option is `--no_add_shared_references`.

However, if you specify `--arm_linux`, the default option is `--add_shared_references`.

See also

- *--arm_linux* on page 2-11
- *--sysv* on page 2-90.

2.1.3 `--any_contingency`

This option allows extra space in any execution regions containing `.ANY` sections for linker-generated content such as veneers and alignment padding. Two percent of the space is reserved for veneers.

When a region is about to overflow because of potential padding, `armlink` lowers the priority of the `.ANY` selector.

This option is off by default. That is, `armlink` does not attempt to calculate padding and strictly follows the `.ANY` priorities.

Use this option with the `--scatter` option.

See also

- *--any_placement=algorithm*
- *--any_sort_order=order* on page 2-10
- *--info=topic[,topic,...]* on page 2-44
- *--scatter=file* on page 2-78
- *Syntax of an input section description* on page 3-18
- *Behavior when .ANY sections overflow because of linker-generated content* on page 3-27
- the following in the *Linker User Guide*:
  - *Placing unassigned sections with the .ANY module selector* on page 5-26.

2.1.4 `--any_placement=`*`algorithm`*

Controls the placement of sections that are placed using the `.ANY` module selector.

Syntax

`--any_placement=`*`algorithm`*

where *`algorithm`* is one of the following:

`best_fit` Place the section in the execution region that currently has the least free space but is also sufficient to contain the section.

`first_fit` Place the section in the first execution region that has sufficient space. The execution regions are examined in the order they are defined in the scatter file.

`next_fit` Place the section using the following rules:

- place in the current execution region if there is sufficient free space
- place in the next execution region only if there is insufficient space in the current region
- never place a section in a previous execution region.

`worst_fit` Place the section in the execution region that currently has the most free space.

Use this option with the `--scatter` option.

Usage

The placement algorithms interact with scatter-loading files and `--any_contingency` as follows:

Interaction with normal scatter-loading rules

Scatter-loading with or without `.ANY` assigns a section to the most specific selector. All algorithms continue to assign to the most specific selector in preference to `.ANY` priority or size considerations.

Interaction with `.ANY` priority

Priority is considered after assignment to the most specific selector in all algorithms.

`worst_fit` and `best_fit` consider priority before their individual placement criteria. For example, you might have `.ANY1` and `.ANY2` selectors, with the `.ANY1` region having the most free space. When using `worst_fit` the section is assigned to `.ANY2` because it has higher priority. Only if the priorities are equal does the algorithm come into play.

`first_fit` considers the most specific selector first, then priority. It does not introduce any more placement rules.

`next_fit` also does not introduce any more placement rules. If a region is marked full during `next_fit`, that region cannot be considered again regardless of priority.

Interaction with `--any_contingency`

The priority of a `.ANY` selector is reduced to `0` if the region might overflow because of linker-generated content. This is enabled and disabled independently of the sorting and placement algorithms.

`armlink` calculates a worst-case contingency for each section.

For `worst_fit`, `best_fit`, and `first_fit`, when a region is about to overflow because of the contingency, `armlink` lowers the priority of the related `.ANY` selector.

For `next_fit`, when a possible overflow is detected, `armlink` marks that section as `FULL` and does not consider it again. This stays consistent with the rule that when a section is full it can never be revisited.

Default

The default option is `worst_fit`.

See also

- *--any_contingency* on page 2-8
- *--any_sort_order=order*
- *--info=topic[,topic,...]* on page 2-44
- *--scatter=file* on page 2-78
- *Syntax of an input section description* on page 3-18
- *Behavior when .ANY sections overflow because of linker-generated content* on page 3-27
- the following in the *Linker User Guide*:
  - *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.

2.1.5 `--any_sort_order=`*order*

Controls the sort order of input sections that are placed using the `.ANY` module selector.

Syntax

`--any_sort_order=`*order*

where *order* is one of the following:

`descending_size`
: Sort input sections in descending size order.

`cmdline`
: Sort input sections by command-line index.

By default, sections that have the same properties are resolved using the creation index. You can use the `--tiebreaker` command-line option to resolve sections by the order they appear on the linker command-line.

Use this option with the `--scatter` option.

Usage

The sorting governs the order that sections are processed during `.ANY` assignment. Normal scatter-loading rules, for example `R0` before `RW`, are obeyed after the sections are assigned to regions.

Default

The default option is `descending_size`.

See also

- *--any_contingency* on page 2-8
- *--any_placement=algorithm* on page 2-8
- *--info=topic[,topic,...]* on page 2-44
- *--scatter=file* on page 2-78
- *--tiebreaker=option* on page 2-92
- *Syntax of an input section description* on page 3-18
- the following in the *Linker User Guide*:
  - *Placing unassigned sections with the .ANY module selector* on page 5-26
  - *Examples of using sorting algorithms for .ANY sections* on page 5-35.

2.1.6 `--arm_linux`

This option specifies default settings for use when creating Linux applications.

Default

The following default settings are automatically specified:

- `--add_needed`

- `--add_shared_references`
- `--gnu_linker_defined_syms`
- `--keep=*(.init)`
- `--keep=*(.init_array)`
- `--keep=*(.fini)`
- `--keep=*(.fini_array)`
- `--linux_abitag=2.6.12`
- `--no_as_needed`
- `--no_ref_cpp_init`
- `--no_scanlib`
- `--no_startup`
- `--prelink_support`
- `--sysv`

To override any of the default settings, specify them separately after the `--arm_linux` option.

When migrating from a toolchain earlier than RVCT v4.0, you can replace all of the defaults with a single `--arm_linux` option.

See also

- *--add_needed, --no_add_needed* on page 2-6
- *--add_shared_references, --no_add_shared_references* on page 2-7
- *--as_needed, --no_as_needed* on page 2-13
- *--gnu_linker_defined_syms* on page 2-43
- *--keep=section_id* on page 2-49
- *--library=name* on page 2-54
- *--linux_abitag=version_id* on page 2-56
- *--prelink_support, --no_prelink_support* on page 2-68
- *--ref_cpp_init, --no_ref_cpp_init* on page 2-71
- *--scanlib, --no_scanlib* on page 2-77
- *--search_dynamic_libraries, --no_search_dynamic_libraries* on page 2-78
- *--startup=symbol, --no_startup* on page 2-83
- *--sysv* on page 2-90
- *--arm_linux* on page 2-9 in the *Compiler Reference Guide*.

2.1.7 --arm_only

This option enables the linker to target the ARM instruction set only. If the linker detects any objects requiring Thumb® state, an error is generated.

See also

- the following in the *Compiler Reference Guide*:
  - *--arm* on page 2-8
  - *--arm_only* on page 2-15
  - *--thumb* on page 2-122
- the following in the *Assembler Guide*:
  - *Command syntax* on page 3-2.

2.1.8 --as_needed, --no_as_needed

Controls whether or not a reference to a shared library is added to the DT_NEEDED tags.

Usage

The effect of this option depends on the position on the `armlink` command-line, and applies only to subsequent dynamic shared objects:

- `--as_needed` adds references to subsequent shared objects to the DT_NEEDED tags only if the shared objects are used to resolve symbols
- `--no_as_needed` unconditionally adds references to the DT_NEEDED tags.

Default

The default is `--as_needed`.

However, if you specify `--arm_linux`, the default is `--no_as_needed`.

Example

The following example unconditionally adds a reference to `liby.so` in the DT_NEEDED tags, but only adds tags for `libx.so` and `libz.so` if they are used to resolve symbols:

```
armlink ... libx.so --no-as-needed liby.so --as-needed libz.so
```

See also

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

2.1.9 --autoat, --no_autoat

This option controls the automatic assignment of __at sections to execution regions. __at sections are sections that must be placed at a specific address.

Usage

If enabled, the linker automatically selects an execution region for each __at section. If a suitable execution region does not exist, the linker creates a load region and an execution region to contain the __at section.

If disabled, the standard scatter-loading section selection rules apply.

Default

The default is --autoat.

Restrictions

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

See also

- Chapter 3 *Formal syntax of the scatter-loading description file*
- *Examples of specifying region and section addresses* on page 5-10 in the *Linker User Guide*.

2.1.10 --base_platform

This option specifies the Base Platform linking model. It is a superset of the *Base Platform Application Binary Interface* (BPABI) model, --bpabi option.

Usage

When you specify --base_platform, the linker also acts as if you specified --bpabi with the following exceptions:

- The full choice of memory models is available, including scatter-loading:
  - --dll

  - `--force_so_throw`, `--no_force_so_throw`
  - `--pltgot=`*`type`*
  - `--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 specify `--pltgot_opts=crosslr` then calls to and from a load region marked `RELOC` go by way of the *Procedure Linkage Table* (PLT).

To place unresolved weak references in the dynamic symbol table, use the `IMPORT` steering file command.

Note

If you are linking with `--base_platform`, and the parent load region has the `RELOC` attribute, then all execution regions within that load region must have a `+`*`offset`* base address.

See also

- *--bpabi* on page 2-17
- *--pltgot=type* on page 2-65
- *--pltgot_opts=mode* on page 2-66
- *--scatter=file* on page 2-78
- *Addressing attributes* on page 3-13
- *Linker User Guide*:
  - *Base Platform Application Binary Interface (BPABI) linking model* on page 2-6
  - *Base Platform linking model* on page 2-7.

2.1.11 `--be8`

This option specifies ARMv6 Byte Invariant Addressing big-endian mode.

This is the default Byte Addressing mode for ARMv6 big-endian images and means that the linker reverses the endianness of the instructions to give little-endian code and big-endian data for input objects that have been compiled/assembled as big-endian.

Byte Invariant Addressing mode is only available on ARM processors that support ARMv6 and above.

See also

- *ARM architecture v6* on page 2-12 in the *Developer Guide*
- *ARM Architecture Reference Manual*.

2.1.12 --be32

This option specifies legacy Word Invariant Addressing big-endian mode, that is, identical to big-endian images prior to ARMv6. This produces big-endian code and data.

Word Invariant Addressing mode is the default mode for all pre-ARMv6 big-endian images.

See also

- *ARM architecture v6* on page 2-12 in the *Developer Guide*
- *ARM Architecture Reference Manual*.

2.1.13 --bestdebug, --no_bestdebug

This option selects between linking for smallest code/data size or best debug illusion. Input objects might contain *common data* (COMDAT) groups, but these might not be identical across all input objects because of differences. For example, inlining.

Default

The default is `--no_bestdebug`. This ensures that the code and data of the final image are the same regardless of whether you compile for debug or not. The smallest COMDAT groups are selected when linking, at the expense of a possibly slightly poorer debug illusion.

Usage

Use `--bestdebug` to select COMDAT groups with the best debug view. Be aware that the code and data of the final image might not be the same when building with or without debug.

See also

- *Section elimination* on page 3-13 in the *Linker User Guide*.

2.1.14 --bpabi

This option creates a *Base Platform Application Binary Interface* (BPABI) ELF file for passing to a platform-specific post-linker.

The BPABI model defines a standard-memory model that enables interoperability of BPABI-compliant files across toolchains. When this option is selected:

- *Procedure Linkage Table* (PLT) and *Global Offset Table* (GOT) generation is supported.
- The default value of the `--pltgot` option is `direct`.
- a *dynamic link library* (DLL) placed on the command-line can define symbols.

Restrictions

The BPAPI model does not support scatter-loading. However, scatter-loading is supported in the Base Platform model.

Weak references in the dynamic symbol table are allowed only if the symbol table is defined by a DLL placed on the command-line. You cannot place an unresolved weak reference in the dynamic symbol table with the `IMPORT` steering file command.

See also

- *--base_platform* on page 2-14
- *--dll* on page 2-30
- *--pltgot=type* on page 2-65
- *--shared* on page 2-80
- *--sysv* on page 2-90
- Chapter 4 *BPABI and SysV Shared Libraries and Executables*
- *Linker User Guide*:
  - *Base Platform Application Binary Interface (BPABI) linking model* on page 2-6
  - *Base Platform linking model* on page 2-7.

2.1.15 --branchnop, --no_branchnop

This option causes the linker to replace any branch with a relocation that resolves to the next instruction with a `NOP`. This is the default behavior. However, there are cases where you might want to disable the option, for example, when performing verification or pipeline flushes.

Default

The default is `--branchnop`.

Use `--no_branchnop` to disable this behavior.

See also

- *--inline, --no_inline* on page 2-47
- *--tailreorder, --no_tailreorder* on page 2-91
- *Inlining* on page 3-26 in the *Linker User Guide*.

2.1.16 `--callgraph, --no_callgraph`

This option creates a file containing a static callgraph of functions. The callgraph gives definition and reference information for all functions in the image.

Usage

The callgraph file:

- is saved in the same directory as the generated image.
- has the same name as the linked image. Use the `--callgraph_file=`*`filename`* option to specify a different callgraph filename.
- has a default output format of HTML. Use the `--callgraph_output=`*`fmt`* option to control the output format.

Note

If the linker is to calculate the function stack usage, any functions defined in the assembler files must have the appropriate:

- `PROC` and `ENDP` directives
- `FRAME PUSH` and `FRAME POP` directives.

For each function `func` the linker lists the:

- processor state for which the function is compiled (ARM or Thumb)
- set of functions that call `func`
- set of functions that are called by `func`
- number of times the address of `func` is used in the image.

In addition, the callgraph identifies functions that are:

- called through interworking veneers

- defined outside the image
- permitted to remain undefined (weak references)
- called through a *Procedure Linkage Table* (PLT)
- not called but still exist in the image.

The static callgraph also gives information about stack usage. It lists the:

- size of the stack frame used by each function
- maximum size of the stack used by the function over any call sequence, that is, over any acyclic chain of function calls.

If there is a cycle, or if the linker detects a function with no stack size information in the call chain, `+ Unknown` is added to the stack usage. A reason is added to indicate why stack usage is unknown.

The linker reports missing stack frame information if there is no debug frame information for the function.

For indirect functions, the linker cannot reliably determine which function made the indirect call. This might affect how the maximum stack usage is calculated for a call chain. The linker lists all function pointers used in the image.

Use frame directives in assembly language code to describe how your code uses the stack. These directives ensure that debug frame information is present for debuggers to perform stack unwinding or profiling.

Default

The default is `--no_callgraph`.

See also

- *--callgraph_file=filename*
- *--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
- Chapter 3 *Formal syntax of the scatter-loading description file*
- Chapter 7 *Directives Reference* in the *Assembler Guide*.

2.1.17 `--callgraph_file=`*`filename`*

This option controls the output filename of the callgraph.

The default filename is the same as the linked image.

Note

If you use the `--partial` option to create a partially linked object, then no callgraph file is created.

See also

- *--callgraph, --no_callgraph* on page 2-18
- *--callgraph_output=fmt*
- *--cgfile=type* on page 2-21
- *--cgsymbol=type* on page 2-21
- *--cgundefined=type* on page 2-22
- *--output=file* on page 2-62
- Chapter 3 *Formal syntax of the scatter-loading description file*.

2.1.18 `--callgraph_output=`*`fmt`*

This option controls the output format of the callgraph.

Syntax

`--callgraph_output=`*`fmt`*

Where *`fmt`* can be one of the following:

`html`	Outputs the callgraph in HTML format.
`text`	Outputs the callgraph in plain text format.

Default

The default is `--callgraph_output=html`.

See also

- *--callgraph, --no_callgraph* on page 2-18
- *--callgraph_file=filename* on page 2-19
- *--cgfile=type* on page 2-21
- *--cgsymbol=type* on page 2-21
- *--cgundefined=type* on page 2-22
- Chapter 3 *Formal syntax of the scatter-loading description file*.

2.1.19 --cgfile=*type*

This option controls what files are used to obtain the symbols to be included in the callgraph.

Syntax

`--cgfile=`*type*

where *type* can be one of the following:

`all`	Includes symbols from all files.
`user`	Includes only symbols from user defined objects and libraries.
`system`	Includes only symbols from system libraries.

Default

The default is `--cgfile=all`.

See also

- *--callgraph, --no_callgraph* on page 2-18
- *--callgraph_file=filename* on page 2-19
- *--callgraph_output=fmt* on page 2-20
- *--cgsymbol=type*
- *--cgundefined=type* on page 2-22
- Chapter 3 *Formal syntax of the scatter-loading description file*.

2.1.20 --cgsymbol=*type*

This option controls what symbols are included in the callgraph.

Syntax

Where *type* can be one of the following:

`all`	Includes both local and global symbols.
`locals`	Includes only local symbols.
`globals`	Includes only global symbols.

Default

The default is `--cgsymbol=all`.

See also

- *--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
- *--cgundefined=type*
- Chapter 3 *Formal syntax of the scatter-loading description file*.

2.1.21 --cgundefined=*type*

This option controls what undefined references are included in the callgraph.

Syntax

`--cgundefined=`*type*

Where *type* can be one of the following:

`all`	Includes both function entries and calls to undefined weak references.
`entries`	Includes function entries for undefined weak references.
`calls`	Includes calls to undefined weak references.
`none`	Omits all undefined weak references from the output.

Default

The default is `--cgundefined=all`.

See also

- *--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
- Chapter 3 *Formal syntax of the scatter-loading description file*.

2.1.22 --combreloc, --no_combreloc

This option enables or disables the linker reordering of the dynamic relocations so that a dynamic loader can process them more efficiently.

Default

The default is `--combreloc`.

2.1.23 `--comment_section, --no_comment_section`

This option controls the inclusion of a comment section `.comment` in the final image.

Use `--no_comment_section` to strip the text in the `.comment` section, to help reduce the image size.

Note

You can also use the `--filtercomment` option to merge comments.

Default

The default is `--comment_section`.

See also

- *--filtercomment, --no_filtercomment* on page 2-39
- *About merging comment sections* on page 3-29 in the *Linker User Guide*.

2.1.24 `--compress_debug, --no_compress_debug`

This option causes the linker to compress `.debug_*` sections, if it is sensible to do so. This removes some redundancy and reduces debug table size. Using `--compress_debug` can significantly increase the time required to link an image. Debug compression can only be performed on DWARF3 debug data, not DWARF2.

Default

The default is `--no_compress_debug`.

2.1.25 `--cppinit, --no_cppinit`

This option enables the linker to use alternative C++ libraries with a different initialization symbol if required.

Syntax

`--cppinit=`*`symbol`*

If `--cppinit=symbol` is not specified then the default, `__cpp_initialize__aeabi_` is assumed.

`--no_cppinit` does not take a *symbol* argument.

Effect

The linker adds a non-weak reference to *symbol* if any static constructor or destructor sections are detected.

For `--cppinit=__cpp_initialize__aeabi_`, the linker processes R_ARM_TARGET1 relocations as R_ARM_REL32, because this is required by the `__cpp_initialize__aeabi_` function. In all other cases R_ARM_TARGET1 relocations are processed as R_ARM_ABS32.

See also

- *--ref_cpp_init, --no_ref_cpp_init* on page 2-71.

2.1.26 `--cpu=list`

This option lists the supported architecture and processor names that you can use with `--cpu=name`.

See also

- *--cpu=name*.

2.1.27 `--cpu=name`

This option enables the linker to determine the target ARM processor or architecture. This option has the same format as that supported by the compiler.

Syntax

`--cpu=name`

Where *name* is the name of an ARM processor or architecture.

Usage

The link phase fails if any of the component object files rely on features that are incompatible with the selected processor. The linker also uses this option to optimize the choice of system libraries and any veneers that need to be generated when building

the final image. The default is to select a CPU that is compatible with all of the component object files. That is, to select the most up-to-date architecture among all input objects.

Note

If the `--cpu` option has a built-in *floating-point unit* (FPU) then the linker implies `--fpu=built-in_fpu`. For example, `--cpu=cortex-a8` implies `--fpu=vfpv3`.

See also

- *--cpu=list* on page 2-24
- *--fpu=list* on page 2-42
- *--fpu=name* on page 2-42
- the following in the *Compiler Reference Guide*:
  - *--cpu=name* on page 2-30
  - *--fpu=list* on page 2-62
  - *--fpu=name* on page 2-62.

2.1.28 `--crosser_veneershare`, `--no_crosser_veneershare`

Enables or disables veneer sharing across execution regions.

The default is `--crosser_veneershare`, and enables veneer sharing across execution regions.

`--no_crosser_veneershare` prohibits veneer sharing across execution regions.

See also

- *--veneershare, --no_veneershare* on page 2-97.

2.1.29 `--datacompressor=opt`

This option enables you to specify one of the supplied algorithms for RW data compression. If you do not specify a data compression algorithm, the linker chooses the most appropriate one for you automatically. In general, it is not necessary to override this choice.

Syntax

`--datacompressor=opt`

Where *opt* is one of the following:

`on` Enables RW data compression to minimize ROM size.

`off` Disables RW data compression.

`list` Lists the data compressors available to the linker.

`id` *id* is a data compression algorithm as shown in Table 2-1.

Table 2-1 Data compressor algorithms

id	Compression algorithm
0	run-length encoding
1	run-length encoding, with LZ77 on small-repeats
2	complex LZ77 compression

Specifying a compressor adds a decompressor to the code area. If the final image does not have compressed data, the decompressor is not added.

Default

The default is `--datacompressor=on`.

See also

- *RW data compression* on page 3-20 in the *Linker User Guide*.

2.1.30 `--debug, --no_debug`

This option controls the generation of debug information in the output file. Debug information includes debug input sections and the symbol/string table.

Default

The default is `--debug`.

Usage

Use `--no_debug` to exclude debug information from the output file. The resulting ELF image is smaller, but you cannot debug it at source level. The linker discards any debug input section it finds in the input objects and library members, and does not include the

symbol and string table in the image. This only affects the image size as loaded into the debugger. It has no effect on the size of any resulting binary image that is downloaded to the target.

If you are using `--partial` the linker creates a partially-linked object without any debug data.

Note

Do not use `--no_debug` if a `fromelf --fieldoffsets` step is required. If your image is produced without debug information, `fromelf` cannot:

- translate the image into other file formats
- produce a meaningful disassembly listing.

See also

- *--fieldoffsets* on page 2-29 in the *Utilities Guide*.

2.1.31 `--device=list`

This option lists the supported device names that can be used with the `--device=`*`name`* option.

See also

- *--device=name*.

2.1.32 `--device=`*`name`*

This option selects a specific device and associated processor settings. This option follows the same format as that supported by the ARM compiler.

Note

The link phase fails if any of the component object files rely on features that are incompatible with the selected processor. The linker also uses this option to optimize the choice of system libraries and any veneers that need to be generated when building the final image. The default is to select a device that is compatible with all of the component object files.

Syntax

`--device=`*`name`*

where *name* is a specific device name.

To get a full list of the available devices, use the `--device=list` option.

See also

- *--device=list* on page 2-27
- *--device=name* on page 2-44 in the *Compiler Reference Guide*.

2.1.33 `--diag_error=tag[,tag,...]`

This option sets diagnostic messages that have a specific tag to error severity.

Syntax

`--diag_error=tag[,tag,...]`

Where *tag* can be:

- a diagnostic message number to set to error severity
- `warning`, to treat all warnings as errors.

See also

- *--diag_remark=tag[,tag,...]*
- *--diag_warning=tag[,tag,...]* on page 2-29.

2.1.34 `--diag_remark=tag[,tag,...]`

This option sets diagnostic messages that have a specific tag to remark severity.

Syntax

`--diag_remark=tag[,tag,...]`

Where *tag* is a comma-separated list of diagnostic message numbers.

See also

- *--diag_error=tag[,tag,...]*
- *--diag_warning=tag[,tag,...]* on page 2-29.

2.1.35 `--diag_style=arm|ide|gnu`

This option changes the formatting of warning and error messages.

Default

The default is `--diag_style=arm`.

Usage

`--diag_style=gnu` matches the format reported by the GNU Compiler, `gcc`.

`--diag_style=ide` matches the format reported by Microsoft Visual Studio.

2.1.36 `--diag_suppress=tag[,tag,...]`

This option suppresses all diagnostic messages that have a specific tag.

Syntax

`--diag_suppress=tag[,tag,...]`

Where *tag* is a comma-separated list of diagnostic message numbers that must be suppressed.

Example

To suppress the warning messages that have numbers `L6314W` and `L6305W`, use the following command:

```
armlink --diag_suppress=L6314,L6305 ...
```

2.1.37 `--diag_warning=tag[,tag,...]`

This option sets diagnostic messages that have a specific tag to warning severity.

Syntax

`--diag_warning=tag[,tag,...]`

Where *tag* can be:

- a diagnostic message number to set to warning severity
- `error`, to downgrade all errors to warnings.

See also

- *--diag_error=tag[,tag,...]* on page 2-28
- *--diag_remark=tag[,tag,...]* on page 2-28.

2.1.38 --dll

This option creates a BPABI *dynamically linked library* (DLL). The DLL is marked as a shared object in the ELF file header.

You must use --bpabi with --dll to produce a BPABI-compliant DLL.

You can also use --dll with --base_platform.

Note

By default, this option disables unused section elimination. Use the --remove option to re-enable unused sections when building a DLL.

See also

- *--base_platform* on page 2-14
- *--bpabi* on page 2-17
- *--remove, --no_remove* on page 2-74
- *--shared* on page 2-80
- *--sysv* on page 2-90
- Chapter 4 *BPABI and SysV Shared Libraries and Executables*.

2.1.39 --dynamic_debug

This option forces the linker to output dynamic relocations for debug sections. Using this option allows an OS-aware debugger, to debug shared libraries produced by armlink.

Use --dynamic_debug with --sysv and --sysv --shared images and shared libraries.

See also

- *--shared* on page 2-80
- *--sysv* on page 2-90
- Chapter 4 *BPABI and SysV Shared Libraries and Executables*.

2.1.40 --dynamic_linker=*name*

This option specifies the dynamic linker to use to load and relocate the file at runtime.

Usage

When you link with shared objects, the dynamic linker to use is stored in the executable. This option specifies a particular dynamic linker to use when the file is executed. If you are working on ARM Linux platforms, the linker assumes that the default dynamic linker is `/lib/ld-linux.so.3`.

See also

- Chapter 4 *BPABI and SysV Shared Libraries and Executables*.

2.1.41 `--eager_load_debug`, `--no_eager_load_debug`

The `--no_eager_load_debug` option causes the linker to remove debug section data from memory after object loading. This lowers the peak memory usage of the linker at the expense of some linker performance, because much of the debug data has to be loaded again when the final image is written.

Using `--no_eager_load_debug` option does not affect the debug data that is written into the ELF file.

The default is `--eager_load_debug`.

Note

The resulting image or object built without debug information might differ by a small number of bytes. This is because the `.comment` section contains the linker command line used, where the options have differed from the default (the default is `--eager_debug_load`). Therefore `--no_eager_load_debug` images are a little larger and contain Program Header and possibly a Section Header a small number of bytes later. Use `--no_comment_section` to eliminate this difference.

See also

- *--comment_section, --no_comment_section* on page 2-23.

2.1.42 `--edit=`*`file_list`*

This option enables you to specify steering files containing commands to edit the symbol tables in the output binary. You can specify commands in a steering file to:

- Hide global symbols. Use this option to hide specific global symbols in object files. The hidden symbols are not publicly visible.
- Rename global symbols. Use this option to resolve symbol naming conflicts.

Syntax

`--edit=file_list`

Where *file_list* can be more than one steering file separated by a comma. Do not include a space after the comma.

Example

```
--edit=file1 --edit=file2 --edit=file3
```

```
--edit=file1,file2,file3
```

See also

- *Steering file commands in alphabetical order* on page 2-102
- *Hiding and renaming global symbols* on page 4-16.

2.1.43 --emit_debug_overlay_relocs

Outputs only relocations of debug sections with respect to overlaid program sections to aid an overlay-aware debugger.

See also

- *--emit_debug_overlay_section*
- *--emit_non_debug_relocs* on page 2-33
- *--emit_relocs* on page 2-33
- *ABI for the ARM Architecture: Support for Debugging Overlaid Programs*.

2.1.44 --emit_debug_overlay_section

In a relocatable file, a debug section refers to a location in a program section by way of a relocated location. A reference from a debug section to a location in a program section has the following format:

<debug_section_index, debug_section_offset>, <program_section_index, program_section_offset>

During static linking the pair of *program* values is reduced to single value, the execution address. This is ambiguous in the presence of overlaid sections.

To resolve this ambiguity, use this option to output a `.ARM.debug_overlay` section of type `SHT_ARM_DEBUG_OVERLAY = SHT_LOUSER + 4` containing a table of entries as follows:

debug_section_offset, debug_section_index, program_section_index

See also

- *--emit_debug_overlay_relocs* on page 2-32
- *--emit_relocs*
- *ABI for the ARM Architecture: Support for Debugging Overlaid Programs*.

2.1.45 --emit_non_debug_relocs

Retains only relocations from non-debug sections in an executable file.

See also

- *--emit_relocs*.

2.1.46 --emit_relocs

Retains all relocations in the executable file. This results in larger executable files.

This is equivalent to the GNU ld `--emit-relocs` option.

See also

- *--emit_debug_overlay_relocs* on page 2-32
- *--emit_debug_overlay_section* on page 2-32
- *--emit_non_debug_relocs*
- *ABI for the ARM Architecture: Support for Debugging Overlaid Programs*.

2.1.47 --entry=*location*

This option specifies the unique initial entry point of the image.

Syntax

`--entry=`*location*

Where *location* is one of the following:

entry_address
: A numerical value, for example: `--entry=0x0`.

symbol
: Specifies an image entry point as the address of *symbol*, for example: `--entry=reset_handler`.

offset+object(*section*)

Specifies an image entry point as an *offset* inside a *section* within a particular *object*, for example:
`--entry=8+startup.o(startupseg)`

There must be no spaces within the argument to `--entry`. The input section and object names are matched without case-sensitivity. You can use the following simplified notation:

- `object(section)`, if `offset` is zero.
- `object`, if there is only one input section. `armlink` generates an error message if there is more than one code input section in `object`.

Note

If the entry address of your image is in Thumb state, then the least significant bit of the address must be set to 1. The linker does this automatically if you specify a symbol. For example, if the entry code starts at address `0x8000` in Thumb state you must use `--entry=0x8001`.

Note

If you use `--ltcg`, then only `--entry=`*symbol* can be used.

Usage

The image can contain multiple entry points, but the initial entry point specified with this option is stored in the executable file header for use by the loader. There can be only one occurrence of this option on the command line. RealView® Debugger uses this entry address to initialize the *Program Counter* (PC) when an image is loaded. The initial entry point must meet the following conditions:

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

See also

- *--ltcg* on page 2-59
- *--startup=symbol, --no_startup* on page 2-83.

2.1.48 `--errors=`*file*

This option redirects the diagnostics from the standard error stream to *file*.

The specified file is created at the start of the link stage. If a file of the same name already exists, it is overwritten.

If *file* is specified without path information, it is created in the current directory.

2.1.49 --exceptions, --no_exceptions

This option controls the generation of exception tables in the final image.

Default

The default is --exceptions.

Usage

Using --no_exceptions generates an error message if any exceptions sections are present in the image after unused sections have been eliminated. Use this option to ensure that your code is exceptions free.

See also

- *Using command-line options to handle C++ exceptions* on page 3-36 in the *Linker User Guide*.

2.1.50 --exceptions_tables=*action*

This option specifies how exception tables are generated for objects that do not already contain exception unwinding tables.

Syntax

--exceptions_tables=*action*

Where *action* is one of the following:

nocreate — The linker does not create missing exception tables.

unwind — The linker creates an unwinding table for each section in your image that does not already have an exception table.

cantunwind — The linker creates a nounwind table for each section in your image that does not already have an exception table.

Default

The default is `--exceptions_tables=nocreate`.

See also

- *Using command-line options to handle C++ exceptions* on page 3-36 in the *Linker User Guide*.

2.1.51 `--execstack, --no_execstack`

To support non-executable stacks, the linker generates the appropriate `PT_GNU_STACK` program header when you specify either:

- `--sysv`
- `--arm_linux`, because this option implies `--sysv`.

The linker derives the executable status of the stack from the presence of the `.note.GNU-stack` section in input objects:

- If any of the input objects does not contain a `.note.GNU-stack` section, the linker assumes the final image requires an executable stack.
- If no input object has a `.note.GNU-stack` section then the linker does not generate a PT_GNU_STACK program header.
- If at least one object has a `.note.GNU-stack` then the linker generates a PT_GNU_STACK program header. It is marked non-executable if all input objects have a `.note.GNU-stack` section that is non-executable. In all other cases the program header is marked executable.

To override the choice made by the linker, use:

- `--execstack` to force the use of an executable stack
- `--no_execstack` to force the use of a non-executable stack.

See also

- *--arm_linux* on page 2-11
- *--sysv* on page 2-90.

2.1.52 `--export_all, --no_export_all`

This option controls the exporting of symbols to the dynamic symbols table.

Default

The default is `--export_all` for building shared libraries and *dynamically linked libraries* (DLLs).

The default is `--no_export_all` for building applications.

Usage

Use `--export_all` to dynamically export all global, non-hidden symbols from the executable or DLL to the dynamic symbol table. Use `--no_export_all` to prevent the exporting of symbols to the dynamic symbols table.

`--export_all` always exports non-hidden symbols into the dynamic symbol table. The dynamic symbol table is created if necessary.

You cannot use `--export_all` to produce a statically linked image because it always exports non-hidden symbols, forcing the creation of a dynamic segment.

For more precise control over the exporting of symbols, use one or more steering files.

See also

- *--export_dynamic, --no_export_dynamic*
- *Hiding and renaming global symbols* on page 4-16 in the *Linker User Guide*.

2.1.53 `--export_dynamic`, `--no_export_dynamic`

If an executable has dynamic symbols, then `--export_dynamic` exports all externally visible symbols.

Usage

`--export_dynamic` exports non-hidden symbols into the dynamic symbol table only if a dynamic symbol table already exists.

You can use `--export_dynamic` to produce a statically linked image if there are no imports or exports.

`--no_export_dynamic` is the default.

See also

- *--export_all, --no_export_all* on page 2-36.

2.1.54 --feedback=*file*

This option generates a feedback file, for the next time a file is compiled, to inform the compiler about unused functions.

When you next compile the file, use the compiler option `--feedback=file` to specify the feedback file to use. Unused functions are placed in their own sections for possible future elimination by the linker.

See also

- *--feedback_image=option*
- *--feedback_type=type* on page 2-39
- *--feedback=filename* on page 2-56 in the *Compiler User Guide*
- *Feedback* on page 3-17 in the *Linker User Guide*.

2.1.55 --feedback_image=*option*

This option changes the behavior of the linker when writing a feedback file. Use this option to produce a feedback file where an executable ELF image cannot normally be created.

Syntax

`--feedback_image=option`

Where *option* is one of the following:

`none` Uses the scatter file to determine region size limits. Disables region overlap and region size overflow messages. Does not write an ELF image. Error messages are still produced if a region overflows the 32-bit address space.

`noerrors` Uses the scatter file to determine region size limits. Warns on region overlap and region size overflow messages. Writes an ELF image, which might not be executable. Error messages are still produced if a region overflows the 32-bit address space.

`simple` Ignores the scatter file. Disables ROPI/RWPI errors and warnings. Writes an ELF image, which might not be executable.

`full` Enables all error and warning messages and writes a valid ELF image.

Default

The default option is `--feedback_image=full`.

Example

Use `--feedback_image=noerrors` if your code does not fit into the region limits described in your scatter file before unused functions are removed using linker feedback.

See also

- *--feedback=file* on page 2-38
- *--feedback_type=type*
- *--feedback=filename* on page 2-56 in the *Compiler User Guide*
- *Feedback* on page 3-17 in the *Linker User Guide*.

2.1.56 `--feedback_type=type`

This option controls the information that the linker puts into the feedback file.

Syntax

`--feedback_type=type`

Where *type* is a comma-separated list from the following topic keywords:

`[no]iw` controls functions that require interworking support.

`[no]unused` controls unused functions in the image.

Default

The default option is `--feedback_type=unused,noiw`.

See also

- *--feedback=file* on page 2-38
- *--feedback_image=option* on page 2-38
- *--feedback=filename* on page 2-56 in the *Compiler User Guide*
- *Feedback* on page 3-17 in the *Linker User Guide*.

2.1.57 `--filtercomment, --no_filtercomment`

The linker always removes identical comments. The `--filtercomment` option allows the linker to pre-process the `.comment` section and remove some information that prevents merging.

Use `--no_filtercomment` to prevent the linker from modifying the `.comment` section.

Default

The default is `--filtercomment`.

2.1.58 `--fini=symbol`

This option specifies the symbol name that is used to define the entry point for finalization code. The dynamic linker executes this code when it unloads the executable file or shared object.

See also

- Chapter 4 *BPABI and SysV Shared Libraries and Executables*.

2.1.59 `--first=section_id`

This option places the selected input section first in its execution region. This can, for example, place the section containing the vector table first in the image.

Syntax

`--first=section_id`

Where *section_id* is one of the following:

symbol Selects the section that defines *symbol*. You must not specify a symbol that has more than one definition, because only one section can be placed first. For example: `--first=reset`

object(*section*)

Selects *section* from *object*. There must be no space between *object* and the following open parenthesis. For example: `--first=init.o(init)`

object Selects the single input section in *object*. If you use this short form and there is more than one input section, the linker generates an error message. For example: `--first=init.o`

Usage

The `--first` option cannot be used with `--scatter`. Instead, use the `+FIRST` attribute in a scatter-loading file.

See also

- *Section placement* on page 3-10 in the *Linker User Guide*.

2.1.60 --force_explicit_attr

The --cpu option checks the FPU attributes if the CPU chosen has a built-in FPU.

The error message L6463E: Input Objects contain *archtype* instructions but could not find valid target for *archtype* architecture based on object attributes. Suggest using --cpu option to select a specific cpu. is given in one of two situations:

- the ELF file contains instructions from architecture *archtype* yet the build attributes claim that *archtype* is not supported
- the build attributes are inconsistent enough that the linker cannot map them to an existing CPU.

If setting the --cpu option still fails, use --force_explicit_attr to cause the linker to retry the CPU mapping using build attributes constructed from --cpu=*archtype*. This might help if the error is being given solely because of inconsistent build attributes.

See also

- *--cpu=name* on page 2-24
- *--fpu=name* on page 2-42
- *--cpu=name* on page 2-30 in the *Compiler Reference Guide*
- *--fpu=name* on page 2-62 in the *Compiler Reference Guide*.
- *Command syntax* on page 3-2 in the *Assembler Guide*.

2.1.61 --force_so_throw, --no_force_so_throw

This option controls whether an image can throw an exception or not. By default, exception tables are discarded if no code throws an exception.

Default

The default is --no_force_so_throw.

Usage

Use --force_so_throw to specify that all shared objects might throw an exception and so force the linker to keep the exception tables, regardless of whether the image can throw an exception or not. If the --sysv option is used then --force_so_throw is automatically set.

See also

- *--sysv* on page 2-90

- Chapter 4 *BPABI and SysV Shared Libraries and Executables*.

2.1.62 --fpic

This option enables you to link *Position-Independent Code* (PIC), that is, code that has been compiled using the `--apcs=/fpic` qualifier. Relative addressing is only implemented when your code makes use of System V shared libraries.

Note

The linker outputs a downgradeable error if `--shared` is used and `--fpic` is not used.

See also

- *--shared* on page 2-80.

2.1.63 --fpu=list

This option lists the supported FPU architecture names that you can use with the `--fpu=name` option.

See also

- *--fpu=name*.

2.1.64 --fpu=*name*

This option enables the linker to determine the target FPU architecture.

The linker fails if any of the component object files rely on features that are incompatible with the selected FPU architecture. The linker also uses this option to optimize the choice of system libraries and any veneers that need to be generated when building the final image. The default is to select an FPU that is compatible with all of the component object files.

This option has the same format as that supported by the compiler.

See also

- *--fpu=list*
- *--fpu=name* on page 2-62 in the *Compiler Reference Guide*.

2.1.65 --gnu_linker_defined_syms

This option enables support for the GNU equivalent of input section symbols.

Table 2-2 GNU equivalent of input sections

GNU Symbol	ARM symbol	Description
__start_*SectionName*	*SectionName*$$Base	Address of the start of the consolidated section called *SectionName*.
__stop_*SectionName*	*SectionName*$$Limit	Address of the byte beyond the end of the consolidated section called *SectionName*

Note

- A reference to *SectionName* by a GNU input section symbol is sufficient for armlink to prevent the section from being removed as unused.
- A reference by an ARM input section symbol is not sufficient to prevent the section from being removed as unused.

This option is enabled by default when you specify --arm_linux. It is disabled by default in all other cases.

Usage

If you want GNU-style behavior when treating the ARM symbols SectionName$$Base and SectionName$$Limit, then specify --gnu_linker_defined_syms.

See also

- *--arm_linux* on page 2-11.

2.1.66 --help

This option displays a summary of the main command-line options.

Default

This is the default if you specify armlink without any options or source files.

See also

- *--show_cmdline* on page 2-80

- *--vsn* on page 2-100.

2.1.67 --import_unresolved, --no_import_unresolved

When linking a shared object with `--sysv --shared` unresolved symbols are normally imported.

If you explicitly list object files on the linker command-line, specify the `--no_import_unresolved` option so that any unresolved references cause an undefined symbol error rather than being imported.

`--import_unresolved` is the default option.

See also

- *--shared* on page 2-80
- *--sysv* on page 2-90.

2.1.68 --info=*topic[,topic,...]*

This option prints information about specific topics. You can write the output to a text file using `--list=`*`file`*.

Syntax

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

Where *`topic`* is a comma-separated list from the following topic keywords:

`any` For sections placed using the `.ANY` module selector, lists:

- the sort order
- the placement algorithm
- the sections that are assigned to each execution region in the order they are assigned by the placement algorithm.
- Information about the continency space and policy used for each region.

This keyword also displays additional information when you use the execution region attribute `ANY_SIZE` in a scatter-loading file.

`architecture` Summarizes the image architecture by listing the CPU, FPU and byte order.

`common` Lists all common sections that are eliminated from the image. Using this option implies `--info=common,totals`.

`compression` Gives extra information about the RW compression process.

`debug` Lists all rejected input debug sections that are eliminated from the image as a result of using `--remove`. Using this option implies `--info=debug,totals`.

`exceptions` Gives information on exception table generation and optimization.

`inline` Lists all functions that are inlined by the linker, and the total number of inlines if `--inline` is used.

`inputs` Lists the input symbols, objects and libraries.

`libraries` Lists the full path name of every library automatically selected for the link stage.

You can use this option with `--info_lib_prefix` to display information about a specific library.

`merge` Lists the **`const`** strings that are merged by the linker. Each item lists the merged result, the strings being merged, and the associated object files.

`pltgot` Lists the PLT entries built for the executable or DLL.

`sizes` Lists the code and data (RO Data, RW Data, ZI Data, and Debug Data) sizes for each input object and library member in the image. Using this option implies `--info=sizes,totals`.

`stack` Lists the stack usage of all global symbols.

`summarysizes` Summarizes the code and data sizes of the image.

`summarystack` Summarizes the stack usage of all global symbols.

`tailreorder` Lists all the tail calling sections that are moved above their targets, as a result of using `--tailreorder`.

`totals` Lists the totals of the code and data (RO Data, RW Data, ZI Data, and Debug Data) sizes for input objects and libraries.

`unused` Lists all unused sections that are eliminated from the image as a result of using `--remove`.

`unusedsymbols`

Lists all symbols that have been removed by unused section elimination.

`veneers` Lists the linker-generated veneers.

`veneercallers`

Lists the linker-generated veneers with additional information about the callers to each veneer. Use with `--verbose` to list each call individually.

`veneerpools` Displays information on how the linker has placed veneer pools.

`visibility` Lists the symbol visibility information. You can use this option with either `--info=inputs` or `--verbose` to enhance the output.

The output from `--info=sizes,totals` always includes the padding values in the totals for input objects and libraries.

If you are using RW data compression (the default), or if you have specified a compressor using the `--datacompressor=`*id* option, the output from `--info=sizes,totals` includes an entry under `Grand Totals` to reflect the true size of the image.

Note

Spaces are not permitted between topic keywords in the list. For example, you can enter `--info=sizes,totals` but not `--info=sizes, totals`.

See also

- *--any_placement=algorithm* on page 2-8
- *--any_sort_order=order* on page 2-10
- *--datacompressor=opt* on page 2-25
- *--info_lib_prefix=opt*
- *--inline, --no_inline* on page 2-47
- *--merge, --no_merge* on page 2-61
- *--remove, --no_remove* on page 2-74
- *--tailreorder, --no_tailreorder* on page 2-91
- *--verbose* on page 2-98
- *RW data compression* on page 3-20 in the *Linker User Guide*
- *Getting information about images* on page 3-41 in the *Linker User Guide*
- *Placing unassigned sections with the .ANY module selector* on page 5-26 in the *Linker User Guide*

2.1.69 `--info_lib_prefix=`*opt*

This option is a filter for the `--info=libraries` option. The linker only displays the libraries that have the same prefix as the filter.

Syntax

`armlink --info=libraries --info_lib_prefix=`*`opt`*

Where *opt* is the prefix of the required library.

Example

- Displaying a list of libraries without the filter:

  `armlink --info=libraries test.o`

  Produces a list of libraries, for example:

  ```
  install_directory\...\lib\armlib\c_4.l
  install_directory\...\lib\armlib\fz_4s.l
  install_directory\...\lib\armlib\h_4.l
  install_directory\...\lib\armlib\m_4s.l
  install_directory\...\lib\armlib\vfpsupport.l
  ```

- Displaying a list of libraries with the filter:

  `armlink --info=libraries --info_lib_prefix=c test.o`

  Produces a list of libraries with the specified prefix, for example:

  ```
  install_directory\...\lib\armlib\c_4.l
  ```

2.1.70 `--init=`*`symbol`*

This option specifies the symbol name that is used to define initialization code. The dynamic linker executes this code when it loads the executable file or shared object.

See also

- Chapter 4 *BPABI and SysV Shared Libraries and Executables*.

2.1.71 `--inline, --no_inline`

This option enables or disables branch inlining to optimize small function calls in your image.

Default

The default is `--no_inline`.

Note

This branch optimization is off by default because enabling it changes the image such that debug information might be incorrect. If enabled, the linker makes no attempt to correct the debug information.

See also

- *--branchnop, --no_branchnop* on page 2-17
- *--tailreorder, --no_tailreorder* on page 2-91
- *Inlining* on page 3-26 in the *Linker User Guide*.

2.1.72 `--inlineveneer, --no_inlineveneer`

This option enables or disables the generation of inline veneers to give greater control over how the linker places sections.

Default

The default is `--inlineveneer`.

See also

- *--piveneer, --no_piveneer* on page 2-64
- *--veneershare, --no_veneershare* on page 2-97
- *Veneers* on page 3-23 in the *Linker User Guide*.

2.1.73 *input_file_list*

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

Usage

The linker sorts through the input file list in order. If the linker is unable to resolve input file problems then a diagnostic message is produced.

The symdefs files can be included in the list to provide global symbol values for previously generated image files.

You can use libraries in the input file list in the following ways:

- Specify a library to be added to the list of libraries that is used to extract members if they resolve any non weak unresolved references. For example, specify `mystring.lib` in the input file list.

Note

Members from the libraries in this list are added to the image only when they resolve an unresolved non weak reference.

- Specify particular members to be extracted from a library and added to the image as individual objects. Members are selected from a comma separated list of patterns that can include wild characters. Spaces are allowed but if you use them you must enclose the whole input file list in quotes.

  The following shows an example of an input file list both with and without spaces:

  `mystring.lib(strcmp.o,std*.o)`

  `"mystring.lib(strcmp.o, std*.o)"`

The linker automatically searches the appropriate C and C++ libraries in order to select the best standard functions for your image. You can use `--no_scanlib` to prevent automatic searching of the standard libraries.

The linker processes the input file list in the following order:

1. Objects are added to the image unconditionally.
2. Members selected from libraries using patterns are added to the image unconditionally, as if they are objects. For example, to add all `a*.o` objects and `stdio.o` from `mystring.lib` use the following:

   `"mystring.lib(stdio.o, a*.o)"`
3. The standard C or C++ libraries are added to the list of libraries that are later used to resolve any remaining references.

See also

- *--scanlib, --no_scanlib* on page 2-77
- *Library searching, selection, and scanning* on page 3-44 in the Linker User Guide
- *Accessing symbols in another image* on page 4-12 in the Linker User Guide.

2.1.74 `--keep=section_id`

This option specifies input sections that must not be removed by unused section elimination.

Syntax

`--keep=section_id`

Where *section_id* is one of the following:

symbol
: Specifies that an input section defining *symbol* is to be retained during unused section elimination. If multiple definitions of *symbol* exist, armlink generates an error message.

  For example, you might use --keep=int_handler.

  To keep all sections that define a symbol ending in _handler, use --keep=*_handler.

object(*section*)
: Specifies that *section* from *object* is to be retained during unused section elimination. For example, to keep the vect section from the vectors.o object use: --keep=vectors.o(vect)

  To keep all sections from the vectors.o object where the first three characters of the name of the section are vec, use: --keep=vectors.o(vec*)

object
: Specifies that the single input section from *object* is to be retained during unused section elimination. If you use this short form and there is more than one input section in *object*, the linker generates an error message.

  For example, you might use --keep=dspdata.o.

  To keep the single input section from each of the objects that has a name starting with dsp, use --keep=dsp*.o.

All forms of the *section_id* argument can contain the * and ? wild characters. Matching is case-insensitive, even on hosts with case-sensitive file naming. For example:

- --keep foo.o(Premier*) causes the entire match for Premier* to be case-insensitive
- --keep foo.o(Premier) causes a case-sensitive match for the string Premier.

Use *.o to match all object files. Use * to match all object files and libraries.

You can specify multiple --keep options on the command line.

Matching a symbol that has the same name as an object

If you name a symbol with the same name as an object, then --keep=*symbol_id* searches for a symbol that matches *symbol_id*:

- If a symbol is found, it matches the symbol.
- If no symbol is found, it matches the object.

You can force --keep to match an object with --keep=*symbol_id*(). Therefore, to keep both the symbol and the object, specify --keep foo.o --keep foo.o().

See also

- *The image structure* on page 3-2 in the *Linker User Guide*.

2.1.75 --keep_protected_symbols

Use this option to explicitly keep STV_PROTECTED symbols even if you are not using dynamic linking.

For example, your application might export functions provided by an API to shared objects that are loaded using a custom loader. However, the linker unused section elimination optimization causes the sections to be removed, even if those sections include STV_PROTECTED symbols. To prevent section containing STV_PROTECTED symbols from being removed, specify `--keep_protected_symbols`.

See also

- *--dll* on page 2-30
- *--max_visibility=type* on page 2-61
- *--override_visibility* on page 2-63
- *--shared* on page 2-80
- *Automatic dynamic symbol table rules in SysV models* on page 4-8
- *Automatic dynamic symbol table rules in the BPABI DLL-like model* on page 4-13
- *Unused section elimination* on page 3-14 in the *Linker User Guide*.

2.1.76 --largeregions, --no_largeregions

This option controls the sorting order of sections in large execution regions to minimize the distance between sections that call each other.

Usage

If the execution region contains more code than the range of a branch instruction then the linker switches to large region mode. In this mode the linker sorts according to the approximated average call depth of each section in ascending order. The linker might also place distribute veneers amongst the code sections to minimize the number of veneers.

Note

Large region mode can result in large changes to the layout of an image even when small changes are made to the input.

To disable large region mode and revert to lexical order, use `--no_largeregions`. Section placement is then predictable and image comparisons are more predictable. However some branches might not reach the target causing the link step to fail. If this happens you must place code/data sections explicitly using an appropriate scatter-loading description file or write your own veneer.

Default

The default option is `--no_largeregions`. However, if at least one execution region contains more code than the smallest inter-section branch then the default option is `--largeregions`. The smallest inter-section branch depends on the code in the region and the target processor:

32Mb Execution region contains only ARM.

16Mb Execution region contains Thumb, Thumb-2 is supported.

4Mb Execution region contains Thumb, no Thumb-2 support.

See also

- *--sort=algorithm* on page 2-81
- *Veneers* on page 3-23 in the *Linker User Guide*.

2.1.77 `--last=`*`section_id`*

This option places the selected input section last in its execution region. For example, this can force an input section that contains a checksum to be placed last in the RW section.

Syntax

`--last=`*`section_id`*

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

`symbol` Selects the section that defines *`symbol`*. You must not specify a symbol that has more than one definition because only a single section can be placed last. For example: `--last=checksum`

`object``(`*`section`*`)`

Selects the *`section`* from *`object`*. There must be no space between *`object`* and the following open parenthesis. For example: `--last=checksum.o(check)`

`object` Selects the single input section from *`object`*. If there is more than one input section in *`object`*, `armlink` generates an error message.

Usage

The `--last` option cannot be used with `--scatter`. Instead, use the `+LAST` attribute in a scatter-loading file.

See also

- *Section placement* on page 3-10 in the *Linker User Guide*.

2.1.78 `--legacyalign, --no_legacyalign`

By default, the linker assumes execution regions and load regions to be four-byte aligned. This option enables the linker to minimize the amount of padding that it inserts into the image.

The `--no_legacyalign` option instructs the linker to insert padding to force natural alignment. Natural alignment is the highest known alignment for that region.

Use this option to ensure strict conformance with the ELF specification.

Default

The default is `--legacyalign`.

See also

- *Section placement* on page 3-10 in the *Linker User Guide*.

2.1.79 `--libpath=`*`pathlist`*

This option specifies a list of paths that are used to search for the ARM standard C and C++ libraries.

The default path for the parent directory containing the ARM libraries is specified by the `RVCT`*`ver`*`LIB` environment variable, where *`ver`* is the version of the compilation tools installed. For example,`ARMCC41LIB`. Any paths specified here override the path specified by the environment variable.

Syntax

`--libpath=`*`pathlist`*

Where *`pathlist`* is a comma-separated list of paths that are only used to search for required ARM libraries. Do not include spaces between the comma and the path name when specifying multiple path names, for example, *`path1,path2,path3,...,pathn`*.

Note

This option does not affect searches for user libraries. Use `--userlibpath` instead for user libraries.

See also

- *--userlibpath=pathlist* on page 2-95
- *Library searching, selection, and scanning* on page 3-44 in the *Linker User Guide*.

2.1.80 `--library=`*name*

This option enables the linker to search either a dynamic library, `lib`*name*`.so`, or a static library, `lib`*name*`.a`, depending on whether dynamic library searching is enabled at that point on the command line:

- if dynamic linking is enabled, the linker dynamically links with the library, `lib`*name*`.so`
- if dynamic linking is disabled it links with the static library, `lib`*name*`.a`.

Dynamic linking is enabled by default. Use the `--[no_]search_dynamic_libraries` option to control the searching of dynamic or static libraries.

Usage

The `--library` option enables you to link against a library without specifying the full library filename on the command-line.

If you specify the `--[no_]search_dynamic_libraries` option, it applies to the following `--library` options up until the next `--[no_]search_dynamic_libraries` option.

References to the shared library are added to the image and resolved to the library by the dynamic loader at runtime. The order in which references are resolved to libraries is the order in which libraries are specified on the command line. This is also the order in which the dependencies are resolved by the dynamic linker. You can specify the runtime location of libraries using the `--runpath` option.

Example

Example 2-2 on page 2-55 searches for `libfoo.so` before `libfoo.a`, but only searches for `libbar.a`.

Example 2-2 Searching dynamic libraries

Use armlink with the following command-line options:

```
--arm_linux --shared --fpic --search_dynamic_libraries --library=foo
--no_search_dynamic_libraries --library=bar
```

See also

- *--arm_linux* on page 2-11
- *--runpath=pathlist* on page 2-76
- *--search_dynamic_libraries, --no_search_dynamic_libraries* on page 2-78.

2.1.81 --library_type=*lib*

This option selects the library to be used at link time.

Note

This option can be used with the compiler, assembler or linker.

Use this option with the linker to override all other --library_type options.

Syntax

--library_type=*lib*

Where *lib* can be one of:

standardlib	Specifies that the full RealView Compilation Tools runtime libraries are selected at link time.
microlib	Specifies that the *C micro-library* (microlib) is selected at link time.

Default

If you do not specify --library_type at link time and no object file specifies a preference, then the linker assumes --library_type=standardlib.

See also

- *Building an application with microlib* on page 3-4 in the Libraries and Floating Point Support Guide.

2.1.82 --licretry

If you are using floating licenses, this option makes up to 10 attempts to obtain a license when you invoke armlink.

Usage

Use this option if your builds are failing to obtain a license from your license server, and only after you have ruled out any other problems with the network or the license server setup.

It is recommended that you place this option in the RVCT40_LINKOPT environment variable. In this way, you do not have to modify your build files.

See also

- *FLEXnet for ARM Tools License Management Guide*.

2.1.83 --linux_abitag=*version_id*

This option enables you to specify the minimum compatible Linux kernel version for the executable file you are building. This is then stored in the output ELF so it can be checked when running the executable on the target.

The information you specify with --linux_abitag is written into a section called .note.ABI-tag. If there is no information, the linker does not produce a .note.ABI-tag section in the output ELF file.

See also

- Chapter 4 *BPABI and SysV Shared Libraries and Executables*.

2.1.84 --list=*file*

This option redirects the diagnostics from output of the --info, --map, --symbols, --verbose, --xref, --xreffrom, and --xrefto commands to *file*.

The specified file is created when diagnostics are output. If a file of the same name already exists, it is overwritten. However, if diagnostics are not output, a file is not created. In this case, the contents of any existing file with the same name remain unchanged.

If *file* is specified without a path, it is created in the output directory, that is, the directory where the output image is being written.

See also

- *--info=topic[,topic,...]* on page 2-44
- *--map, --no_map* on page 2-59
- *--symbols, --no_symbols* on page 2-89
- *--verbose* on page 2-98
- *--xref, --no_xref* on page 2-101
- *--xrefdbg, --no_xrefdbg* on page 2-101.

2.1.85 --list_mapping_symbols, --no_list_mapping_symbols

This option enables or disables the addition of mapping symbols `$a`, `$d`, `$t`, and `$t.x` in the output produced by `--symbols`.

Mapping symbols are used to flag transitions between ARM code, Thumb code, and data.

Default

The default is `--no_list_mapping_symbols`.

See also

- *--symbols, --no_symbols* on page 2-89
- *Using the Linker*:
  - *About mapping symbols* on page 4-2
- *ELF for the ARM Architecture* (AAELF).

2.1.86 --load_addr_map_info, --no_load_addr_map_info

This option includes load addresses for execution regions in the map file.

If an input section is compressed, then the load address has no meaning and `COMPRESSED` is displayed instead.

For sections that do not have a load address, such as ZI data, the load address is blank

Default

The default is `--no_load_addr_map_info`.

Restrictions

You must use the `--map` with this option.

Example

The following example shows the format of the map file output:

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

0x00008000   0x00008000   0x00000008   Code   RO          25  * !!!main            __main.o(c_4.l)
0x00010000   COMPRESSED   0x00001000   Data   RW           2    dataA              data.o
0x00003000        -       0x00000004   Zero   RW           2    .bss               test.o
```

See also

- *--map, --no_map* on page 2-59.

2.1.87 `--locals, --no_locals`

The `--locals` option adds local symbols in the output symbol table.

The effect of the `--no_locals` option is different for images and object files.

When producing an executable image `--no_locals` removes local symbols from the output symbol table.

For object files built with the `--partial` option, the `--no_locals` option:

- Keeps mapping symbols and build attributes in the symbol table.
- Removes those local symbols that can be removed without loss of functionality.

  Symbols that cannot be removed, such as the targets for relocations, are kept. For these symbols, the names are removed. These are marked as `[Anonymous Symbol]` in the `fromelf --text` output.

`--no_locals` is a useful optimization if you want to reduce the size of the output symbol table in the final image.

Default

The default is `--locals`.

See also

- *--privacy* on page 2-68
- *--privacy* on page 2-45 in the *Utilities Guide*
- *--strip=option[,option,...]* on page 2-53 in the *Utilities Guide*.

2.1.88 --ltcg

This option enables link-time code generation optimizations. You must use this option if any of your input objects have been compiled with --ltcg.

You can also use this option with the Profiler-guided optimization option, --profile.

See also

- *--profile=filename* on page 2-69
- *--ltcg* on page 2-88 in the *Compiler Reference Guide*.
- *--profile=filename* on page 2-107 in the *Compiler Reference Guide*.

2.1.89 --mangled, --unmangled

This option instructs the linker to display mangled or unmangled C++ symbol names in diagnostic messages, and in listings produced by the --xref, --xreffrom, --xrefto, and --symbols options.

Default

The default is --unmangled.

Usage

If --unmangled is selected, C++ symbol names are displayed as they appear in your source code.

If --mangled is selected, C++ symbol names are displayed as they appear in the object symbol tables.

See also

- *--symbols, --no_symbols* 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.

2.1.90 --map, --no_map

This option enables or disables the printing of a memory map.

The map contains the address and the size of each load region, execution region, and input section in the image, including linker-generated input sections. This can be output to a text file using --list=*file*.

Default

The default is `--no_map`.

See also

- *--list=file* on page 2-56
- *--load_addr_map_info, --no_load_addr_map_info* on page 2-57
- *--section_index_display=type* on page 2-79.

2.1.91 `--match=crossmangled`

This option instructs the linker to match the following combinations together:

- a reference to an unmangled symbol with the mangled definition
- a reference to a mangled symbol with the unmangled definition.

Libraries and matching combinations operate as follows:

- If the library members define a mangled definition, and there is an unresolved unmangled reference, the member is loaded to satisfy it.
- If the library members define an unmangled definition, and there is an unresolved mangled reference, the member is loaded to satisfy it.

Note

This option has no effect if used with partial linking. The partial object contains all the unresolved references to unmangled symbols, even if the mangled definition exists. Matching is done only in the final link step.

2.1.92 `--max_veneer_passess=`*`value`*

This option specifies a limit to the number of veneer generation passes the linker attempts to make when both the following conditions are met:

- a Section that is sufficiently large has a relocation that requires a veneer
- the linker cannot place the veneer close enough to the call site.

The linker attempts to diagnose the failure if the maximum number of veneer generation passes you specify is exceeded, and displays a warning message. You can downgrade this warning message using `--diag_remark`.

Syntax

`--max_veneer_passes=`*`value`*

Where *value* is the maximum number of veneer passes the linker is to attempt. The minimum value you can specify is one.

Default

The default number of passes is 10.

See also

- *--diag_remark=tag[,tag,...]* on page 2-28
- *--diag_warning=tag[,tag,...]* on page 2-29.

2.1.93 --max_visibility=*type*

This option controls the visibility of all symbol definitions.

Syntax

`--max_visibility=`*type*

Where *type* can be one of:

`default`	Default visibility.
`protected`	Protected visibility.

Usage

Use `--max_visibility=protected` to limit the visibility of all symbol definitions. Global symbol definitions that normally have default visibility, are given protected visibility when this option is specified.

Default

The default is `--max_visibility=default`.

See also

- *--keep_protected_symbols* on page 2-51
- *--override_visibility* on page 2-63.

2.1.94 --merge, --no_merge

This option enable or disables the merging of **`const`** strings that are placed in shareable sections by the compiler. Using `--merge` can reduce the size of the image if there are similarities between **`const`** strings.

For a listing of the merged **const** strings you can use --info=merge.

Default

The default is --merge.

By default, merging happens between different load and execution regions. Therefore, code from one execution or load region might use a string stored in different region. If you do not want this behavior, then do one of the following:

- use the PROTECTED load region attribute if you are using scatter-loading
- globally disable merging with --no_merge.

See also

- *--info=topic[,topic,...]* on page 2-44.

2.1.95 --muldefweak, --no_muldefweak

This option enables or disables multiple weak definitions of a symbol.

If enabled, the linker chooses the first definition that it encounters and discards all the other duplicate definitions. If disabled, the linker generates an error message for all multiply defined weak symbols.

Default

The default is --no_muldefweak.

When --arm_linux is used, --muldefweak is the default.

See also

- *--arm_linux* on page 2-11.

2.1.96 --output=*file*

This option specifies the name of the output file. The file can be either a partially-linked object or an executable image, depending on the command-line options used.

Syntax

--output=*file*

If `--output=file` is not specified, the linker uses the following default filenames:

`__image.axf` if the output is an executable image

`__object.o` if the output is a partially-linked object.

If *file* is specified without path information, it is created in the current working directory. If path information is specified, then that directory becomes the default output directory.

See also

- *--callgraph_file=filename* on page 2-19
- *--partial* on page 2-64.

2.1.97 --override_visibility

This option enables `EXPORT` and `IMPORT` directives in a steering file to override the visibility of a symbol.

By default:

- only symbol definitions with `STV_DEFAULT` or `STV_PROTECTED` visibility can be exported
- only symbol references with `STV_DEFAULT` visibility can be imported.

When you specify `--override_visibility`, any global symbol definition can be exported and any global symbol reference can be imported.

See also

- *--keep_protected_symbols* on page 2-51
- *--undefined_and_export=symbol* on page 2-93
- *EXPORT* on page 2-103
- *IMPORT* on page 2-106.

2.1.98 --pad=*num*

This option enables you to set a value for padding bytes. The linker assigns this value to all padding bytes inserted in load or execution regions.

Syntax

`--pad=num`

Where *num* is an integer, which can be given in hexadecimal format. For example, setting *num* to `0xFF` might help to speed up ROM programming time. If *num* is greater than `0xFF`, then the padding byte is set to `(char)`*num*.

Note

Padding is only inserted:

- Within load regions. No padding is present *between* load regions.
- Between fixed execution regions (in addition to forcing alignment). Padding is not inserted up to the maximum length of a load region unless it has a fixed execution region at the top.
- Between sections to ensure that they conform to alignment constraints.

2.1.99 `--partial`

This option creates a partially-linked object that can be used in a subsequent link step.

See also

- *Partial linking model* on page 2-4 in the *Linker User Guide*.

2.1.100 `--piveneer, --no_piveneer`

This option enables or disables the generation of a veneer for a call from PI code to absolute code. When using `--no_piveneer`, an error message is produced if the linker detects a call from PI code to absolute code.

Default

The default is `--piveneer`.

See also

- *--inlineveneer, --no_inlineveneer* on page 2-48
- *--veneershare, --no_veneershare* on page 2-97
- *Veneers* on page 3-23 in the *Linker User Guide*.

2.1.101 --pltgot=*type*

This option specifies the type of *Procedure Linkage Table* (PLT) and *Global Offset Table* (GOT) to use, corresponding to the different addressing modes of the *Base Platform Application Binary Interface* (BPABI).

Note

This option is supported only when using `--base_platform` or `--bpabi`.

Syntax

`--pltgot=`*type*

Where *type* is one of the following:

`none` References to imported symbols are added as dynamic relocations for processing by a platform specific post-linker.

`direct` References to imported symbols are resolved to read-only pointers to the imported symbols. These are direct pointer references.

Use this type to turn on PLT generation when using `--base_platform`.

`indirect` The linker creates a GOT and possibly a PLT entry for the imported symbol. The reference refers to PLT or GOT entry.

This type is not supported if you have multiple load regions.

`sbrel` Same referencing as indirect with one exception. GOT entries are stored as offsets from the static base address for the segment held in R9 at runtime.

This type is not supported if you have multiple load regions.

Default

When the options `--bpabi` or `--dll` are used, the default is `--pltgot=direct`.

When the `--base_platform` option is used, the default is `--pltgot=none`.

See also

- *--base_platform* on page 2-14
- *--bpabi* on page 2-17
- *--dll* on page 2-30
- *--pltgot_opts=mode* on page 2-66

- *Linker User Guide*:
  - *Base Platform Application Binary Interface (BPABI) linking model* on page 2-6
  - *Base Platform linking model* on page 2-7.

2.1.102 `--pltgot_opts=mode`

This option enables or disables weak references when generating *Procedure Linkage Table* (PLT) entries.

Syntax

`--pltgot_opts=mode`

Where *mode* is one of the following:

`crosslr` Calls to and from a load region marked `RELOC` go by way of the PLT.

`noweakrefs` Generates a NOP, for a function call, or zero, for data. No PLT entry is generated. Weak references to imported symbols remain unresolved.

`weakrefs` Weak references produce a PLT entry. These references must be resolved at a later link stage.

Default

The default is `--pltgot_opts=noweakrefs`.

See also

- *--base_platform* on page 2-14
- *--pltgot=type* on page 2-65.

2.1.103 `--predefine="string"`

This option enables commands to be passed to the pre-processor specified on the first line of the scatter file. You can also use the synonym:`--pd="string"`.

Syntax

`--predefine="string"`

More than one `--predefine` option can be used on the command-line.

Use this option with `--scatter`.

Example

Example 2-3 shows how the scatter file looks before pre-processing.

Example 2-3 Scatter file before pre-processing

```
Scatter file:
#! armcc -E
lr1 BASE
{
    er1 BASE
    {
        *(+RO)
    }
    er2 BASE2
    {
        *(+RW+ZI)
    }
}
```

Use `armlink` with the command-line options:

`--predefine="-DBASE=0x8000" --predefine="-DBASE2=0x1000000" --scatter=`*`file`*

This passes the command-line options: `-DBASE=0x8000 -DBASE2=0x1000000` to the compiler to pre-process the scatter file.

Example 2-4 shows how the scatter file looks after pre-processing:

Example 2-4 Scatter file after pre-processing

```
lr1 0x8000
{
    er1 0x8000
    {
        *(+RO)
    }
    er2 0x1000000
    {
        *(+RW+ZI)
    }
}
```

2.1.104 --prelink_support, --no_prelink_support

This option enables or disables the linker addition of:

- an extra empty program header table entry to an application
- some extra DT_NULL dynamic tags to both applications and shared libraries.

The prelink tool uses this reserved space to write extra information that is needed by the dynamic loader.

The --prelink_support option only has an effect when the --sysv option is selected. Building for ARM Linux with the --arm_linux command line option turns on several command line options that make the linker behave like GNU ld, and includes --sysv.

Use --no_prelink_support to force the linker not to reserve the extra space when building for ARM Linux.

Default

The default is --prelink_support when --sysv is specified.

See also

- *--arm_linux* on page 2-11
- *--sysv* on page 2-90.

2.1.105 --privacy

The effect of this option is different for images and object files.

When producing an executable image it removes local symbols from the output symbol table.

For object files built with the --partial option, this option:

- Changes section names to a default value, for example, changes code section names to .text.
- Keeps mapping symbols and build attributes in the symbol table.
- Removes those local symbols that can be removed without loss of functionality.

  Symbols that cannot be removed, such as the targets for relocations, are kept. For these symbols, the names are removed. These are marked as [Anonymous Symbol] in the fromelf --text output.

Note

To help protect your code in images and objects that are delivered to third parties, use the `fromelf --privacy` command.

See also

- *--locals, --no_locals* on page 2-58
- *--partial* on page 2-64
- *Protecting code in images and objects with fromelf* on page 2-2 in the *Utilities Guide*
- *--privacy* on page 2-45 in the *Utilities Guide*
- *--strip=option[,option,...]* on page 2-53 in the *Utilities Guide*.

2.1.106 `--profile=filename`

This option enables profile driven feedback to be passed back to the compiler.

Usage

When using link time code generation (`--ltcg`) you can pass the profiler file *`filename`* back to the compiler to enable it to use profile driven feedback.

See also

- *--ltcg* on page 2-59
- *--ltcg* on page 2-88 in the *Compiler Reference Guide*
- *--profile=filename* on page 2-107 in the *Compiler Reference Guide*.

2.1.107 `--project=filename, --no_project`

This option instructs the linker to load the specified project template file.

Syntax

```
--no_project
--project=filename
```

Where *`filename`* is the name of a project template file.

Note

To use *`filename`* as a default project file, set the `RVDS_PROJECT` environment variable to *`filename`*.

`--no_project` prevents the default project template file specified by the environment variable `RVDS_PROJECT` from being used.

Default

The default is `--no_project`.

Restrictions

Options from a project template file are only set when they do not conflict with options already set on the command line. If an option from a project template file conflicts with an existing command-line option, the command-line option takes precedence.

Example

Consider the following project template file:

```
<!-- suiteconf.cfg -->
<suiteconf name="Platform Baseboard for ARM926EJ-S">
    <tool name="armlink">
        <cmdline>
            --cpu=ARM926EJ-S
            --fpu=vfpv2
        </cmdline>
    </tool>
</suiteconf>
```

When the `RVDS_PROJECT` environment variable is set to point to this file, the command:

```
armlink foo.o
```

results in an actual command line of:

```
armlink --cpu=ARM926EJ-S --fpu=VFPv2 foo.o
```

See also

- *--reinitialize_workdir* on page 2-72
- *--workdir=directory* on page 2-100.

2.1.108 `--reduce_paths, --no_reduce_paths`

This option enables or disables the elimination of redundant path name information in file paths.

Mode

Effective on Windows systems only.

Default

The default is `--no_reduce_paths`.

Usage

Windows systems impose a 260 character limit on file paths. Where path names exist whose absolute names expand to longer than 260 characters, you can use the `--reduce_paths` option to reduce absolute path name length by matching up directories with corresponding instances of `..` and eliminating the directory/`..` sequences in pairs.

Note

It is recommended that you avoid using long and deeply nested file paths, in preference to minimizing path lengths using the `--reduce_paths` option.

Example

Linking the file:

```
..\..\..\xyzzy\xyzzy\objects\file.c
```

from the directory:

```
\foo\bar\baz\gazonk\quux\bop
```

results in an actual path of:

```
\foo\bar\baz\gazonk\quux\bop\..\..\..\xyzzy\xyzzy\objects\file.o
```

Linking the same file from the same directory using the option `--reduce_paths` results in an actual path of:

```
\foo\bar\baz\xyzzy\xyzzy\objects\file.c
```

2.1.109 `--ref_cpp_init, --no_ref_cpp_init`

This option enables or disables the linker adding a reference to the C++ static object initialization routine in the RVCT libraries. The default reference added is `__cpp_initialize__aeabi_`. To change this you can use `--cppinit`.

Usage

Use `--no_ref_cpp_init` if you are not going to use the RVCT libraries. For example, if you are building an ARM Linux application.

Default

The default is `--ref_cpp_init`.

See also

- *--cppinit, --no_cppinit* on page 2-23.

2.1.110 `--reinitialize_workdir`

This option enables you to reinitialize the project template working directory set using `--workdir`.

When the directory set using `--workdir` refers to an existing working directory containing modified project template files, specifying this option causes the working directory to be deleted and recreated with new copies of the original project template files.

Restrictions

This option must be used in combination with the `--workdir` option.

See also

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

2.1.111 `--reloc`

This option creates a single relocatable load region with contiguous execution regions.

Usage

Only use this option for legacy systems with the type of relocatable ELF images that conform to the *ELF for the ARM Architecture* specification. The generated image might not be compliant with the ELF for the ARM Architecture specification.

When relocated MOVT and MOVW instructions are encountered in an image being linked with --reloc, armlink produces the following additional dynamic tags:

DT_RELA The address of a relocation table.

DT_RELASZ

The total size, in bytes, of the DT_RELA relocation table.

DT_RELAENT

The size, in bytes, of the DT_RELA relocation entry.

Note

For new systems, consider using images that conform to the *Base Platform Application Binary Interface* (BPABI).

See also

- *Type 1, one load region and contiguous execution regions* on page 5-48 in the *Linker User Guide*
- *Type 3, two load regions and non-contiguous execution regions* on page 5-52 in the *Linker User Guide*
- *Base Platform ABI for the ARM Architecture*
- *ELF for the ARM Architecture*.

2.1.112 --remarks

This option forces the linker to display remarks that are otherwise hidden by default when used with the --diag_remarks option.

Note

The linker does not issue remarks by default.

See also

- *--diag_remark=tag[,tag,...]* on page 2-28
- *--errors=file* on page 2-34.

2.1.113 --remove, --no_remove

This option enables or disables the removal of unused input sections from the image. An input section is considered used if it contains the image entry point, or if it is referred to from a used section.

Default

The default is --remove. However, if you also specify the --bpabi and --sysv options, the default is --no_remove.

Usage

By default, unused section elimination is disabled when building *dynamically linked libraries* (DLLs) or shared objects, Use --remove to re-enable unused section elimination.

Use --no_remove when debugging to retain all input sections in the final image even if they are unused.

Use --remove with the --keep option to retain specific sections in a normal build.

See also

- *--keep=section_id* on page 2-49
- *Section elimination* on page 3-13 in the *Linker User Guide*.

2.1.114 --ro_base=*address*

This option sets both the load and execution addresses of the region containing the RO output section at a specified address.

Syntax

--ro_base=*address*

Where *address* must be word-aligned.

Default

If this option is not specified, and no scatter-loading file is specified, the default is --ro_base=0x8000.

Restrictions

You cannot use `--ro_base` with `--scatter`, `--shared`, or `--sysv`.

See also

- *--ropi*
- *--rosplit*
- *--rw_base=address* on page 2-76
- *--rwpi* on page 2-77.
- *--scatter=file* on page 2-78.

2.1.115 `--ropi`

This option makes the load and execution region containing the RO output section position-independent. If this option is not used, the region is marked as absolute. Usually each read-only input section must be *Read-Only Position-Independent* (ROPI). If this option is selected, the linker:

- checks that relocations between sections are valid
- ensures that any code generated by the linker itself, such as interworking veneers, is ROPI.

Note

The linker gives a downgradeable error if `--ropi` is used without `--rwpi` or `--rw_base`.

Restrictions

You cannot use `--ropi` with `--scatter`, `--shared`, or `--sysv`.

See also

- *--ro_base=address* on page 2-74
- *--rosplit*
- *--rw_base=address* on page 2-76
- *--rwpi* on page 2-77.

2.1.116 `--rosplit`

This option splits the default RO load region into two RO output sections, one for `RO-CODE` and one for `RO-DATA`.

Restrictions

You cannot use `--rosplit` with `--scatter`, `--shared`, or `--sysv`.

See also

- *--ro_base=address* on page 2-74
- *--ropi* on page 2-75
- *--rw_base=address*
- *--rwpi* on page 2-77.

2.1.117 `--runpath=`*`pathlist`*

This option specifies a list of paths to be added to the search paths. The Linux dynamic linker uses these paths to locate the required Shared Objects.

Syntax

`--runpath=`*`pathlist`*

Where *`pathlist`* is a comma-separated list of paths. Do not include spaces between the comma and the path name when specifying multiple path names, for example, *`path1,path2,path3,...,pathn`*.

2.1.118 `--rw_base=`*`address`*

This option sets the execution addresses of the region containing the RW output section at a specified address.

Syntax

`--rw_base=`*`address`*

Where *`address`* must be word-aligned.

Restrictions

You cannot use `--rw_base` with `--scatter`, `--shared`, or `--sysv`.

See also

- *--ro_base=address* on page 2-74
- *--ropi* on page 2-75
- *--rosplit* on page 2-75
- *--rwpi* on page 2-77

- *--split* on page 2-83.

2.1.119 --rwpi

This option makes the load and execution region containing the RW and ZI output section position-independent. If this option is not used the region is marked as absolute. This option requires a value for --rw-base. If --rw-base is not specified, --rw-base=0 is assumed. Usually each writable input section must be read-write position-independent (RWPI).

If this option is selected, the linker:

- checks that the PI attribute is set on input sections to any read-write execution regions
- checks that relocations between sections are valid
- generates entries relative to the static base in the table Region$$Table.

  This is used when regions are copied, decompressed, or initialized.

Restrictions

You cannot use --rwpi with --scatter, --shared, or --sysv.

See also

- *--ro_base=address* on page 2-74
- *--ropi* on page 2-75
- *--rosplit* on page 2-75
- *--rw_base=address* on page 2-76.

2.1.120 --scanlib, --no_scanlib

This option enables or disables scanning of the default libraries to resolve references. Use --no_scanlib if you want to link your own libraries.

Default

The default is --scanlib.

2.1.121 --scatter=*file*

This option creates an image memory map using the scatter-loading description contained in the specified file. The description provides grouping and placement details of the various regions and sections in the image.

Syntax

`--scatter=`*file*

Where *file* is the name of a scatter file.

Usage

To modify the placement of any unassigned input sections when `.ANY` selectors are present, use the following command-line options with `--scatter`:

- `--any_contingency`
- `--any_placement`
- `--any_sort_order`
- `--tiebreaker`.

The `--scatter` option cannot be used with `--bpabi`, `--dll`, `--first`, `--last`, `--partial`, `--reloc`, `--ro-base`, `--ropi`, `--rosplit`, `--rw-base`, `--rwpi`, `--split`, `--shared`, `--startup`, and `--sysv`.

See also

- *--any_contingency* on page 2-8
- *--any_placement=algorithm* on page 2-8
- *--any_sort_order=order* on page 2-10
- *--tiebreaker=option* on page 2-92
- Chapter 5 *Using Scatter-loading Description Files* in the *Linker User Guide*.

2.1.122 --search_dynamic_libraries, --no_search_dynamic_libraries

This option enables or disables the searching of dynamic libraries as specified by the `--library` option.

Usage

The `--search_dynamic_libraries` setting applies to any following `--library` options until a `--no_search_dynamic_libraries` option appears on the command line. For libraries specified with `--library`:

- libraries following `--search_dynamic_libraries` use the dynamic version, `.so`
- libraries following `--no_search_dynamic_libraries` use the static version, `.a`.

Default

The default is `--search_dynamic_libraries`.

See also

- *--arm_linux* on page 2-11
- *--library=name* on page 2-54.

2.1.123 `--section_index_display=`*type*

This option changes the display of the index column when printing memory map output. Use this option with `--map`.

Syntax

`--section_index_display=`*type*

Where *type* is one of the following:

`cmdline` Alters the display of the map file to show the order that a section appears on the command-line. The command-line order is defined as `File.Object.Section` where:

- `Section` is the section index, `sh_idx`, of the `Section` in the `Object`
- `Object` is the order that `Object` appears in the `File`
- `File` is the order the `File` appears on the command line.

The order the `Object` appears in the `File` is only significant if the file is an `ar` archive.

`internal` The index value represents the order in which the linker creates the section.

`input` The index value represents the section index of the section in the original input file.

Usage

Use `--map` with `--section_index_display=input` when you want to find the exact section in an input object.

Default

The default is `--section_index_display=internal`.

See also

- *--map, --no_map* on page 2-59
- *--tiebreaker=option* on page 2-92.

2.1.124 `--shared`

This option creates a *System V* (SysV) shared object.

Usage

You must use this option with `--fpic` and `--sysv`.

Note

By default, this option disables unused section elimination. Use the `--remove` option to re-enable unused section elimination when building a shared object.

See also

- *--bpabi* on page 2-17
- *--dll* on page 2-30
- *--runpath=pathlist* on page 2-76
- *--soname=name* on page 2-81
- *--sysv* on page 2-90
- Chapter 4 *BPABI and SysV Shared Libraries and Executables*.

2.1.125 `--show_cmdline`

This option outputs the command-line used by the linker. It shows the command-line after processing by the linker, and can be useful to check:

- the command-line a build system is using

- how the linker is interpreting the supplied command-line, for example, the ordering of command line options.

The commands are shown normalized, and the contents of any via files are expanded.

The output is sent to the standard output stream (`stdout`).

See also

- *--help* on page 2-43
- *--vsn* on page 2-100.

2.1.126 `--soname=name`

This option specifies the shared object runtime name that is used as the dependency name by any object that links against this shared object. This dependency is stored in the resultant file.

See also

- Chapter 4 *BPABI and SysV Shared Libraries and Executables*.

2.1.127 `--sort=algorithm`

This option specifies the sorting algorithm to determine the order of sections in an execution region to minimize the distance between sections that call each other. The sorting algorithms conform to the standard rules placing input section in ascending order by attributes.

Sort algorithms can also be specified in a scatter file for individual execution regions using the `SORTTYPE` keyword.

Syntax

`--sort=algorithm`

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

`Alignment` Sorts input sections by ascending order of alignment value.

`AvgCallDepth` Sorts all Thumb code before ARM code and then sorts according to the approximated average call depth of each section in ascending order.

Use this algorithm to minimize the number of long branch veneers.

Note

The approximation of the average call depth depends on the order of input sections. Therefore, this sorting algorithm is more dependent on the order of input sections than using, say, `RunningDepth`.

`BreadthFirstCallTree`

This is similar to the `CallTree` algorithm except that it uses a breadth-first traversal when flattening the Call Tree into a list.

`CallTree` The linker flattens the call tree into a list containing the read-only code sections from all execution regions that have `CallTree` sorting enabled.

Sections in this list are copied back into their execution regions, followed by all the non read-only code sections, sorted lexically. Doing this ensures that sections calling each other are placed close together.

Note

This sorting algorithm is less dependent on the order of input sections than using either `RunningDepth` or `AvgCallDepth`.

`Lexical` Sorts according to the name of the section and then by input order if the names are the same.

`LexicalState` Sorts Thumb code before ARM code, then sorts lexically.

`List` Provides a list of the available sorting algorithms. The linker terminates after displaying the list.

`ObjectCode` Sorts code sections by tiebreaker. All other sections are sorted lexically. This is most useful when used with `--tiebreaker=cmdline` because it attempts to group all the sections from the same object together in the memory map.

`RunningDepth` Sorts all Thumb code before ARM code and then sorts according to the running depth of the section in ascending order. The running depth of a section S is the average call depth of all the sections that call S, weighted by the number of times that they call S.

Use this algorithm to minimize the number of long branch veneers.

Default

The default algorithm is `--sort=Lexical`. However, if at least one execution region exceeds the maximum branch size, the default algorithm is `--sort=AvgCallDepth`.

See also

- *--largeregions, --no_largeregions* on page 2-51
- *Execution region description* on page 3-8
- *Section placement* on page 3-10 in the *Linker User Guide*.

2.1.128 `--split`

This option splits the default load region, that contains the RO and RW output sections, into the following load regions:

- One region containing the RO output section. The default load address is `0x8000`, but a different address can be specified with the `--ro_base` option.
- One region containing the RW and ZI output sections. The load address is specified with the `--rw_base` option. This option requires a value for `--rw_base`. If `--rw_base` is not specified, `--rw_base=0` is assumed.

Both regions are root regions.

Restrictions

You cannot use `--split` with `--scatter`, `--shared`, or `--sysv`.

See also

- *--ro_base=address* on page 2-74
- *--rw_base=address* on page 2-76
- *The image structure* on page 3-2 in the *Linker User Guide*.

2.1.129 `--startup=`*`symbol`*`, --no_startup`

This option enables the linker to use alternative C libraries with a different startup symbol if required.

Syntax

`--startup=`*`symbol`*

By default, *`symbol`* is set to `__main`.

`--no_startup` does not take a *symbol* argument.

Default

The default is `--startup=__main`.

Usage

The linker includes the C library startup code if there is a reference to a symbol that is defined by the C library startup code. This symbol reference is called the startup symbol. It is automatically created by the linker when it sees a definition of main(). The `--startup` option allows you to change this symbol reference.

- If the linker finds a definition of `main()` and does not find a reference to (or definition of) *`symbol`*, then it generates an error.
- If the linker finds a definition of `main()` and a reference to (or definiton of) *`symbol`*, and no entry point is specified, then the linker generates a warning.

See also

- *--entry=location* on page 2-33.

2.1.130 `--strict`

This option instructs the linker to report conditions that might result in failure as errors, rather than warnings. An example of such a condition is taking the address of an interworking function from a non-interworking function.

See also

- *--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
- *--strict_enum_size, --no_strict_enum_size*
- *--strict_flags, --no_strict_flags* on page 2-85
- *--strict_ph, --no_strict_ph* on page 2-85
- *--strict_relocations, --no_strict_relocations* on page 2-86
- *--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.

2.1.131 `--strict_enum_size`, `--no_strict_enum_size`

The option `--strict_enum_size` causes the linker to display an error message if the enum size is not consistent across all inputs. This is the default.

Use --no_strict_enum_size for compatibility with objects built using RVCT v3.1 and earlier.

See also

- *--strict* on page 2-84
- *--strict_flags, --no_strict_flags*
- *--strict_ph, --no_strict_ph*
- *--strict_relocations, --no_strict_relocations* on page 2-86
- *--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
- *--enum_is_int* on page 2-53 in the *Compiler Reference Guide*.

2.1.132 --strict_flags, --no_strict_flags

The option --strict_flags prevents the EF_ARM_HASENTRY flag from being generated.

Default

The default is --no_strict_flags.

See also

- *--strict* on page 2-84
- *--strict_enum_size, --no_strict_enum_size* on page 2-84
- *--strict_ph, --no_strict_ph*
- *--strict_relocations, --no_strict_relocations* on page 2-86
- *--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
- *ARM ELF Specification (SWS ESPC 0003 B-02)*.

2.1.133 --strict_ph, --no_strict_ph

The linker writes the contents of load regions into the output ELF file in the order that load regions are written in the scatter file. Each load region is represented by one ELF program segment. In RVCT v2.2 the Program Header Table entries describing the program segments are given the same order as the program segments in the ELF file. To be more compliant with the ELF specification, in RVCT v3.0 and later the Program Header Table entries are sorted in ascending virtual address order.

Use the `--no_strict_ph` command-line option to switch off the sorting of the Program Header Table entries.

See also

- *--strict* on page 2-84
- *--strict_enum_size, --no_strict_enum_size* on page 2-84
- *--strict_flags, --no_strict_flags* on page 2-85
- *--strict_relocations, --no_strict_relocations*
- *--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.

2.1.134 `--strict_relocations, --no_strict_relocations`

This option enables you to ensure *Application Binary Interface* (ABI) compliance of legacy or third party objects. It checks that branch relocation applies to a branch instruction bit-pattern. The linker generates an error if there is a mismatch.

Usage

Use `--strict_relocations` to instruct the linker to report instances of obsolete and deprecated relocations.

Relocation errors and warnings are most likely to occur if you are linking object files built with previous versions of the ARM tools.

Default

The default is `--no_strict_relocations`.

See also

- *--strict* on page 2-84
- *--strict_enum_size, --no_strict_enum_size* on page 2-84
- *--strict_flags, --no_strict_flags* on page 2-85
- *--strict_ph, --no_strict_ph* on page 2-85
- *--strict_symbols, --no_strict_symbols* on page 2-87
- *--strict_visibility, --no_strict_visibility* on page 2-88
- *Linker User Guide*:
  - *About mapping symbols* on page 4-2.

2.1.135 --strict_symbols, --no_strict_symbols

The option --strict_symbols checks that the mapping symbol type matches ABI symbol type. The linker displays a warning if the types do not match.

A mismatch can occur only if you have hand-coded your own assembler.

Default

The default is --no_strict_symbols.

Example

In the following assembler code the symbol sym has type STT_FUNC and is ARM:

```
    area code, readonly
        DCD sym + 4
        ARM
sym  PROC
        NOP
        THUMB
        NOP
        ENDP
        END
```

The difference in behavior is the meaning of DCD sym + 4:

- In pre-ABI linkers the state of the symbol is the state of the only of the mapping symbol at that location. In this example, the state is Thumb.
- In ABI linkers the type of the symbol is the state of the location of symbol plus the offset.

See also

- *--strict* on page 2-84
- *--strict_enum_size, --no_strict_enum_size* on page 2-84
- *--strict_flags, --no_strict_flags* on page 2-85
- *--strict_ph, --no_strict_ph* on page 2-85
- *--strict_relocations, --no_strict_relocations* on page 2-86
- *--strict_visibility, --no_strict_visibility* on page 2-88
- *Linker User Guide*:
  - *About mapping symbols* on page 4-2.

2.1.136 --strict_visibility, --no_strict_visibility

A linker is not allowed to match a symbol reference with STT_HIDDEN visibility to a dynamic shared object. Some older linkers might allow this.

Use --no_strict_visibility to allow a hidden visibility reference to match against a shared object.

Default

The default is --strict_visibility.

See also

- *--strict* on page 2-84
- *--strict_enum_size, --no_strict_enum_size* on page 2-84
- *--strict_flags, --no_strict_flags* on page 2-85
- *--strict_ph, --no_strict_ph* on page 2-85
- *--strict_relocations, --no_strict_relocations* on page 2-86
- *--strict_symbols, --no_strict_symbols* on page 2-87.

2.1.137 --strict_wchar_size, --no_strict_wchar_size

The option --strict_wchar_size causes the linker to display an error message if the wide character size is not consistent across all inputs. This is the default.

Use --no_strict_wchar_size for compatibility with objects built using RVCT v3.1 and earlier.

See also

- *--strict* on page 2-84
- *--strict_enum_size, --no_strict_enum_size* on page 2-84
- *--strict_flags, --no_strict_flags* on page 2-85
- *--strict_ph, --no_strict_ph* on page 2-85
- *--strict_relocations, --no_strict_relocations* on page 2-86
- *--strict_symbols, --no_strict_symbols* on page 2-87
- *--strict_visibility, --no_strict_visibility*
- the following in the *Compiler Reference Guide*:
  - *--wchar16* on page 2-135
  - *--wchar32* on page 2-135.

2.1.138 --symbolic

Sets the DF_SYMBOLIC flag in the SHT_DYNAMIC section for a shared library. This flag changes the symbol resolution algorithm of the dynamic linker for references within the library. The dynamic linker searches for symbols starting with the shared object rather than the executable image. If the referenced symbol cannot be found in the shared object, the dynamic linker searches the executable image and other shared objects as usual.

See also

- *--dynamic_linker=name* on page 2-30.

2.1.139 --symbols, --no_symbols

This option enables or disables the listing of each local and global symbol used in the link step, and its value.

Note

This does not include mapping symbols. Use `--list_mapping_symbols` to include mapping symbols in the output.

Default

The default is `--no_symbols`.

See also

- *--list_mapping_symbols, --no_list_mapping_symbols* on page 2-57.

2.1.140 --symdefs=*file*

This option creates a file containing the global symbol definitions from the output image.

Default

By default, all global symbols are written to the symdefs file. If a symdefs file called *file* already exists, the linker restricts its output to the symbols already listed in this file.

Note

If you do not want this behavior, be sure to delete any existing symdefs file before the link step.

Usage

If *file* is specified without path information, the linker searches for it in the directory where the output image is being written. If it is not found, it is created in that directory.

You can use the symbol definitions file as input when linking another image.

See also

- *Accessing symbols in another image* on page 4-12 in the *Linker User Guide*.

2.1.141 --symver_script=*file*

This option enables implicit symbol versioning where *file* is a symbol version script.

See also

- *Symbol versioning* on page 4-19 in the *Linker User Guide*.

2.1.142 --symver_soname

This option enables implicit symbol versioning in order to force static binding. Where a symbol has no defined version, the linker uses the *shared object name* (SONAME) of the file being linked.

Default

This is the default if you are generating a BPABI-compatible executable file but where you do not specify a version script with the option --symver_script.

See also

- *Symbol versioning* on page 4-19 in the *Linker User Guide*
- *Base Platform ABI for the ARM Architecture* (BPABI).

2.1.143 --sysv

This option creates a *System V* (SysV) formatted ELF executable file that can be used on ARM Linux.

See also

- *--bpabi* on page 2-17
- *--dll* on page 2-30
- *--prelink_support, --no_prelink_support* on page 2-68
- *--runpath=pathlist* on page 2-76
- *--shared* on page 2-80
- Chapter 4 *BPABI and SysV Shared Libraries and Executables*.

2.1.144 --tailreorder, --no_tailreorder

This option moves tail calling sections immediately before their target, if possible, to optimize the branch instruction at the end of a section. A tail calling section is a section that contains a branch instruction at the end of the section. The branch must have a relocation that targets a function at the start of a section.

Default

The default is `--no_tailreorder`.

Restrictions

The linker:

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

See also

- *--branchnop, --no_branchnop* on page 2-17
- *--inline, --no_inline* on page 2-47
- *Inlining* on page 3-26 in the *Linker User Guide*.

2.1.145 --thumb2_library, --no_thumb2_library

Enables you to link against the combined ARM and Thumb-2 library for use with Cortex-A and Cortex-R series processors.

Use the `--no_thumb2_library` option to revert to the ARMv5T and later libraries.

Default

The default is `--thumb2_library`.

See also

- *Library naming conventions* on page 2-117 in the *Libraries and Floating Point Support Guide*.

2.1.146 `--tiebreaker=`*option*

A tiebreaker is used when a sorting algorithm requires a total ordering of sections. It is used to resolve the order when the sorting criteria results in more than one input section with equal properties.

Syntax

`--tiebreaker=`*option*

where *option* is one of:

`creation` The order that the linker creates sections in its internal section data structure.

When the linker creates an input section for each ELF section in the input objects, it increments a global counter. The value of this counter is stored in the section as the creation index.

The creation index of a section is unique apart from the special case of inline veneers.

`cmdline` The order that the section appears on the linker command-line. The command-line order is defined as `File.Object.Section` where:

- `Section` is the section index, `sh_idx`, of the `Section` in the `Object`
- `Object` is the order that `Object` appears in the `File`
- `File` is the order the `File` appears on the command line.

The order the `Object` appears in the `File` is only significant if the file is an `ar` archive.

This option is useful if you are doing a binary difference between the results of different links, link1 and link2. If link2 has only small changes from link1, then you might want the differences in one source file to be localized. In general, creation index works well for objects, but because of the multiple pass selection of members from libraries, a small

difference such as calling a new function can result in a different order of objects and therefore a different tiebreak. The command-line index is more stable across builds.

Use this option with the `--scatter` option.

Default

The default option is `creation`.

See also

- *--any_sort_order=order* on page 2-10
- *--map, --no_map* on page 2-59
- *--section_index_display=type* on page 2-79
- *--sort=algorithm* on page 2-81.

Linker User Guide:
- *Examples of using sorting algorithms for .ANY sections* on page 5-35.

2.1.147 `--undefined=`*`symbol`*

This option causes the linker to:

1. Create a symbol reference to the specified symbol name.
2. Issue an implicit `--keep(`*`symbol`*`)` to prevent any sections brought in to define that symbol from being removed.

Syntax

`--undefined=`*`symbol`*

See also

- *--keep=section_id* on page 2-49
- *--undefined_and_export=symbol*.

2.1.148 `--undefined_and_export=`*`symbol`*

This option causes the linker to:

1. Create a symbol reference to the specified symbol name.
2. Issue an implicit `--keep(`*`symbol`*`)` to prevent any sections brought in to define that symbol from being removed.

3. Add an implicit EXPORT *symbol* to push the specified symbol into the dynamic symbol table.

Syntax

`--undefined_and_export=`*symbol*

Usage

Be aware of the following when using this option:

- It does not change the visibility of a symbol unless you specify the `--override_visibility` option.
- A warning is issued if the visibility of the specified symbol is not high enough.
- A warning is issued if the visibility of the specified symbol is overridden because you also specified the `--override_visibility` option.
- Hidden symbols are not exported unless you specify the `--override_visibility` option.

See also

- *--keep=section_id* on page 2-49
- *--override_visibility* on page 2-63
- *--undefined=symbol* on page 2-93
- *EXPORT* on page 2-103.

2.1.149 `--unresolved=`*symbol*

This option takes each reference to an undefined symbol and matches it to the global definition of the specified symbol.

Syntax

`--unresolved=`*symbol*

Where *symbol* must be both defined and global, otherwise it appears in the list of undefined symbols and the link step fails.

Usage

This option is particularly useful during top-down development, because it enables you to test a partially-implemented system by matching each reference to a missing function to a dummy function.

2.1.150 --use_definition_visibility

When the linker combines global symbols the visibility of the symbol is set with the strictest visibility of the symbols being combined. Therefore, a symbol reference with STV_HIDDEN visibility combined with a definition with STV_DEFAULT visibility results in a definition with STV_HIDDEN visibility.

This option enables the linker to use the visibility of the definition in preference to the visibility a reference when combining symbols. For example, a symbol reference with STV_HIDDEN visibility combined with a definition with STV_DEFAULT visibility results in a definition with STV_DEFAULT visibility.

This can be useful when you want a reference to not match a Shared Library, but you want to export the definition.

Note

This option is not ELF-compliant and is disabled by default. To create ELF-compliant images, you must use symbol references with the appropriate visibility.

See also

- *Symbol visibility* on page 4-5.

2.1.151 --userlibpath=*pathlist*

This option specifies a list of paths that are used to search for user libraries.

Syntax

`--userlibpath=`*`pathlist`*

Where *pathlist* is a comma-separated list of paths that are used to search for the required libraries. Do not include spaces between the comma and the path name when specifying multiple path names, for example, *path1,path2,path3,...,pathn*.

See also

- *--libpath=pathlist* on page 2-53

- *Library searching, selection, and scanning* on page 3-44 in the *Linker User Guide*.

2.1.152 --veneer_inject_type=*type*

This option controls the veneer layout when --largeregions mode is on.

Syntax

--veneer_inject_type=*type*

where *type* is one of:

individual
: The linker places veneers to ensure they can be reached by the largest amount of sections that use the veneer. Veneer reuse between execution regions is permitted. This type minimizes the number of veneers that are required but disrupts the structure of the image the most.

pool
: The linker:

  1. Collects veneers from a contiguous range of the execution region
  2. Places all the veneers generated from that range into a pool.
  3. Places that pool at the end of the range.

  A large execution region might have more than one range and therefore more than one pool. Although this type has much less impact on the structure of image, it has fewer opportunities for reuse. This is because a range of code cannot reuse a veneer in another pool. The linker calculates the range based on the presence of branch instructions that the linker predicts might require veneers. A branch is predicted to require a veneer when either:

  - a state change is required
  - the distance from source to target plus a contingency greater than the branch range.

  You can set the size of the contingency with the --veneer_pool_size=*size* option. By default the contingency size is set to 102400 bytes. The --info=veneerpools option provides information on how the linker has placed veneer pools.

Restrictions

You must use --largeregions with this option.

See also

- *--info=topic[,topic,...]* on page 2-44
- *--largeregions, --no_largeregions* on page 2-51
- *--veneer_pool_size=size*.

2.1.153 --veneer_pool_size=*size*

Sets the contingency size for the veneer pool in an execution region.

Syntax

`--veneer_pool_size=`*`pool`*

where *`pool`* is the size in bytes.

Default

The default size is 102400 bytes.

See also

- *--veneer_inject_type=type* on page 2-96.

2.1.154 --veneershare, --no_veneershare

This option enables or disables veneer sharing. Veneer sharing can cause a significant decrease in image size.

default

The default is `--veneershare`.

See also

- *--inlineveneer, --no_inlineveneer* on page 2-48
- *--piveneer, --no_piveneer* on page 2-64
- *Veneers* on page 3-23 in the *Linker User Guide*.

2.1.155 --verbose

This option prints detailed information about the link operation, including the objects that are included and the libraries from which they are taken. This output is particular useful for tracing undefined symbols reference or multiply defined symbols. Because this output is typically quite long, you might want to use this command with the `--list=file` command to redirect the information to *file*.

Use `--verbose` to output diagnostics to `stdout`.

See also

- *--muldefweak, --no_muldefweak* on page 2-62
- *--unresolved=symbol* on page 2-94.

2.1.156 --version_number

This option displays the version of `armlink` you are using.

Syntax

```
armlink --version_number
```

The linker displays the version number in the format `nnnbbb`, where:

- `nnn` is the version number
- `bbb` is the build number.

Example

Version 4.0.0 build 821 is displayed as `400821`.

See also

- *--help* on page 2-43
- *--vsn* on page 2-100

2.1.157 --vfemode=*mode*

Virtual Function Elimination (VFE) is a technique that enables the linker to identify more unused sections.

Use this option to specify how VFE, and *Runtime Type Information* (RTTI) objects, are eliminated.

Syntax

`--vfemode=`*mode*

Where *mode* is one of the following:

`on` Use the command-line option `--vfemode=on` to make the linker VFE aware.

In this mode the linker chooses `force` or `off` mode based on the content of object files:

- Where every object file contains VFE information or does not refer to C++ libraries, the linker assumes `force` mode and continues with the elimination.
- If any object file is missing VFE information and refers to a C++ library, for example, where code has been compiled with a previous release of the ARM tools, the linker assumes `off` mode, and VFE is disabled silently. Choosing `off` mode to disable VFE in this situation ensures that the linker does not remove a virtual function that is used by an object with no VFE information.

`off` Use the command-line option `--vfemode=off` to make `armlink` ignore any extra information supplied by the compiler. In this mode, the final image is the same as that produced by compiling and linking without VFE awareness.

`force` Use the command-line option `--vfemode=force` to make the linker VFE aware and force the VFE algorithm to be applied. If some of the object files do not contain VFE information, for example, where they have been compiled with a previous release of the ARM tools, the linker continues with the elimination but displays a warning to alert you to possible errors.

`force_no_rtti`

Use the command-line option `--vfemode=force_no_rtti` to make the linker VFE aware and force the removal of all RTTI objects. In this mode all virtual functions are retained.

Default

The default is `--vfemode=on`.

See also

- *Section elimination* on page 3-13 in the *Linker User Guide*.

2.1.158 --via=*file*

This option reads a further list of input filenames and linker options from *file*.

You can enter multiple --via options on the linker command line. The --via options can also be included within a via file.

See also

- *Overview of via files* on page A-2 in the *Compiler Reference Guide*.

2.1.159 --vsn

This option displays the version information and the license details.

See also

- *--help* on page 2-43
- *--show_cmdline* on page 2-80.

2.1.160 --workdir=*directory*

This option enables you to provide a working directory for a project template.

Note

Project templates only require working directories if they include files, for example, RealView Debugger configuration files.

Syntax

--workdir=*directory*

Where *directory* is the name of the project directory.

Restrictions

If you specify a project working directory using --workdir, then you must specify a project file using --project.

See also

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

2.1.161 --xref, --no_xref

This option lists all cross-references between input sections.

Default

The default is --no_xref.

See also

- *--xref{from|to}=object(section)*.

2.1.162 --xrefdbg, --no_xrefdbg

This option lists all cross-references between input debug sections.

Default

The default is --no_xrefdbg.

2.1.163 --xref{from|to}=*object(section)*

This option lists cross-references:

- from input *section* in *object* to other input sections
- to input *section* in *object* from other input sections.

This is a useful subset of the listing produced by the --xref linker option if you are interested in references from or to a specific input section. You can have multiple occurrences of this option to list references from or to more than one input section.

See also

- *--list=file* on page 2-56
- *--xref, --no_xref*
- *--xrefdbg, --no_xrefdbg*.

2.2 Steering file commands in alphabetical order

Steering file commands enable you to:

- manage symbols in the symbol table
- control the copying of symbols from the static symbol table to the dynamic symbol table
- store information about the libraries that a link unit depends on.

Note

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

The steering file commands supported by the linker are:

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

2.2.1 EXPORT

The EXPORT command specifies that a symbol can be accessed by other shared objects or executables.

Note

A symbol can be exported only if the reference has STV_DEFAULT visibility. You must use the --override_visibility command-line option to permit the linker to override symbol visibility to STV_DEFAULT.

Syntax

EXPORT *pattern* [AS *replacement_pattern*] [,*pattern* [AS *replacement_pattern*]] *

where:

pattern
Is a string, optionally including wildcard characters (either * or ?), that matches zero or more defined global symbols. If *pattern* does not match any defined global symbol, the linker ignores the command. The operand can match only defined global symbols.

If the symbol is not defined, the linker issues:

```
Warning: L6331W: No eligible global symbol matches pattern symbol
```

replacement_pattern
Is a string, optionally including wildcard characters (either * or ?), to which the defined global symbol is to be renamed. Wild characters must have a corresponding wildcard in *pattern*. The characters matched by the *replacement_pattern* wildcard are substituted for the *pattern* wildcard.

For example:

```
EXPORT my_func AS func1
```

renames and exports the defined symbol my_func as func1.

Usage

You cannot export a symbol to a name that already exists. Only one wildcard character (either * or ?) is permitted in EXPORT.

The defined global symbol is included in the dynamic symbol table (as *replacement_pattern* if given, otherwise as *pattern*), if a dynamic symbol table is present.

See also

- *--override_visibility* on page 2-63
- *IMPORT* on page 2-106
- *Hiding and renaming global symbols* on page 4-16 in the *Linker User Guide*.

2.2.2 HIDE

The HIDE command makes defined global symbols in the symbol table anonymous.

Syntax

```
HIDE  pattern [,pattern] *
```

where:

pattern Is a string, optionally including wildcard characters, that matches zero or more defined global symbols. If *pattern* does not match any defined global symbol, the linker ignores the command. You cannot hide undefined symbols.

Usage

HIDE and SHOW can be used to make certain global symbols anonymous in an output image or partially linked object. Hiding symbols in an object file or library can be useful as a means of protecting intellectual property, as shown in Example 2-5. This example produces a partially linked object with all global symbols hidden, except those beginning with os_.

Example 2-5 Using the HIDE command

```
steer.txt

; Hides all global symbols
HIDE *
; Shows all symbols beginning with 'os_'
SHOW os_*
```

Link this example with the command:

```
armlink --partial input_object.o --edit steer.txt -o partial_object.o
```

The resulting partial object can be linked with other objects, provided they do not contain references to the hidden symbols. When symbols are hidden in the output object, SHOW commands in subsequent link steps have no effect on them. The hidden references are removed from the output symbol table.

See also

- *SHOW* on page 2-113
- *Hiding and renaming global symbols* on page 4-16 in the *Linker User Guide*.

2.2.3 IMPORT

The IMPORT command specifies that a symbol is defined in a shared object at runtime.

Note

A symbol can be imported only if the reference has STV_DEFAULT visibility. You must use the --override_visibility command-line option to enable the linker to override symbol visibility to STV_DEFAULT.

Syntax

IMPORT *pattern* [AS *replacement_pattern*] [,*pattern* [AS *replacement_pattern*]] *

where:

pattern Is a string, optionally including wildcard characters (either * or ?), that matches zero or more undefined global symbols. If *pattern* does not match any undefined global symbol, the linker ignores the command. The operand can match only undefined global symbols.

replacement_pattern

Is a string, optionally including wildcard characters (either * or ?), to which the symbol is to be renamed. Wild characters must have a corresponding wildcard in *pattern*. The characters matched by the *pattern* wildcard are substituted for the *replacement_pattern* wildcard.

For example:

```
IMPORT my_func AS func
```

imports and renames the undefined symbol my_func as func.

Usage

You cannot import a symbol that has been defined in the current shared object or executable. Only one wildcard character (either * or ?) is permitted in IMPORT.

The undefined symbol is included in the dynamic symbol table (as *replacement_pattern* if given, otherwise as *pattern*), if a dynamic symbol table is present.

Note

The IMPORT command only affects undefined global symbols. Symbols that have been resolved by a shared library are implicitly imported into the dynamic symbol table. The linker ignores any IMPORT directive that targets an implicitly imported symbol.

See also

- *EXPORT* on page 2-103
- *Hiding and renaming global symbols* on page 4-16 in the *Linker User Guide*.

2.2.4 RENAME

The RENAME command renames defined and undefined global symbol names.

Syntax

RENAME *pattern* AS *replacement_pattern* [,*pattern* AS *replacement_pattern*] *

where:

pattern Is a string, optionally including wildcard characters (either * or ?), that matches zero or more global symbols. If *pattern* does not match any global symbol, the linker ignores the command. The operand can match both defined and undefined symbols.

replacement_pattern

Is a string, optionally including wildcard characters (either * or ?), to which the symbol is to be renamed. Wild characters must have a corresponding wildcard in *pattern*. The characters matched by the *pattern* wildcard are substituted for the *replacement_pattern* wildcard.

For example, for a symbol named func1:

```
RENAME f* AS my_f*
```

renames func1 to my_func1.

Usage

You cannot rename a symbol to a global symbol name that already exists, even if the target symbol name is being renamed itself.

You cannot rename a symbol to the same name as another symbol. For example, you cannot do the following:

```
RENAME foo1 bar
RENAME foo2 bar
```

Renames only take effect at the end of the link step. Therefore, renaming a symbol does not remove its original name. This means that you cannot do the following:

```
RENAME func1 func2
RENAME func2 func3
```

The linker gives an error that func1 cannot be renamed to func2 as a symbol already exists with that name.

Only one wildcard character (either * or ?) is permitted in RENAME.

Example

Given an image containing the symbols `func1`, `func2`, and `func3`, you might have a steering file containing the following commands:

```
;invalid, func2 already exists
EXPORT func1 AS func2

; valid
RENAME func3 AS b2
;invalid, func3 still exists because the link step is not yet complete
EXPORT func1 AS func3
```

See also

- *Hiding and renaming global symbols* on page 4-16 in the *Linker User Guide*.

2.2.5 REQUIRE

The REQUIRE command creates a DT_NEEDED tag in the dynamic array. DT_NEEDED tags specify dependencies to other shared objects used by the application, for example, a shared library.

Syntax

REQUIRE *pattern* [,*pattern*] *

where:

pattern Is a string representing a filename. No wild characters are permitted.

Usage

The linker inserts a DT_NEEDED tag with the value of *pattern* into the dynamic array. This tells the dynamic loader that the file it is currently loading requires *pattern* to be loaded.

Note

DT_NEEDED tags inserted as a result of a REQUIRE command are added after DT_NEEDED tags generated from shared objects or DLLs placed on the command line.

See also

- *Hiding and renaming global symbols* on page 4-16 in the *Linker User Guide*.

2.2.6 RESOLVE

The RESOLVE command matches specific undefined references to a defined global symbol.

Syntax

RESOLVE *pattern* AS *defined_pattern*

where:

pattern Is a string, optionally including wildcard characters (either * or ?), that matches zero or more undefined global symbols. If *pattern* does not match any undefined global symbol, the linker ignores the command. The operand can match only undefined global symbols.

defined_pattern

Is a string, optionally including wildcard characters, that matches zero or more defined global symbols. If *defined_pattern* does not match any defined global symbol, the linker ignores the command. You cannot match an undefined reference to an undefined symbol.

Usage

RESOLVE is an extension of the existing armlink --unresolved command-line option. The difference is that --unresolved enables all undefined references to match one single definition, whereas RESOLVE enables more specific matching of references to symbols.

The undefined references are removed from the output symbol table.

RESOLVE works when performing partial-linking and when linking normally.

For example, you might have two files file1.c and file2.c, as shown in Example 2-6.

Example 2-6 Using the RESOLVE command

```
file1.c

extern int foo;
extern void MP3_Init(void);
extern void MP3_Play(void);

int main(void)
{
  int x = foo + 1;
  MP3_Init();
```

```
  MP3_Play();
  return x;
}


file2.c:

int foobar;
void MyMP3_Init()
{
}
void MyMP3_Play()
{
}
```

Create a steering file `ed.txt` containing the line

```
RESOLVE MP3* AS MyMP3*
```

Enter the following command:

```
armlink file1.o file2.o --edit ed.txt --unresolved foobar
```

This command has the following effects:

- The references from `file1.o` (`foo`, `MP3_Init()` and `MP3_Play()`) are matched to the definitions in `file2.o` (`foobar`, `MyMP3_Init()` and `MyMP3_Play()` respectively), as specified by the steering file `ed.txt`.
- The `RESOLVE` command in `ed.txt` matches the `MP3` functions and the `--unresolved` option matches any other remaining references, in this case, `foo` to `foobar`.
- The output symbol table, whether it is an image or a partial object, does not contain the symbols `foo`, `MP3_Init` or `MP3_Play`.

See also

- *Hiding and renaming global symbols* on page 4-16 in the *Linker User Guide*.

2.2.7 SHOW

The SHOW command makes global symbols visible. This command is useful if you want to unhide a specific symbol that are hidden using a HIDE command with a wildcard.

Syntax

SHOW *pattern* [,*pattern*] *

where:

pattern Is a string, optionally including wildcard characters, that matches zero or more global symbols. If *pattern* does not match any global symbol, the linker ignores the command.

Usage

The usage of SHOW is closely related to that of HIDE.

See also

- *HIDE* on page 2-105
- *Hiding and renaming global symbols* on page 4-16 in the *Linker User Guide*.

Chapter 3
Formal syntax of the scatter-loading description file

This chapter describes the format of scatter-loading description files and includes the following sections:

- *Expression evaluation in scatter files* on page 3-30
- *Scatter files containing relative base address load regions and a ZI execution region* on page 3-36.

For more information on how to use a scatter-loading description file, see Chapter 5 *Using Scatter-loading Description Files* in the *Linker User Guide*.

3.1 BNF notation and syntax

Table 3-1 summarizes the *Backus-Naur Form* (BNF) symbols that are used to describe a formal language.

Table 3-1 BNF syntax

Symbol	Description
"	Quotation marks are used to indicate that a character that is normally part of the BNF syntax is used as a literal character in the definition. The definition `B"+"C`, for example, can only be replaced by the pattern `B+C`. The definition `B+C` can be replaced by, for example, patterns `BC`, `BBC`, or `BBBC`.
A ::= *B*	Defines *A* as *B*. For example, `A::= B"+" | C` means that *A* is equivalent to either `B+` or `C`. The `::=` notation is used to define a higher level construct in terms of its components. Each component might also have a `::=` definition that defines it in terms of even simpler components. For example, `A::= B` and `B::= C | D` means that the definition *A* is equivalent to the patterns `C` or `D`.
[*A*]	Optional element *A*. For example, `A::= B[C]D` means that the definition *A* can be expanded into either `BD` or `BCD`.
A+	Element *A* can have one or more occurrences. For example, `A::= B+` means that the definition *A* can be expanded into `B`, `BB`, or `BBB`.
*A**	Element *A* can have zero or more occurrences.
A | *B*	Either element *A* or *B* can occur, but not both.
(*A B*)	Element *A* and *B* are grouped together. This is particularly useful when the `|` operator is used or when a complex pattern is repeated. For example, `A::=(B C)+ (D | E)` means that the definition *A* can be expanded into any of `BCD`, `BCE`, `BCBCD`, `BCBCE`, `BCBCBCD`, or `BCBCBCE`.

Note

The BNF definitions in this section, contain additional line returns and spaces to improve readability. They are not required in the scatter-loading definition and are ignored if present in the file.

3.2 Syntax of a scatter-loading description file

Figure 3-1 shows the components and organization of a typical scatter-loading description file.

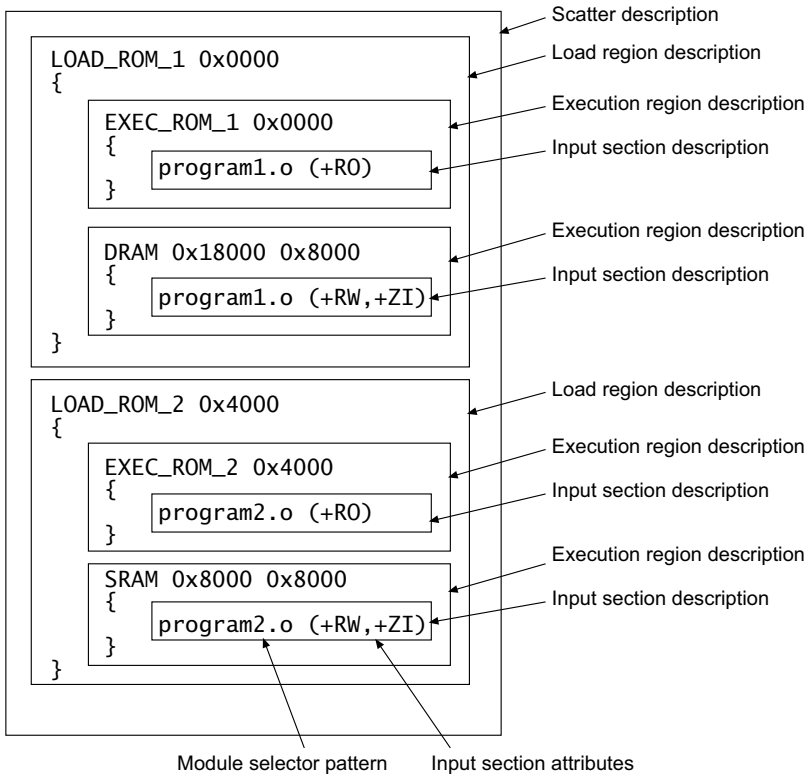

Figure 3-1 Components of a scatter-loading description file

3.3 Load region description

A load region has:

- a name (used by the linker to identify different load regions)
- a base address (the start address for the code and data in the load view)
- attributes that specify the properties of the load region
- an optional maximum size specification
- one or more execution regions.

Figure 3-2 shows the components of a typical load region description.

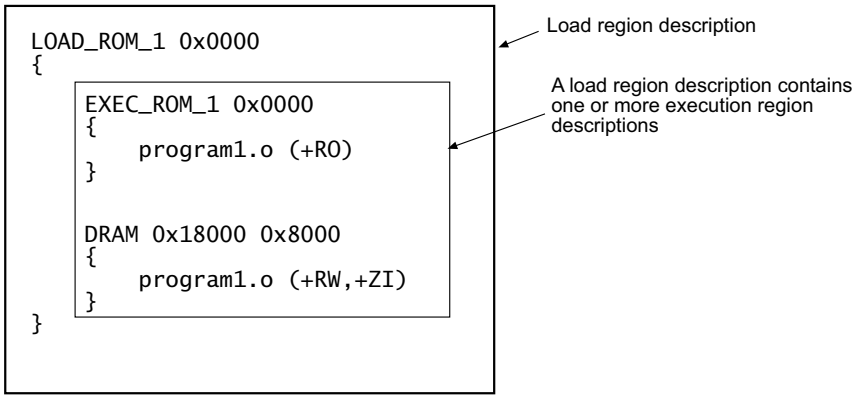

Figure 3-2 Components of a load region description

See also:

- *Syntax of a load region description*.

3.3.1 Syntax of a load region description

The syntax, in BNF, is:

```
load_region_description ::=
  load_region_name  (base_address  | ("+" offset)) [attribute_list] [max_size]
       "{"
          execution_region_description+
       "}"
```

where:

load_region_name

Names the load region.

base_address Specifies the address where objects in the region are to be linked. *base_address* must satisfy the alignment constraints of the load region.

+offset Describes a base address that is *offset* bytes beyond the end of the preceding load region. The value of *offset* must be zero modulo four. If this is the first load region, then *+offset* means that the base address begins *offset* bytes from zero.

If you use *+offset*, then the load region might inherit certain attributes from a previous load region.

See also *Considerations when using a relative address +offset for load regions* on page 3-16.

attribute_list

Specifies the properties of the load region contents:

ABSOLUTE Absolute address. The load address of the region is specified by the base designator. This is the default, unless you use PI or RELOC.

ALIGN *alignment*. Increase the alignment constraint for the load region from 4 to *alignment*. *alignment* must be a positive power of 2. If the load region has a *base_address* then this must be *alignment* aligned. If the load region has a *+offset* then the linker aligns the calculated base address of the region to an *alignment* boundary.

This can also affect the offset in the ELF file. For example, the following causes the data for FOO to be written out at 4k offset into the ELF file:

```
FOO +4 ALIGN 4096
```

NOCOMPRESS RW data compression is enabled by default. The NOCOMPRESS keyword enables you to specify that the contents of a load region must not be compressed in the final image.

OVERLAY The OVERLAY keyword enables you to have multiple load regions at the same address. ARM tools do not provide an overlay mechanism. To use multiple load regions at the same address, you must provide your own overlay manager.

PI This region is position independent.

RELOC This region is relocatable.

max_size Specifies the maximum size of the load region. This is the size of the load region before any decompression or zero initialization take place. If the optional *max_size* value is specified, armlink generates an error if the region has more than *max_size* bytes allocated to it.

execution_region_description

Specifies the execution region name, address, and contents. See *Execution region description* on page 3-8.

Note

The *Backus-Naur Form* (BNF) definitions contain additional line returns and spaces to improve readability. They are not required in the scatter-loading definition and are ignored if present in the file.

3.4 Execution region description

An execution region has:

- a name (used by the linker to identify different execution regions)
- a base address (either absolute or relative)
- attributes that specify the properties of the execution region
- an optional maximum size specification
- one or more input section descriptions (the modules placed into this execution region).

Figure 3-3 shows the components of a typical execution region description.

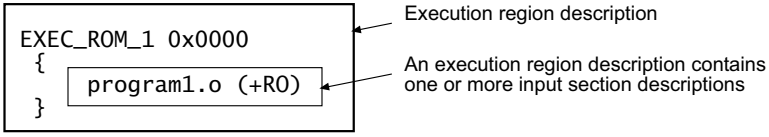

Figure 3-3 Components of an execution region description

See also:

- *Syntax of an execution region description*.

3.4.1 Syntax of an execution region description

The execution region description syntax, in BNF, is:

```
execution_region_description ::=

  exec_region_name (base_address | "+" offset) [attribute_list] [max_size | length]
        "{"
            input_section_description*
        "}"
```

where:

exec_region_name

Names the execution region.

base_address Specifies the address where objects in the region are to be linked. *base_address* must be word-aligned.

Note

Using `ALIGN` on an execution region causes both the load address and execution address to be aligned.

+offset Describes a base address that is *offset* bytes beyond the end of the preceding execution region. The value of *offset* must be zero modulo four.

If this is the first execution region in the load region then *+offset* means that the base address begins *offset* bytes after the base of the containing load region.

If you use *+offset*, then the execution region might inherit certain attributes from the parent load region, or from a previous execution region within the same load region.

See also *Considerations when using a relative address +offset for execution regions* on page 3-17.

attribute_list

This specifies the properties of the execution region contents:

ABSOLUTE Absolute address. The execution address of the region is specified by the base designator.

ALIGN *alignment* Increase the alignment constraint for the execution region from 4 to *alignment*. *alignment* must be a positive power of 2. If the execution region has a *base_address* then this must be *alignment* aligned. If the execution region has a *+offset* then the linker aligns the calculated base address of the region to an *alignment* boundary.

Note

ALIGN on an execution region causes both the load address and execution address to be aligned. This can result in padding being added to the ELF file.

ANY_SIZE *max_size* Specifies the maximum size within the execution region that armlink can fill with unassigned sections. You can use a simple expression to specify the *max_size*. That is, you cannot use functions such as ImageLimit().

Note

max_size is not the contingency, but the maximum size permitted for placing unassigned sections in an execution region. For example, if an execution region is to be filled only with .ANY sections, a two

percent contingency is still set aside for veneers. This leaves 98% of the region for .ANY section assignements.

Be aware of the following restrictions when using this keyword:

- *max_size* must be less than or equal to the region size
- you can use ANY_SIZE on a region without a .ANY selector but it is ignored by armlink.

EMPTY [-]*length*
: Reserves an empty block of memory of a given length in the execution region, typically used by a heap or stack. No section can be placed in a region with the EMPTY attribute.

  length represent a stack that grows down in memory. If the length is given as a negative value, the *base_address* is taken to be the end address of the region.

FILL *value*
: Creates a linker generated region containing a value. If you specify FILL, you must give a value, for example: FILL 0xFFFFFFFF. The FILL attribute replaces the following combination: EMPTY ZEROPAD PADVALUE.

  In certain situations, for example, simulation, this is preferable to spending a long time in a zeroing loop.

FIXED
: Fixed address. The linker attempts to make the execution address equal the load address by inserting padding. This makes the region a root region. If this is not possible the linker produces an error.

  Note

  The linker inserts padding with this attribute.

NOCOMPRESS
: RW data compression is enabled by default. The NOCOMPRESS keyword enables you to specify that RW data in an execution region must not be compressed in the final image.

OVERLAY	Use for sections with overlaying address ranges. If consecutive execution regions have the same +*offset* then they are given the same base address. See *Using overlays to place sections* on page 5-38 in the *Linker User Guide* for more information.
PADVALUE	Defines the value of any padding. If you specify PADVALUE, you must give a value, for example: `EXEC 0x10000 PADVALUE 0xFFFFFFFF EMPTY ZEROPAD 0x2000` This creates a region of size `0x2000` full of 0xFFFFFFFF. PADVALUE must be a word in size. PADVALUE attributes on load regions are ignored.
PI	This region contains only position independent sections.
SORTTYPE	Specifies the sorting algorithm for the execution region, for example: `ER1 +0 SORTTYPE=CallTree` See *--sort=algorithm* on page 2-81 for more information.
UNINIT	Use to create execution regions containing uninitialized data or memory-mapped I/O.
ZEROPAD	Zero-initialized sections are written in the ELF file as a block of zeros and, therefore, do not have to be zero-filled at runtime. This sets the load length of a ZI output section to `Image$$`*region_name*`$$ZI$$Length`. Only root execution regions can be zero-initialized using the ZEROPAD attribute. Using the ZEROPAD attribute with a non root execution region generates a warning and the attribute is ignored. In certain situations, for example, simulation, this is preferable to spending a long time in a zeroing loop.

max_size For an execution region marked EMPTY or FILL the *max_size* value is interpreted as the length of the region. Otherwise the *max_size* value is interpreted as the maximum size of the execution region.

[-]length Can only be used with EMPTY to represent a stack that grows down in memory. If the length is given as a negative value, the *base_address* is taken to be the end address of the region.

input_section_description

Specifies the content of the input sections. See *Input section description* on page 3-18.

Note

The *Backus-Naur Form* (BNF) definitions contain additional line returns and spaces to improve readability. They are not required in the scatter-loading definition and are ignored if present in the file.

3.5 Addressing attributes

A subset of the load and execution region attributes inform the linker about the content of the region and how it behaves after linking. These attributes are:

`ABSOLUTE` The content is placed at a fixed address that does not change after linking.

`PI` The content does not depend on any fixed address and might be moved after linking without any extra processing.

`RELOC` The content depends on fixed addresses, relocation information is output to enable the content to be moved to another location by another tool.

Note

You cannot explicitly use this attribute for an execution region.

`OVERLAY` The content is placed at a fixed address that does not change after linking. The content might overlap with other regions with `OVERLAY`.

In general, all the execution regions within a load region have the same addressing attribute. To make this easy to select, the addressing attributes can be inherited from a previous region so that they only have to be set in one place. The rules for setting and inheriting addressing attributes are:

- Explicitly setting the addressing attribute:
  - A load region can be explicitly set with the `ABSOLUTE`, `PI`, `RELOC`, or `OVERLAY` attributes.
  - An execution region can be explicitly set with the `ABSOLUTE`, `PI`, or `OVERLAY` attributes. For an execution region to be set with the `RELOC` attribute, it must inherit from the parent load region.
- Implicitly setting the addressing attribute when none is specified:
  - The `OVERLAY` attribute cannot be inherited. A region with the `OVERLAY` attribute cannot inherit.
  - A base address load or execution region always defaults to `ABSOLUTE`.
  - A `+offset` load region inherits the addressing attribute from the previous load region or `ABSOLUTE` if no previous load region exists.
  - A `+offset` execution region inherits the addressing attribute from the previous execution region or parent load region if no previous execution region exists.

Inheritance rules for setting the addressing attributes are shown in Example 3-1 on page 3-14, Example 3-2 on page 3-14, and Example 3-3 on page 3-15.

Example 3-1 Load region inheritance

```
LR1 0x8000 PI
{
    ...
}
LR2 +0              ; LR2 inherits PI from LR1
{
    ...
}
LR3 0x1000          ; LR3 does not inherit because it has no relative base,
                      gets default of ABSOLUTE
{
    ...
}
LR4 +0              ; LR4 inherits ABSOLUTE from LR3
{
    ...
}
LR5 +0 RELOC        ; LR5 does not inherit because it explicitly sets RELOC
{
    ...
}
LR6 +0 OVERLAY      ; LR6 does not inherit, an OVERLAY cannot inherit
{
    ...
}
LR7 +0              ; LR7 cannot inherit OVERLAY, gets default of ABSOLUTE
{
    ...
}
```

Example 3-2 Execution region inheritance

```
LR1 0x8000 PI
{
    ER1 +0          ; ER1 inherits PI from LR1
    {
        ...
    }
    ER2 +0          ; ER2 inherits PI from ER1
    {
        ...
    }
    ER3 0x10000     ; ER3 does not inherit because it has no relative base
                      address and gets the default of ABSOLUTE
```

```
    {
        ...
    }
    ER4 +0         ; ER4 inherits ABSOLUTE from ER3
    {
        ...
    }
    ER5 +0 PI      ; ER5 does not inherit, it explicitly sets PI
    {
        ...
    }
    ER6 +0 OVERLAY ; ER6 does not inherit, an OVERLAY cannot inherit
    {
        ...
    }
    ER7 +0        ; ER7 cannot inherit OVERLAY, gets the default of ABSOLUTE
    {
        ...
    }
}
```

Example 3-3 Inheriting RELOC

```
LR1 0x8000 RELOC
{
    ER1 +0 ; inherits RELOC from LR1
    {
        ...
    }
    ER2 +0 ; inherits RELOC from ER1
    {
        ...
    }
    ER3 +0 RELOC ; Error cannot explicitly set RELOC on an execution region
    {
        ...
    }
}
```

See also:

- *Considerations when using a relative address +offset for load regions* on page 3-16
- *Considerations when using a relative address +offset for execution regions* on page 3-17.

3.6 Considerations when using a relative address +offset for load regions

Be aware of the following when using *+offset* to specify a load region base address:

- If the *+offset* load region LR2 follows a load region LR1 containing ZI data, then LR2 overlaps the ZI data. To fix this, use the `ImageLimit()` function to specify the base address of LR2.
- A *+offset* load region LR2 inherits the attributes of the load region LR1 immediately before it, unless:
  - LR1 has the `OVERLAY` attribute
  - LR2 has an explicit attribute set.

  If a load region is unable to inherit an attribute, then it gets the attribute `ABSOLUTE`.

See also:

- *Addressing attributes* on page 3-13.

3.7 Considerations when using a relative address +offset for execution regions

Be aware of the following when using *+offset* to specify a load region base address:

- The first execution region inherits the attributes of the parent load region, unless an attribute is explicitly set on that execution region.
- A *+offset* execution region ER2 inherits the attributes of the execution region ER1 immediately before it, unless:
  - — ER1 has the OVERLAY attribute
  - — ER2 has an explicit attribute set.

  If an execution region is unable to inherit an attribute, then it gets the attribute ABSOLUTE.
- If the parent load region has the RELOC attribute, then all execution regions within that load region must have a *+offset* base address.

See also:

- *Addressing attributes* on page 3-13.

3.8 Input section description

An input section description is a pattern that identifies input sections by:

- Module name (object filename, library member name, or library filename). The module name can use wildcard characters.
- Input section name, or input section attributes such as `READ-ONLY`, or `CODE`.
- Symbol name.

Figure 3-4 shows the components of a typical input section description.

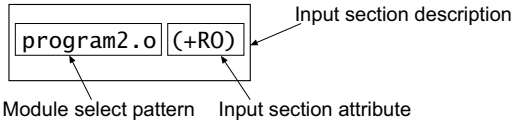

Figure 3-4 Components of an input section description

Note

Ordering in an execution region does not affect the ordering of sections in the output image.

See also:

- *Syntax of an input section description*
- *Example of module and input select patterns* on page 3-22.

3.8.1 Syntax of an input section description

The input section description syntax, in BNF, is:

```
input_section_description ::=

  module_select_pattern
        [ "(" input_section_selector ( "," input_section_selector )* ")" ]

input_section_selector ::=

        ("+" input_section_attr | input_section_pattern | input_symbol_pattern |
section_properties)
```

where:

module_select_pattern

A pattern constructed from literal text. The wildcard character * matches zero or more characters and ? matches any single character.

Matching is case-insensitive, even on hosts with case-sensitive file naming.

Use *.o to match all objects. Use * to match all object files and libraries.

An input section matches a module selector pattern when *module_select_pattern* matches one of the following:

- The name of the object file containing the section.
- The name of the library member (without leading path name).
- The full name of the library (including path name) the section is extracted from. If the names contain spaces, use wild characters to simplify searching. For example, use *libname.lib to match C:\lib dir\libname.lib.

The following module selector patterns describe the placement order of an input section within the execution region:

.ANY module selector for unassigned sections

The special module selector pattern .ANY enables you to assign input sections to execution regions without considering their parent module. Use .ANY to fill up the execution regions with input sections that do not have to be placed at specific locations.

Modified selectors

You cannot have two * selectors in a scatter file. You can, however, use two modified selectors, for example *A and *B, and you can use a .ANY selector together with a * module selector. The * module selector has higher precedence than .ANY. If the portion of the file containing the * selector is removed, the .ANY selector then becomes active.

Note

- Only input sections that match both *module_select_pattern* and at least one *input_section_attr* or *input_section_pattern* are included in the execution region.

  If you omit (+ *input_section_attr*) and *(input_section_pattern)*, the default is +RO.

- Do not rely on input section names generated by the compiler, or used by ARM library code. These can change between compilations if, for example, different compiler options are used. In addition, section naming conventions used by the compiler are not guaranteed to remain constant between releases.

---

input_section_attr

An attribute selector matched against the input section attributes. Each *input_section_attr* follows a +.

If you are specifying a pattern to match the input section name, the name must be preceded by a +. You can omit any comma immediately followed by a +.

The selectors are not case-sensitive. The following selectors are recognized:

- RO-CODE
- RO-DATA
- RO, selects both RO-CODE and RO-DATA
- RW-DATA
- RW-CODE
- RW, selects both RW-CODE and RW-DATA
- ZI
- ENTRY, that is, a section containing an ENTRY point.

The following synonyms are recognized:

- CODE for RO-CODE
- CONST for RO-DATA
- TEXT for RO
- DATA for RW
- BSS for ZI.

The following pseudo-attributes are recognized:

- FIRST
- LAST.

Use FIRST and LAST to mark the first and last sections in an execution region if the placement order is important. For example, if a specific input section must be first in the region and an input section containing a checksum must be last.

There can be only one FIRST or one LAST attribute for an execution region, and it must follow a single *input_section_attr*. For example:

***(section, +FIRST)**

This pattern is correct.

***(+FIRST, section)**

This pattern is incorrect and produces an error message.

input_section_pattern

A pattern that is matched, without case sensitivity, against the input section name. It is constructed from literal text. The wildcard character * matches 0 or more characters, and ? matches any single character.

Note

If you use more than one *input_section_pattern*, ensure that there are no duplicate patterns in different execution regions in order to avoid ambiguity errors.

input_symbol_pattern

You can select the input section by the name of a global symbol that the section defines. This enables you to choose individual sections with the same name from partially linked objects.

The :gdef: prefix distinguishes a global symbol pattern from a section pattern. For example, use :gdef:mysym to select the section that defines mysym. The following example shows a description file in which ExecReg1 contains the section that defines global symbol mysym1, and the section that contains global symbol mysym2:

```
LoadRegion 0x8000
{
    ExecReg1 +0
    {
        *(:gdef:mysym1)
        *(:gdef:mysym2)
    }
                          ; rest of scatter description
}
```

Note

If you use more than one *input_symbol_pattern*, ensure that there are no duplicate patterns in different execution regions in order to avoid ambiguity errors.

The order of input section descriptors is not significant.

section_properties

A section property can be `+FIRST` or `+LAST`.

Note

The BNF definitions contain additional line returns and spaces to improve readability. They are not required in the scatter-loading definition and are ignored if present in the file.

See also

- *Input section description* on page 3-18
- the following in the *Linker User Guide*:
  - *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.

3.8.2 Example of module and input select patterns

Examples of *`module_select_pattern`* specifications are:

- `*` matches any module or library
- `*.o` matches any object module
- `math.o` matches the `math.o` module
- `*armlib*` matches all C libraries supplied by ARM
- `*math.lib` matches any library path ending with `math.lib`. For example, `C:\apps\lib\math\satmath.lib`.

Examples of *`input_section_selector`* specifications are:

- `+RO` is an input section attribute that matches all RO code and all RO data
- `+RW,+ZI` is an input section attribute that matches all RW code, all RW data, and all ZI data
- `BLOCK_42` is an input section pattern that matches sections named `BLOCK_42`. There can be multiple ELF sections with the same `BLOCK_42` name that possess different attributes, for example `+RO-CODE,+RW`.

See also

- *Input section description* on page 3-18.

3.9 Resolving multiple matches

An input section must be unique. In the case of multiple matches, the linker attempts to assign the input section to a region based on a *module_select_pattern* and *input_section_selector* pair that is the most specific. However, if a unique match cannot be found, the linker faults the scatter-loading description.

The following variables are used to describe multiple matches:

- *m1* and *m2* represent module selector patterns
- *s1* and *s2* represent input section selectors.

For example, if input section A matches *m1,s1* for execution region R1, and A matches *m2,s2* for execution region R2, the linker:

- assigns A to R1 if *m1,s1* is more specific than *m2,s2*
- assigns A to R2 if *m2,s2* is more specific than *m1,s1*
- diagnoses the scatter-loading description as faulty if *m1,s1* is not more specific than *m2,s2* and *m2,s2* is not more specific than *m1,s1*.

`armlink` uses the following sequence to determine the most specific *module_select_pattern*, *input_section_selector* pair is as follows:

1. For the module selector patterns:

   m1 is more specific than *m2* if the text string *m1* matches pattern *m2* and the text string *m2* does not match pattern *m1*.

2. For the input section selectors:
   - If *s1* and *s2* are both patterns matching section names, the same definition as for module selector patterns is used.
   - If one of *s1*, *s2* matches the input section name and the other matches the input section attributes, *s1* and *s2* are unordered and the description is diagnosed as faulty.
   - If both *s1* and *s2* match input section attributes, the determination of whether *s1* is more specific than *s2* is defined by the relationships below:
     - `ENTRY` is more specific than `RO-CODE`, `RO-DATA`, `RW-CODE` or `RW-DATA`
     - `RO-CODE` is more specific than `RO`
     - `RO-DATA` is more specific than `RO`
     - `RW-CODE` is more specific than `RW`
     - `RW-DATA` is more specific than `RW`
     - There are no other members of the (*s1* more specific than *s2*) relationship between section attributes.

3. For the *module_select_pattern, input_section_selector* pair, *m1,s1* is more specific than *m2,s2* only if any of the following are true:
   a. *s1* is a literal input section name that is, it contains no pattern characters, and *s2* matches input section attributes other than +ENTRY
   b. *m1* is more specific than *m2*
   c. *s1* is more specific than *s2*.

   The conditions are tested in order so condition a takes precedence over condition b and c, and condition b takes precedence over condition c.

This matching strategy has the following consequences:

- Descriptions do not depend on the order they are written in the file.
- Generally, the more specific the description of an object, the more specific the description of the input sections it contains.
- The *input_section_selector*s are not examined unless:
  — Object selection is inconclusive.
  — One selector fully names an input section and the other selects by attribute. In this case, the explicit input section name is more specific than any attribute, other than ENTRY, that selects exactly one input section from one object. This is true even if the object selector associated with the input section name is less specific than that of the attribute.

The .ANY module selector is available to assign any sections that cannot be resolved from the scatter-file description.

Example 3-4 shows multiple execution regions and pattern matching.

Example 3-4 Multiple execution regions and pattern matching

```
LR_1 0x040000
{
    ER_ROM 0x040000              ; The startup exec region address is the same
    {                            ; as the load address.
        application.o (+ENTRY)   ; The section containing the entry point from
    }                            ; the object is placed here.
    ER_RAM1 0x048000
    {
        application.o (+RO-CODE) ; Other RO code from the object goes here
    }
    ER_RAM2 0x050000
    {
        application.o (+RO-DATA) ; The RO data goes here
```

```
    }
    ER_RAM3 0x060000
    {
        application.o (+RW)      ; RW code and data go here
    }
    ER_RAM4 +0                   ; Follows on from end of ER_R3
    {
        *.o (+RO, +RW, +ZI)      ; Everything except for application.o goes here
    }
}
```

See also:

- Chapter 3 *Formal syntax of the scatter-loading description file*
- *Placing unassigned sections with the .ANY module selector* on page 5-26 in the *Linker User Guide*

3.10 Behavior when .ANY sections overflow because of linker-generated content

Linker-generated content might cause .ANY regions to overflow. This is because the linker does not know the address of a section until it is assigned to a region. Therefore, when filling .ANY regions, the linker cannot calculate the contingency space and cannot determine if calling functions require veneers. The linker provides a contingency algorithm that gives a worst-case estimate for padding and an additional two percent for veneers. To enable this algorithm use the --any_contingency command-line option.

The following diagram is a representation of the notional image layout during .ANY placement:

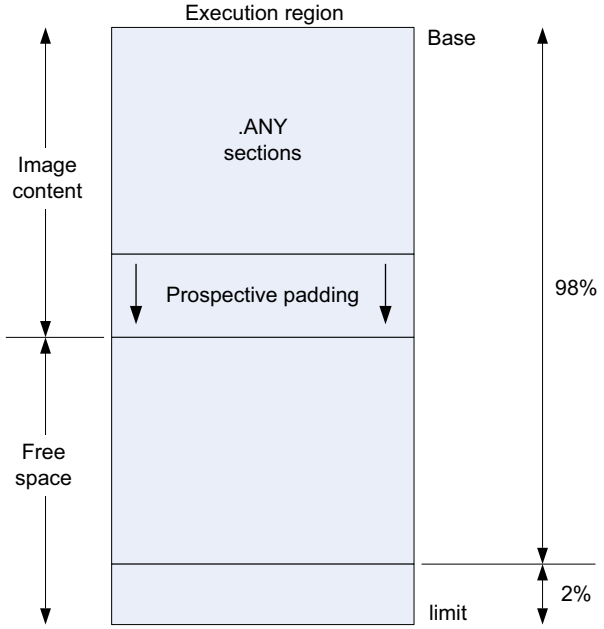

Figure 3-5 .ANY contingency

The downward arrows for prospective padding show that the prospective padding continues to grow as more sections are added to the .ANY selector.

Prospective padding is dealt with before the two percent veneer contingency.

When the prospective padding is cleared the priority is set to zero. When the two percent is cleared the priority is decremented again.

You can also use the ANY_SIZE keyword on an execution region to specify the maximum amount of space in the region to set aside for .ANY section assignments.

3.10.1 See also

- *--any_contingency* on page 2-8
- *Execution region description* on page 3-8
- *Input section description* on page 3-18
- *Resolving multiple matches* on page 3-24
- *Placing unassigned sections with the .ANY module selector* on page 5-26 in the *Linker User Guide*.

3.11 Resolving path names

The linker matches wildcard patterns in scatter files against any combination of forward slashes and backslashes it finds in path names. This might be useful where the paths are taken from environment variables or multiple sources, or where you want to use the same scatter file to build on Windows or Unix platforms.

Note

Use forward slashes in path names to ensure they are understood on Windows and Unix platforms.

See also:

- Chapter 3 *Formal syntax of the scatter-loading description file*.

3.12 Expression evaluation in scatter files

Scatter files frequently contain numeric constants.You can use specify numeric constants using:

- Expressions.
- Execution address built-in functions.
- `ScatterAssert` function with load address related functions that take an expression as a parameter. An error message is generated if this expression does not evaluate to true.
- The symbol related function, `defined(`*`global_symbol_name`*`) ? `*`expr1`*` : `*`expr2`*.

See also:

- *Expression usage in scatter files*
- *Expression rules in scatter files*
- *Execution address built-in functions for use in scatter files* on page 3-31
- *ScatterAssert function* on page 3-32
- *Symbol related function in a scatter file* on page 3-32
- *Examples of expression evaluation in scatter-loading files* on page 3-33.

3.12.1 Expression usage in scatter files

Expressions can be used in the following places:

- load and execution region *`base_address`*
- load and execution region `+`*`offset`*
- load and execution region *`max_size`*
- parameter for the `ALIGN`, `FILL` or `PADVALUE` keywords
- parameter for the `ScatterAssert` function.

See also

- *Expression evaluation in scatter files*.

3.12.2 Expression rules in scatter files

Expressions follow the C-Precedence rules and are made up of the following:

- Decimal or hexadecimal numbers.
- Arithmetic operators: `+`, `-`, `/`, `*`, `~`, `OR`, and `AND`

  The `OR` and `AND` operators map to the C operators | and & respectively.

- Logical operators: LOR, LAND, and !

  The LOR and LAND operators map to the C operators || and && respectively.

- Relational operators: <, <=, >, >=, and ==

  Zero is returned when the expression evaluates to false and nonzero is returned when true.

- Conditional operator: *Expression* ? *Expression1* : *Expression2*

  This matches the C conditional operator. If *Expression* evaluates to nonzero then *Expression1* is evaluated otherwise *Expression2* is evaluated.

- Functions that return numbers. See *Execution address built-in functions for use in scatter files* for more information.

All operators match their C counterparts in meaning and precedence.

Expressions are not case sensitive and parentheses can be used for clarity.

See also

- *Expression evaluation in scatter files* on page 3-30.

3.12.3 Execution address built-in functions for use in scatter files

The execution address related functions can only be used when specifying a *base_address* or +*offset* value. They map to combinations of the linker defined symbols shown in Table 3-2.

Table 3-2 Execution address related functions

Function	Linker defined symbol value
ImageBase(*region_name*)	Image$$*region_name*$$Base
ImageLength(*region_name*)	Image$$*region_name*$$Length + Image$$region_name$$ZI$$Length
ImageLimit(*region_name*)	Image$$region_name$$Base + Image$$region_name$$Length + Image$$region_name$$ZI$$Length

See Table 4-1 on page 4-5 in the *Linker User Guide* for more information on region-related linker symbols.

The parameter *region_name* can be either a load or an execution region name. Forward references are not allowed. The *region_name* can only refer to load or execution regions that have already been defined. See Example 3-6 on page 3-33.

Note

You cannot use these functions when using the `.ANY` selector pattern. This is because a `.ANY` region uses the maximum size when assigning sections. The maximum size might not be available at that point, because the size of all regions is not known until after the `.ANY` assignment.

See also

- *Expression evaluation in scatter files* on page 3-30.

3.12.4 ScatterAssert function

The `ScatterAssert(`*`expression`*`)` function can be used at the top level, or within a load region. It is evaluated after the link has completed and gives an error message if *`expression`* evaluates to false. Example 3-9 on page 3-35 shows how to use the `ScatterAssert` function to write more complex size checks than those allowed by the *`max_size`* of the region.

The load address related functions can only be used within the `ScatterAssert` function. They map to the three linker defined symbol values as shown in Table 3-3.

Table 3-3 Load address related functions

Function	Linker defined symbol value
`LoadBase(`*`region_name`*`)`	`Load$$`*`region_name`*`$$Base`
`LoadLength(`*`region_name`*`)`	`Load$$`*`region_name`*`$$Length`
`LoadLimit(`*`region_name`*`)`	`Load$$`*`region_name`*`$$Limit`

The parameter *`region_name`* can be either a load or an execution region name. Forward references are not allowed. The *`region_name`* can only refer to load or execution regions that have already been defined. See Example 3-6 on page 3-33.

See also

- *Expression evaluation in scatter files* on page 3-30.

3.12.5 Symbol related function in a scatter file

The symbol related function, `defined(`*`global_symbol_name`*`)` returns zero if *`global_symbol_name`* is not defined and nonzero if it is defined.

See also

- *Expression evaluation in scatter files* on page 3-30.

3.12.6 Examples of expression evaluation in scatter-loading files

Example 3-5 Specifying the maximum size in terms of an expression

```
LR1 0x8000 (2 * 1024)
{
    ER1 +0 (1 * 1024)
    {
        *(+RO)
    }
    ER2 +0 (1 * 1024)
    {
        *(+RW +ZI)
    }
}
```

Example 3-6 Placing an execution region after another

```
LR1 0x8000
{
    ER1 0x100000
    {
        *(+RO)
    }
}
LR2 0x100000
{
    ER2 (ImageLimit(ER1))               ; Place ER2 after ER1 has finished
    {
        *(+RW +ZI)
    }
}
```

Example 3-7 Conditionalizing a base address based on the presence of a symbol

```
LR1 0x8000
{
    ER1 (defined(version1) ? 0x8000 : 0x10000)   ; Base address is 0x8000
                                                 ; if version1 is defined
```

```
                                                    ; 0x10000 if not
    {
        *(+RO)
    }
    ER2 +0
    {
        *(+RW +ZI)
    }
}
```

A combination of pre-processor macros and expressions is used in Example 3-8 to copy tightly packed execution regions to execution addresses in a page-boundary. Using the ALIGN scatter loading keyword aligns the load addresses of ER2 and ER3 as well as the execution addresses

Example 3-8 Aligning a base address in execution space but still tightly packed in load space

```
#! armcc -E
#DEFINE START_ADDRESS  0x100000
#DEFINE PAGE_ALIGNMENT 0x100000

#DEFINE MY_ALIGN(address, alignment) ((address +
(alignment-1)) AND ~(alignment-1))

LR1 0x8000
{
    ER0 +0
    {
        *(InRoot$$Sections)
    }
    ER1 START_ADDRESS
    {
        file1.o(*)
    }
    ER2 MY_ALIGN(ImageLimit(ER1), PAGE_ALIGNMENT)
    {
        file2.o(*)
    }
    ER3 MY_ALIGN(ImageLimit(ER2), PAGE_ALIGNMENT)
    {
        file3.o(*)
    }
}
```

Example 3-9 Using ScatterAssert to check the size of multiple regions

```
LR1 0x8000
{
    ER0 +0
    {
        *(+RO)
    }
    ER1 +0
    {
        file1.o(+RW)
    }
    ER2 +0
    {
        file2.o(+RW)
    }
    ScatterAssert((LoadLength(ER1) + LoadLength(ER2)) < 0x1000)
                                          ; LoadLength is compressed size
    ScatterAssert((ImageLength(ER1) + ImageLength(ER2)) < 0x2000)
                                          ; ImageLength is uncompressed size
}
ScatterAssert(ImageLength(LR1) < 0x3000) ; Check uncompressed size of LoadRegion
```

See also

- *Expression evaluation in scatter files* on page 3-30.

3.13 Scatter files containing relative base address load regions and a ZI execution region

You might want to place *Zero Initialized* (ZI) data in load region LR1, and use a relative base address for the next load region LR2, for example:

```
LR1 0x8000
{
    er_progbits +0
    {
        *(+RO,+RW) ; Takes space in the Load Region
    }
    er_zi +0
    {
        *(+ZI) ; Takes no space in the Load Region
    }
}
LR2 +0 ; Load Region follows immediately from LR1
{
    er_moreprogbits +0
    {
        file1.o(+RO) ; Takes space in the Load Region
    }
}
```

Because the linker does not adjust the base address of LR2 to account for ZI data, the execution region `er_zi` overlaps the execution region `er_moreprogbits`. This generates an error when linking.

To correct this, use the `ImageLimit()` function with the name of the ZI execution region to calculate the base address of LR2. For example:

```
LR1 0x8000
{
    er_progbits +0
    {
        *(+RO,+RW) ; Takes space in the Load Region
    }
    er_zi +0
    {
        *(+ZI) ; Takes no space in the Load Region
    }
}
LR2 ImageLimit(er_zi) ; Set the address of LR2 to limit of er_zi
{
    er_moreprogbits +0
    {
```

```
        file1.o(+RO) ; Takes space in the Load Region
    }
}
```

See also:

- *Syntax of a scatter-loading description file* on page 3-4
- *Execution address built-in functions for use in scatter files* on page 3-31
- *Expression evaluation in scatter files* on page 3-30
- *Region-related symbols* on page 4-4 in the *Linker User Guide*.

Chapter 4
BPABI and SysV Shared Libraries and Executables

This chapter describes how the ARM® linker, `armlink`, supports the *Base Platform Application Binary Interface* (BPABI) and *System V* (BPABI) and *System V* (SysV) shared libraries and executables. It contains the following sections:

4.1 About the BPABI

Many embedded systems use an operating system to manage the resources on a device. In many cases this is a large, single executable with a *Real Time Operating System* (RTOS) that tightly integrates with the applications. Other more complex *Operating Systems* (OS) are referred to as a platform OS, for example, ARM Linux. These have the ability to load applications and shared libraries on demand.

To run an application or use a shared library on a platform OS, you must conform to the ABI for the platform and also the ABI for the ARM architecture. This can involve substantial changes to the linker output, for example, a custom file format. To support such a wide variety of platforms, the ABI for the ARM architecture provides the BPABI.

The BPABI provides a base standard from which a platform ABI can be derived. The linker produces a BPABI conforming ELF image or shared library as an output. A platform specific tool called a post-linker translates this ELF output file into a file that can be loaded by the platform OS. Post linker tools are provided by the platform OS vendor. Figure 4-1 shows the BPABI tool flow.

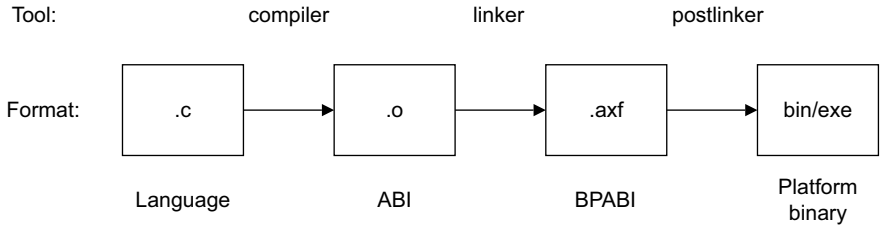

Figure 4-1 BPABI tool flow

See also:

- *Platforms supported by the BPABI* on page 4-3
- *Concepts common to all BPABI models* on page 4-5.

4.2 Platforms supported by the BPABI

The BPABI defines three platform models based on the type of shared library:

Bare metal The bare metal model is designed for an offline dynamic loader or a simple module loader. References between modules are resolved by the loader directly without any additional support structures.

DLL-like The DLL-like model sacrifices transparency between the dynamic and static library in return for better load and run-time efficiency.

SysV The SysV model masks the differences between dynamic and static libraries. ARM Linux uses this format.

See also:

- *Linker support for the BPABI*
- *Linker support for ARM Linux*.

4.2.1 Linker support for the BPABI

The ARM linker supports all three BPABI models enabling you to link a collection of objects and libraries into a:

- bare metal executable image
- BPABI DLL or SysV shared object
- BPABI or SysV executable file.

See also

- *--dll* on page 2-30
- *--shared* on page 2-80
- *--sysv* on page 2-90
- *About the BPABI* on page 4-2
- *Platforms supported by the BPABI*
- *Concepts common to all BPABI models* on page 4-5.

4.2.2 Linker support for ARM Linux

The linker can generate SysV executables and shared libraries with all required data for ARM Linux. However, you must specify other command-line options and libraries in addition to the `--shared` or `--sysv` options.

If all the correct input options and libraries are specified, you can use the ELF file without any post-processing.

See also

- *--dll* on page 2-30
- *--shared* on page 2-80
- *--sysv* on page 2-90
- *About the BPABI* on page 4-2
- *Platforms supported by the BPABI* on page 4-3
- *Concepts common to all BPABI models* on page 4-5.

4.3 Concepts common to all BPABI models

The linker enables you to build *Base Platform Application Binary Interface* (BPABI) shared libraries and to link objects against shared libraries. The following concepts are common to all BPABI models:

- symbol importing
- symbol exporting
- versioning
- visibility of symbols.

See also:

- *Importing and exporting symbols*
- *Symbol visibility*
- *Automatic import and export* on page 4-6
- *Manual import and export* on page 4-6
- *Symbol versioning* on page 4-7
- *RW compression* on page 4-7.

4.3.1 Importing and exporting symbols

In traditional linking, all symbols must be defined at link time for linking into a single executable file containing all the required code and data. In platforms that support dynamic linking, symbol binding can be delayed to load-time or in some cases, run-time. Therefore, the application can be split into a number of modules, where a module is either an executable or a shared library. Any symbols that are defined in modules other than the current module are placed in the dynamic symbol table. Any functions that are suitable for dynamically linking to at load or runtime are also listed in the dynamic symbol table.

There are two ways to control the contents of the dynamic symbol table:

- automatic rules that infer the contents from the ELF symbol visibility property
- manual directives that are present in a steering file.

These rules are slightly different for the SysV model. See *Automatic dynamic symbol table rules in SysV models* on page 4-8 for more information.

4.3.2 Symbol visibility

Each symbol has a visibility property. If the symbol is a reference, the visibility controls the definitions that the linker can use to define the symbol. If the symbol is a definition, the visibility controls whether the symbol can be made visible outside the current module.

The visibility options defined by the ELF specification are:

Table 4-1 Symbol visibility

Symbol	Reference	Definition
STV_DEFAULT	Symbol can be bound to a definition in a shared object.	Symbol can be made visible outside the module. It can be preempted by the dynamic linker by a definition from another module.
STV_PROTECTED	Symbol must be resolved within the module.	Symbol can be made visible outside the module. It cannot be preempted at run-time by a definition from another module.
STV_HIDDEN STV_INTERNAL	Symbol must be resolved within the module.	Symbol is not visible outside the module.

Symbol preemption is most common in SysV systems. Symbol preemption can happen in DLL-like implementations of the BPABI. The platform owner defines how this works. See the documentation for your specific platform for more information.

For more information on setting the symbol visibility for a symbol, see:

- *--use_definition_visibility* on page 2-95
- *EXPORT or GLOBAL* on page 7-78 in the *Assembler Guide*
- *--dllexport_all, --no_dllexport_all* on page 2-50 in the *Compiler Reference Guide*
- *--dllimport_runtime, --no_dllimport_runtime* on page 2-50 in the *Compiler Reference Guide*
- *--hide_all, --no_hide_all* on page 2-71 in the *Compiler Reference Guide.*

4.3.3 Automatic import and export

The linker can automatically import and export symbols. This behavior is dependent on a combination of the symbol visibility, if the output is an executable or a shared library, and if the platform model is SysV. This depends on what type of linking model is being used. See *Using SysV models* on page 4-8 and *Using bare metal and DLL-like models* on page 4-11 for more information.

4.3.4 Manual import and export

Linker steering files can be used to:

- manually control dynamic import and export
- override the automatic rules.

The steering file commands available to control the dynamic symbol table contents are:

- `EXPORT`
- `IMPORT`
- `REQUIRE`.

See also

- *EXPORT* on page 2-103
- *IMPORT* on page 2-106
- *REQUIRE* on page 2-110.

4.3.5 Symbol versioning

Symbol versioning provides a way to tightly control the interface of a shared library.

When a symbol is imported from a shared library that has versioned symbols, the linker binds to the most recent (default) version of the symbol. At load or run-time when the platform OS resolves the symbol version, it always resolves to the version selected by the linker, even if there is a more recent version available. This process is automatic.

When a symbol is exported from an executable or a shared library, it can be given a version. The linker supports implicit symbol versioning where the version is derived from the shared object name (set by `--soname`), or explicit symbol versioning where a script is used to precisely define the versions. See *Symbol versioning* on page 4-19 in the *Linker User Guide* for more information.

See also

- *--soname=name* on page 2-81.

4.3.6 RW compression

The decompressor for compressed RW data is tightly integrated into the start-up code in the ARM C library. When running an application on a platform OS, this functionality must provided by the platform or platform libraries. By default, RW compression is turned off when linking a BPABI or SysV file because there is no decompressor.

4.4 Using SysV models

The linker enables you to build and link SysV shared libraries. It also enables you to create SysV executables. This section describes dynamic symbol table rules, addressing modes, and thread local storage models.

Table 4-2 Turning on SysV support

Command-line options	Description
`--arm_linux`	this implies `--sysv`
`--sysv`	To produce a SysV executable
`--sysv --shared`	To produce a SysV shared library

4.4.1 SysV memory model

SysV files have a standard memory model that is described in the generic ELF specification. There are several platform operating systems that use the SysV format, for example, ARM Linux.

There are no configuration options available for the SysV memory model. The linker ignores any scatter file that you specify on the command-line and uses the standard memory map defined by the generic ELF specification.

4.4.2 Automatic dynamic symbol table rules in SysV models

The following rules apply:

Executable An undefined symbol reference is an undefined symbol error.

Global symbols with `STV_HIDDEN` or `STV_INTERNAL` visibility are never exported to the dynamic symbol table.

Global symbols with `STV_PROTECTED` or `STV_DEFAULT` visibility are not exported to the dynamic symbol table unless `--export_all` is set.

Shared library

An undefined symbol reference with `STV_DEFAULT` visibility is treated as imported and is placed in the dynamic symbol table.

An undefined symbol reference without `STV_DEFAULT` visibility is an undefined symbol error.

Global symbols with `STV_HIDDEN` or `STV_INTERNAL` visibility are never exported to the dynamic symbol table.

Note

STV_HIDDEN or STV_INTERNAL global symbols that are required for relocation can be placed in the dynamic symbol table, however the linker changes them into local symbols to prevent them from being accessed from outside the shared library.

Global symbols with STV_PROTECTED or STV_DEFAULT visibility are always exported to the dynamic symbol table.

Symbol definitions

To improve SysV compatibility with glibc, the linker defines these symbols if they are referenced:

- __init_array_start
- __init_array_end
- __fini_array_start
- __fini_array_end
- __exidx_start
- __exidx_end.

4.4.3 Addressing modes

SysV has a defined model for accessing the program and imported data and code. The linker automatically generates the required *Procedure Linkage Table* (PLT) and *Global Offset Table* (GOT) sections.

Position independent code

SysV shared libraries are almost always compiled with position independent code by using the --apcs=/fpic compiler command-line option. See *--apcs=qualifer...qualifier* on page 2-4 in the *Compiler Reference Guide* for more information.

The linker command-line option --fpic must also be used to declare that a shared library is position independent because this affects the construction of the PLT and GOT sections.

Note

By default, the linker produces an error message if the command-line option --shared is given without the --fpic options. If you must create a shared library that is not position independent, you can turn the error message off by using --diag_suppress=6403.

4.4.4 Thread local storage

The linker supports the ARM Linux thread local storage model. For more information on the linker implementation, see the *Addenda to, and Errata in, the ABI for the ARM Architecture* (ABI-addenda).

4.4.5 Related linker command-line options

The following linker command-line options relate to the SysV memory model:

- *--fpic* on page 2-42
- *--dynamic_debug* on page 2-30
- *--dynamic_linker=name* on page 2-30
- *--export_all, --no_export_all* on page 2-36
- *--linux_abitag=version_id* on page 2-56
- *--runpath=pathlist* on page 2-76
- *--shared* on page 2-80
- *--sysv* on page 2-90.

Changes to command-line defaults

The ARM RealView® Compilation Tools tool chain does not provide shared libraries containing the C and C++ system libraries. The intended usage model of the SysV support is to use the system libraries that come with the platform. For example, in ARM Linux this is `libc.so`.

In order to use `libc.so`, the linker applies the following changes to the default behavior:

- `--arm_linux` sets the default options required for ARM Linux
- `--no_ref_cpp_init` is set to prevent the inclusion of the RealView Compilation Tools C++ initialization code
- The linker defines the required symbols to ensure compatibility with `libc.so`
- `--force_so_throw` is set which forces the linker to keep exception tables.

4.5 Using bare metal and DLL-like models

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

Table 4-3 Turning on BPABI support

Command-line options	Description
`--base_platform`	to use scatter-loading with *Base Platform ABI* (BPABI)
`--bpabi`	to produce a BPABI executable
`--bpabi --dll`	to produce a BPABI DLL

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

- bare metal and DLL-like memory model
- mandatory symbol versioning in the BPABI DLL-like model
- automatic dynamic symbol table rules in the BPABI DLL-like model
- addressing modes in the BPABI DLL-like model
- C++ initialization in the BPABI DLL-like model.

If you are implementing a platform OS, you must use this information in conjunction with the BPABI specification.

See also:

- *Bare metal and DLL-like memory model*
- *Mandatory symbol versioning in the BPABI DLL-like model* on page 4-13
- *Automatic dynamic symbol table rules in the BPABI DLL-like model* on page 4-13
- *Addressing modes in the BPABI DLL-like model* on page 4-14
- *C++ initialization in the BPABI DLL-like model* on page 4-14
- *Related linker command-line options for the BPABI DLL-like model* on page 4-14.

4.5.1 Bare metal and DLL-like memory model

BPABI files have a standard memory model that is described in the BPABI specification. By using the `--bpabi` comand-line option, the linker automatically applies this model and ignores any scatter-loading description file that you specify on the command-line. This is equivalent to the following image layout:

```
LR_1 <read-only base address>
{
    ER_RO  +0
    {
        *(+RO)
    }
}
LR_2 <read-write base address>
{
    ER_RW  +0
    {
        *(+RW)
    }
    ER_ZI  +0
    {
        *(+ZI)
    }
}
```

Customizing the memory model

If the option `--ropi` is specified, `LR_1` is marked as position-independent. Likewise, if the option `--rwpi` is specified, `LR_2` is marked as position-independent.

Note

In most cases, you need to specify the `--ro_base` and `--rw_base` switches, because the default values, `0x8000` and `0` respectively, might not be suitable for your platform. These addresses need not reflect the actual addresses to which the image is relocated at run time.

If you require a more complicated memory layout, use the Base Platform linking model, `--base_platform`.

See also:

- *--base_platform* on page 2-14
- *--ropi* on page 2-75
- *--ro_base=address* on page 2-74
- *--rosplit* on page 2-75
- *--rwpi* on page 2-77
- *--rw_base=address* on page 2-76.

4.5.2 Mandatory symbol versioning in the BPABI DLL-like model

The BPABI DLL-like model requires static binding. This is because a post-linker might translate the symbolic information in a BPABI DLL to an import or export table that is indexed by an ordinal. In which case, it is not possible to search for a symbol at run-time.

Static binding is enforced in the BPABI with the use of symbol versioning. The command-line option `--symver_soname` is on by default for BPABI files, this means that all exported symbols are given a version based on the name of the DLL.

See also:

- *--soname=name* on page 2-81
- *--symver_script=file* on page 2-90
- *--symver_soname* on page 2-90
- *Symbol versioning* on page 4-19 in the *Linker User Guide*.

4.5.3 Automatic dynamic symbol table rules in the BPABI DLL-like model

This section describes the symbol table rules for executables and DLLs.

Executable

An undefined symbol reference is an undefined symbol error.

Global symbols with `STV_HIDDEN` or `STV_INTERNAL` visibility are never exported to the dynamic symbol table.

Global symbols with `STV_PROTECTED` or `STV_DEFAULT` visibility are not exported to the dynamic symbol table unless `--export_all` is set. See *--export_all, --no_export_all* on page 2-36 for more information.

DLL

An undefined symbol reference is an undefined symbol error.

Global symbols with `STV_HIDDEN` or `STV_INTERNAL` visibility are never exported to the dynamic symbol table.

Note

`STV_HIDDEN` or `STV_INTERNAL` global symbols that are required for relocation can be placed in the dynamic symbol table, however the linker changes them into local symbols to prevent them from being accessed from outside the shared library.

Global symbols with STV_PROTECTED or STV_DEFAULT visibility are always exported to the dynamic symbol table.

4.5.4 Addressing modes in the BPABI DLL-like model

The main difference between the bare metal and DLL-like BPABI models is the addressing mode used to access imported and own-program code and data. There are four options available that correspond to categories in the BPABI specification:

- None
- Direct references
- Indirect references
- Relative static base address references.

Selection of the required addressing mode is controlled by the following command-line options:

- *--pltgot=type* on page 2-65
- *--pltgot_opts=mode* on page 2-66.

4.5.5 C++ initialization in the BPABI DLL-like model

A DLL supports the initialization of static constructors via the use of a table of initializers that are given a special section type of SHT_INIT_ARRAY. These initializers contain a relocation of type R_ARM_TARGET1 to a symbol that performs the initialization.

The ELF ABI specification describes R_ARM_TARGET1 as either a relative form, or an absolute form.

The ARM C libraries use the relative form. For example, if the linker detects a definition of the ARM C library __cpp_initialize_aeabi, it uses the relative form of R_ARM_TARGET1 otherwise it uses the absolute form.

4.5.6 Related linker command-line options for the BPABI DLL-like model

The following linker command-line options relate to the *Base Platform Application Binary Interface* (BPABI) DLL-like model:

- *--base_platform* on page 2-14
- *--bpabi* on page 2-17
- *--dll* on page 2-30
- *--dynamic_debug* on page 2-30
- *--export_all, --no_export_all* on page 2-36
- *--pltgot=type* on page 2-65
- *--pltgot_opts=mode* on page 2-66

- --*ropi* on page 2-75
- --*ro_base=address* on page 2-74
- --*rosplit* on page 2-75
- --*runpath=pathlist* on page 2-76
- --*rw_base=address* on page 2-76
- --*rwpi* on page 2-77
- --*soname=name* on page 2-81
- --*symver_script=file* on page 2-90
- --*symver_soname* on page 2-90.

Chapter 5
Features of the Base Platform linking model

This chapter describes features of the Base Platform linking model supported by the ARM linker, `armlink`. It contains:

- *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
- *Placement of PLT sequences with the Base Platform model* on page 5-7.

5.1 Restrictions on the use of scatter files with the Base Platform model

The Base Platform model supports scatter files. Although there are no restrictions on the keywords you can use in a scatter file, there are restrictions on the types of scatter files you can use:

- A load region marked with the RELOC attribute must contain only execution regions with a relative base address of *+offset*. The following examples show valid and invalid scatter files using the RELOC attribute and *+offset* relative base address:

Example 5-1 Valid scatter file example using RELOC and *+offset*

```
# This is valid. All execution regions have +offset addresses.
LR1 0x8000 RELOC
{
    ER_RELATIVE +0
    {
        *(+RO)
    }
}
```

Example 5-2 Invalid scatter file example using RELOC and *+offset*

```
# This is not valid. One execution region has an absolute base address.
LR1 0x8000 RELOC
{
    ER_RELATIVE +0
    {
        *(+RO)
    }
    ER_ABSOLUTE 0x1000
    {
        *(+RW)
    }
}
```

- Any load region that requires a PLT section must contain at least one execution region containing code, that is not marked OVERLAY. This execution region is used to hold the PLT section. An OVERLAY region cannot be used as the PLT must remain in memory at all times. The following examples show valid and invalid scatter files that define execution regions requiring a PLT section:

Example 5-3 Valid scatter file example for a load region that requires a PLT section

```
# This is valid. ER_1 contains code and is not OVERLAY.
LR_NEEDING_PLT 0x8000
{
    ER_1 +0
    {
        *(+RO)
    }
}
```

Example 5-4 Invalid scatter file example for a load region that requires a PLT section

```
# This is not valid. All execution regions containing code are marked OVERLAY.
LR_NEEDING_PLT 0x8000
{
    ER_1 +0 OVERLAY
    {
        *(+RO)
    }
    ER_2 +0
    {
        *(+RW)
    }
}
```

- If a load region requires a PLT section, then the PLT section must be placed within the load region. By default, if a load region requires a PLT section, the linker places the PLT section in the first execution region containing code. You can override this choice with a scatter loading selector.

  If there is more than one load region containing code, the PLT section for a load region with name *name* is `.plt_`*name*. If there is only one load region containing code, the PLT section is called `.plt`.

  The following examples show valid and invalid scatter files that place a PLT section:

Example 5-5 Valid scatter file example for placing a PLT section

```
#This is valid. The PLT section for LR1 is placed in LR1.
LR1 0x8000
{
    ER1 +0
```

```
    {
        *(+RO)
    }
    ER2 +0
    {
        *(.plt_LR1)
    }
}
LR2 0x10000
{
    ER1 +0
    {
        *(other_code)
    }
}
```

Example 5-6 Invalid scatter file example for placing a PLT section

```
#This is not valid. The PLT section of LR1 has been placed in LR2.
LR1 0x8000
{
    ER1 +0
    {
        *(+RO)
    }
}
LR2 0x10000
{
    ER1 +0
    {
        *(.plt_LR1)
    }
}
```

See also:

- *Load region description* on page 3-5
- *Execution region description* on page 3-8
- *Addressing attributes* on page 3-13
- *Placement of PLT sequences with the Base Platform model* on page 5-7
- the following in the *Linker User Guide*:
  - *Base Platform linking model* on page 2-7.

5.2 Example scatter file for the Base Platform linking model

This example shows the use of a scatter file with the Base Platform linking model.

The standard *Base Platform Application Binary Interface* (BPABI) memory model in scatter-file format, with relocatable load regions is:

Example 5-7 Standard BPABI scatter file with relocatable load regions

```
LR1 0x8000 RELOC
{
    ER_RO +0
    {
        *(+RO)
    }
}

LR2 0x0 RELOC
{
    ER_RW +0
    {
        *(+RW)
    }
    ER_ZI +0
    {
        *(+ZI)
    }
}
```

This example conforms to the BPABI, because it has the same two-region format as the BPABI specification.

The next example shows two load regions LR1 and LR2 that are not relocatable.

Example 5-8 Scatter file with some load regions that are not relocatable

```
LR1 0x8000
{
    ER_RO +0
    {
        *(+RO)
    }
    ER_RW +0
    {
        *(+RW)
```

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

LR2 0x10000
{
    ER_KNOWN_ADDRESS +0
    {
        *(fixedsection)
    }
}

LR3 0x20000 RELOC
{
    ER_RELOCATABLE +0
    {
        *(floatingsection)
    }
}
```

The linker does not have to generate dynamic relocations between LR1 and LR2 because they have fixed addresses. However, the `RELOC` load region LR3 might be widely separated from load regions LR1 and LR2 in the address space. Therefore, dynamic relocations are required between LR1 and LR3, and LR2 and LR3.

Use the options `--pltgot=direct --pltgot_opts=crosslr` to ensure a PLT is generated for each load region.

See also:

- *--pltgot=type* on page 2-65
- *--pltgot_opts=mode* on page 2-66
- *Load region description* on page 3-5
- *Restrictions on the use of scatter files with the Base Platform model* on page 5-2
- the following in the *Linker User Guide*:
  - *Base Platform Application Binary Interface (BPABI) linking model* on page 2-6
  - *Base Platform linking model* on page 2-7
  - *Concepts common to both BPABI and SysV linking models* on page 2-5.

5.3 Placement of PLT sequences with the Base Platform model

The linker supports *Procedure Linkage Table* (PLT) generation for multiple load regions containing code when in Base Platform mode (`--base_platform`).

To turn on PLT generation when in Base Platform mode use `--pltgot=`*`option`* `that generates PLT sequences`. You can use the option `--pltgot_opts=crosslr` to add entries in the PLT for calls between `RELOC` load-regions. PLT generation for multiple Load Regions is only supported for `--pltgot=direct`.

The `--pltgot_opts=crosslr` option is useful when you have multiple load regions that might be moved relative to each other when the image is dynamically loaded. The linker generates a PLT for each load region so that calls do not have to be extended to reach a distant PLT.

Placement of linker generated PLT sections:

- When there is only one load region there is one PLT. The linker creates a section called `.plt` with an object `anon$$obj.o`.
- When there are multiple load regions, a PLT section is created for each load region that requires one. By default, the linker places the PLT section in the first execution region containing code. You can override this by specifying the exact PLT section name in the scatter-file.

  For example, a load region with name *`LR Name`* the PLT section is called `.plt_`*`LR_NAME`* with an object of `anon$$obj.o`. To precisely name this PLT section in a scatter file, use the selector:

  `anon$$obj.o(.plt_`*`LR_NAME`*`)`

Be aware of the following:

- The linker gives an error message if the PLT for load region *`LR_NAME`* is moved out of load region *`LR_NAME`*.
- The linker gives an error message if load region *`LR_NAME`* contains a mixture of `RELOC` and non-`RELOC` execution regions. This is because it cannot guarantee that the `RELOC` execution regions are able to reach the PLT at run-time.
- `--pltgot=indirect` and `--pltgot=sbrel` are not supported for multiple load regions.

See also:

- *--base_platform* on page 2-14
- *--pltgot=type* on page 2-65
- *--pltgot_opts=mode* on page 2-66

- the following in the *Linker User Guide*:
  - *Base Platform linking model* on page 2-7.