RealView® Compilation Tools

Version 3.1

Linker and Utilities Guide

ARM®

ARM DUI 0206H

RealView Compilation Tools

Linker and Utilities Guide

Release Information

The following changes have been made to this book.

Change History

Date	Issue	Confidentiality	Change
August 2002	A	Non-Confidential	Release 1.2
January 2003	B	Non-Confidential	Release 2.0
September 2003	C	Non-Confidential	Release 2.0.1 for RVDS v2.0
January 2004	D	Non-Confidential	Release 2.1 for RVDS v2.1
December 2004	E	Non-Confidential	Release 2.2 for RVDS v2.2
May 2005	F	Non-Confidential	Release 2.2 for RVDS v2.2 SP1
March 2006	G	Non-Confidential	Release 3.0 for RVDS v3.0
March 2007	H	Non-Confidential	Release 3.1 for RVDS v3.1

Proprietary Notice

Words and logos marked with ® or ™ are registered trademarks or trademarks owned by ARM Limited. Other brands and names mentioned herein may be the trademarks of their respective owners.

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

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

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

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

Confidentiality Status

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

Product Status

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

Web Address

`http://www.arm.com`

Contents
RealView Compilation Tools Linker and Utilities Guide

Chapter 4 Accessing Image Symbols

Chapter 5 Using Scatter-loading Description Files

Chapter 6 BPABI and System V Shared Libraries and Executables

Chapter 7 Using the ARM Librarian

Chapter 8 Using fromelf

Preface

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

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

About this book

This book describes the ARM linker, `armlink`, the image conversion utility, `fromelf`, and the ARM librarian, `armar`, provided with *RealView Compilation Tools* (RVCT).

Intended audience

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

Using this book

This book is organized into the following chapters:

Chapter 1 ***Introduction***

Read this chapter for an introduction to the linker and related utilities in RVCT.

Chapter 2 ***The Linker Command Syntax***

Read this chapter for an explanation of all command-line options accepted by the linker.

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

Read this chapter for information on using linker features and how to create simple images.

Chapter 4 ***Accessing Image Symbols***

Read this chapter for information on accessing symbols in images.

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

Read this chapter for information on using a scatter-loading file to place code and data in memory.

Chapter 6 ***BPABI and System V Shared Libraries and Executables***

Read this chapter for information on using BPABI and System V shared libraries and executables.

Chapter 7 ***Using the ARM Librarian***

Read this chapter for an explanation of the procedures involved in creating and accessing library objects.

Chapter 8 ***Using fromelf***

Read this chapter for a description of the `fromelf` utility and how you can use it to change image format.

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

Typographical conventions

The following typographical conventions are used in this book:

`monospace`	Denotes text that can be entered at the keyboard, such as commands, file and program names, and source code.
`monospace` (underlined `mono`)	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://www.arm.com` for current errata sheets and addenda, and the ARM Frequently Asked Questions.

ARM publications

This book contains reference information that is specific to development tools supplied with RVCT. Other publications included in the suite are:

- *RVCT Essentials Guide* (ARM DUI 0202)
- *RVCT Compiler User Guide* (ARM DUI 0205)
- *RVCT Compiler Reference Guide* (ARM DUI 0348)
- *RVCT Libraries and Floating Point Support Guide* (ARM DUI 0349)
- *RVCT Assembler Guide* (ARM DUI 0204)
- *RVCT Developer Guide* (ARM DUI 0203)
- *NEON Vectorizing Compiler Guide* (ARM DUI 0350)
- *RealView Development Suite Glossary* (ARM DUI 0324).

For full information about the base standard, software interfaces, and standards supported by ARM, see *`install_directory`*`\Documentation\Specifications\...`.

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

- *ARM6-M Architecture Reference Manual* (ARM DDI 0419)

- *ARM7-M Architecture Reference Manual* (ARM DDI 0403)

- *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition* (ARM DDI 0406)

- ARM datasheet or technical reference manual for your hardware device.

Feedback

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

Feedback on RealView Compilation Tools

If you have any problems with RVCT, 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`, and the utility programs, `armar` and `fromelf`, provided with *RealView® Compilation Tools* (RVCT). It contains the following sections:

- *armlink* on page 1-2
- *armar* on page 1-3
- *fromelf* on page 1-4
- *Compatibility with legacy objects and libraries* on page 1-5.

1.1 armlink

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 *Executable and Linking Format* (ELF) executable image
- a partially linked ELF object that can be used as input to 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 (SVr4) specification, or a BPABI or SVr4 executable file.

See Chapter 2 *The Linker Command Syntax* for more information on the ARM linker and all command-line options.

1.2 armar

The ARM librarian, `armar`, enables you to collect and maintain sets of ELF files in standard format `ar` libraries. You can pass libraries to the linker in place of several ELF object files.

See *The ARM librarian* on page 7-7 for more information.

1.3 fromelf

`fromelf` is the ARM image conversion utility. It accepts ELF format input files and converts them to a variety of output formats, including:

- plain binary
- Motorola 32-bit S-record format
- Intel Hex-32 format
- Byte Oriented (Verilog Memory Model) Hex format.

`fromelf` can also produce textual information about the input file and disassemble code.

See Chapter 8 *Using fromelf* for more information.

1.4 Compatibility with legacy objects and libraries

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

Chapter 2

The Linker Command Syntax

This chapter describes the full command syntax for the ARM linker, `armlink`, provided with RVCT. It contains the following sections:

- *About armlink* on page 2-2
- *armlink command syntax* on page 2-5.

2.1 About armlink

The linker automatically selects the appropriate standard C or C++ library variants to link with, based on the build attributes of the objects it is linking.

The linker can link ARM code, Thumb® code, and Thumb-2 code, and automatically generates interworking veneers to switch processor state when required. The linker also generates inline veneers or long branch veneers, where required, to extend the range of branch instructions.

The linker supports command-line options that enable you to specify the locations of code and data within the system memory map. Alternatively, you can use scatter-loading description files to specify the memory locations, at both load and execution time, of individual code and data sections in your output image. This enables you to create complex images spanning multiple memories.

The linker supports Read/Write data compression to minimize ROM size.

The linker can perform unused section elimination to reduce the size of your output image. In addition, the linker enables you to:

- produce debug and reference information about linked files
- generate a static callgraph and list the stack usage over it
- control the contents of the symbol table in output images
- show the sizes of code and data in the output.

The linker can provide feedback, for the next time a file is compiled, to inform the compiler about unused functions. These are placed in their own sections on subsequent compilations for future elimination by the linker.

2.1.1 Input to armlink

Input to `armlink` consists of:

- One or more object files in ELF object format. This format is described in the ARM ELF specification.
- One or more libraries created by `armar` as described in Chapter 7 *Using the ARM Librarian*.
- A symbol definitions file.

2.1.2 Output from armlink

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

- an executable image in ELF executable format

- a shared object in ELF shared object format
- a partially-linked object in ELF object format
- a relocatable object in ELF object format.

For simple images, ELF executable files contain segments that are approximately equivalent to RO and RW output sections in the image. An ELF executable file also has ELF sections that contain the image output sections.

You can use `fromelf` to convert an executable image in ELF executable format to other file formats. See Chapter 8 *Using fromelf* for more information.

Constructing an executable image

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

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

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

Constructing a partially-linked object

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

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

Note

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

2.1.3 Ordering command-line options

In general, command-line options can appear in any order in a single linker invocation. However, the effects of some options depend on how they are combined with other related options. For example, the `--scatter` option cannot be used with any of the memory map options. See *Specifying memory map information for the image* on page 2-12.

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

See *Specifying an input file list* on page 2-6 for more information on the ordering priority of input files.

2.1.4 Specifying command-line options with an environment variable

You can specify command-line options by setting the value of the `RVCT31_LINKOPT` environment variable. The syntax is identical to the command line syntax. The linker reads the value of `RVCT31_LINKOPT` and inserts it at the front of the command string. This means that options specified in `RVCT31_LINKOPT` can be overridden by arguments on the command-line.

2.2 armlink command syntax

This section describes the syntax and options of the `armlink` command.

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

Single-letter options

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

```
-o file
```

Keyword options

All keyword options, or keyword options with arguments, are preceded by a double dash `--`. For example:

```
--output file
```

Note

For command-line arguments that use parentheses, you might have to escape the parentheses characters with a backlashes (\) character on UNIX systems.

The linker command syntax is:

```
armlink [help-options] [project-template-options] [input-file-list]
[linker-control-options] [output-format] [memory-map-options] [debuq-options]
[image-contents-options] [veneer-generation-options] [Byte Addressing mode]
[image-info-options] [diagnostics-options] [via-file]
```

2.2.1 Accessing help and information

To get information on the available command-line options and the tool version number use:

`--help`	Prints a summary of some commonly used command-line options.
`--vsn`	Displays the linker version information and license information.
`--show_cmdline`	Shows how the linker has processed the command line. The commands are shown in their preferred form, and the contents of any via files are expanded.

2.2.2 Project template options

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

`--[no_]project=[`*`filename`*`]`

Enables or disables the use of a project template file.

`--reinitialize_workdir`

Enables you to reinitialize the project template working directory.

`--workdir=`*`directory`*

Enables you to provide a working directory for a project template.

For more information on each of these options, see:

- *--[no_]project=filename* on page 2-74 in the *Compiler Reference Guide*
- *--reinitialize_workdir* on page 2-77 in the *Compiler Reference Guide*
- *--workdir=directory* on page 2-95 in the *Compiler Reference Guide*.

2.2.3 Specifying an input file list

The content of all input files must be unique. If symbols are multiply defined the linker produces an error message.

These options define the input files passed to the linker:

`input-file-list`

This is a space-separated list of objects, libraries, or *symbol definitions* (symdefs) files. 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. See *Accessing symbols in another image* on page 4-8 for more information.

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. For example, specify `mystring.lib(strcmp.o)` in the input file list.

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, the following command unconditionally adds all `a*.o` objects and `stdio.o` from `mylib`:

   ```
   armlink main.o mylib(stdio.o) mylib(a*.o)
   ```

   On UNIX platforms you might need to escape the parentheses, for example:

   ```
   armlink main.o mylib\(stdio.o\)
   ```

3. The standard C or C++ libraries are added to the list of libraries that are later used to resolve any remaining references.

For more information see *Library searching, selection, and scanning* on page 7-3.

`--libpath` *`pathlist`*

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 `RVCT31LIB` environment variable. Any paths specified here override the path specified by `RVCT31LIB`.

`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`*.

This list must end with the parent directory of the ARM library directories `armlib` and `cpplib`.

Note

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

See *Library searching, selection, and scanning* on page 7-3 for more information on including libraries.

`--library_type=`*`lib`*

Enables the relevant library selection to be used at link time.

Where *`lib`* can be one of:

`standardlib`	Specifies that the full RVCT runtime libraries are selected at link time. This is the default.
`microlib`	Specifies that the C micro-library (microlib) is selected at link time.

Note

This option can be used with the compiler, assembler or linker when use of the libraries require more specialized optimizations.

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

For more information see:

- *Building an application with microlib* on page 3-4 in the *Libraries Guide*
- *--library_type=lib* on page 2-54 in the *Compiler Reference Guide*.

`--[no_]reduce_paths`

Enables or disables the elimination of redundant pathname information in file paths. This option is valid for Windows systems only.

Windows systems impose a 260 character limit on file paths. Where relative pathnames exist whose absolute names expand to longer than 260 characters, you can use the `--reduce_paths` option to reduce absolute pathname 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.

See *--[no_]reduce_paths* on page 2-76 in the *Compiler Reference Guide* for more information.

`--[no_]scanlib`

Enables or disables scanning of default libraries (the standard ARM C and C++ libraries) to resolve references. `--scanlib` is the default.

`--userlibpath` *`pathlist`*

Specifies a list of paths that are used to search for user libraries.

`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 *Library searching, selection, and scanning* on page 7-3 for more information on including user libraries.

2.2.4 Controlling linker behavior

These options control how objects are linked:

`--match crossmangled`

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 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.

`--strict` 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.

`--[no_]strict_relocations`

`--strict_relocation` instructs 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.

This option enables you to ensure ABI compliance of objects. It is off by default, and deprecated and obsolete relocations are handled silently by the linker, where possible.

`--unresolved` *`symbol`*

Matches each reference to an undefined symbol to the global definition of *`symbol`*. *`symbol`* must be both defined and global, otherwise it appears in the list of undefined symbols and the link step fails. 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.

2.2.5 Specifying the output type and the output filename

Specify the format and the name of the output file using the following options:

`--bpabi` Creates a BPABI executable for passing to a platform-specific post-linker.

`--dll` Creates a BPABI *Dynamically Linked Library* (DLL). The DLL is marked as a shared object in the ELF file header.

`--output` *`file`*

Specifies the name of the output file. The file can be either a partially-linked object or an executable image. If the output filename is not specified, the linker uses the following defaults:

`__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.

`--partial` Creates a partially-linked object instead of an executable image.

`--reloc` Creates *relocatable* ELF images.

A relocatable image has a dynamic segment that contains relocations that can be used to relocate the image post link-time. Examples of post link-time relocation include advanced ROM construction and dynamic loading at runtime.

If the image is loaded at its link-time address, the relocatable image produced by the linker does not require the relocations to be processed and debug data for the image is valid. Loading the image at a different address to the link-time address and processing the relocations, however, invalidates any debug data present in the image.

Used on its own, `--reloc` makes an image similar to Simple type 1 where the load region attribute is set to `RELOC`. See *Type 1, one load region and contiguous execution regions* on page 3-26 for more information.

`--shared` Creates an SVr4 shared object.

See Chapter 6 *BPABI and System V Shared Libraries and Executables* for more information.

`--sysv` Creates an SVr4 formatted ELF executable file that can be used on ARM Linux.

See Chapter 6 *BPABI and System V Shared Libraries and Executables* for more information.

2.2.6 Specifying memory map information for the image

Use the following options to specify memory maps:

`--[no_]autoat`

Controls the automatic assignment of `__at` sections to execution regions. `__at` sections are sections that must be placed at a specific address. See *Using __at sections to place sections at a specific address* on page 5-37 for more information.

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. See *Input section description* on page 5-19 for more information.

The default is `--no_autoat`.

`--fpic` Enables you to link Position-Independent Code (PIC), that is, code that has been compiled using the `/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.

`--predefine="`*`string`*`"`

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`*`"`.

More than one `--predefine` option can be used on the command-line. Use this option with `--scatter`. Example 2-1 shows how the scatter file looks before pre-processing.

Example 2-1 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 2-2 shows how the scatter file looks after pre-processing:

Example 2-2 Scatter file after pre-processing

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

`--ro-base` *address*

Sets both the load and execution addresses of the region containing the RO output section at *address*. *address* must be word-aligned. If this option is not specified, and no scatter-load file is specified, the default RO base address is `0x8000`.

`--ropi` Makes the load and execution region containing the RO output section position-independent. If this option is not used, the region is marked as absolute. Usually each read-only input section must be read-only position-independent (ROPI). If this option is selected, the linker:

- checks that relocations between sections are valid
- ensures that any code generated by `armlink` itself, such as interworking veneers, is read-only position-independent.

Note

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

`--rosplit` Splits the default RO load region into two RO output sections, one for `RO-CODE` and one for `RO-DATA`.

`--rw-base` *`address`*

Sets the execution addresses of the region containing the RW output section at *`address`*. *`address`* must be word-aligned.

`--rwpi` 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.

`--scatter` *`file`*

Creates the image memory map using the scatter-loading description contained in *`file`*. The description provides grouping and placement details of the various regions and sections in the image. See Chapter 5 *Using Scatter-loading Description Files*.

The `--scatter` option cannot be used with any of the memory map options `--partial`,`--ro-base`, `--rw-base`, `--ropi`, `--rwpi`, `--rosplit`, or `--split`, and with `--reloc`, and `--startup`.

`--split` 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. See *Specifying the image structure* on page 3-2 for more information.

2.2.7 Controlling debug information

These options control debug information in the image:

`--[no_]debug` Use `--debug` to include debug information in the output file. The debug information includes debug input sections and the symbol and string table. This is the default.

Use `--no_debug` to exclude debug information from the output file. The ELF image is smaller, but you cannot debug it at the 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 creating a partially-linked object rather than an image, the linker discards the debug input sections it finds in the input objects, but does produce the symbol and string table in the partially-linked object.

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.

`--[no_]bestdebug`

Selects between linking for smallest code/data size or best debug illusion. Input objects might contain comdat groups, but these might not be identical across all input objects because of differences. For example, inlining.

`--no_bestdebug` is the default. 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.

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 *Common group or section elimination* on page 3-11 for more information.

`--[no_]compress_debug`

`--compress_debug` suggests to the linker to compress `.debug_*` sections, if it is sensible to do so. This removes some redundancy and reduce 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.

`--no_compress_debug` is the default.

`--[no_]dynamic_debug`

`--dynamic_debug` 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 Chapter 6 *BPABI and System V Shared Libraries and Executables* for more information.

2.2.8 Controlling image contents

These options control miscellaneous factors affecting the image contents:

`--[no_]branchnop`

The `--branchnop` option causes the linker to replace any branch with a relocation that resolves to the next instruction with a `NOP`. This is the default.

Use `--no_branchnop` to disable this behavior.

See *Branch inlining* on page 3-22 for more information on controlling branch inlining.

`--[no_]cppinit` *`symbol`*

`The --cppinit` *`symbol`* option enables the linker to use alternative C++ libraries with a different initialization symbol. By default, *`symbol`* is set to: `__cpp_initialize__aeabi_`

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

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

`--cpu=`*`name`*

Enables code generation for a specific ARM processor or architecture.

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.

This option follows the same format as that supported by the compiler. See *--cpu=name* on page 2-15 in the *Compiler Reference Guide* for more information.

`--datacompressor on|off`

RW data compression is enabled by default to minimize ROM size. Use `--datacompressor off` to turn off RW data compression.

`--datacompressor list|`*`id`*

Enable you to specify one of the supplied algorithms for RW data compression:

- Use `--datacompressor list` to get a list of data compressors available to the linker.
- 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. For more information see *RW data compression* on page 3-17.

  If you do want to override the linker, use `--datacompressor` *`id`* to specify a data compression algorithm. Specifying a compressor adds a decompressor to the code area. If the final image does not have compressed data, the decompressor is not added.

`--dynamiclinker` *`name`*

Specifies the dynamic linker used to load and relocate the file at runtime. When you link with shared objects, the dynamic linker uses dependency information stored in the executable to identify the files to load. If you are working on ARM Linux platforms, the linker assumes that the default dynamic linker is `/lib/ld-linux.so.3`.

See Chapter 6 *BPABI and System V Shared Libraries and Executables* for more information.

`--edit` *`file-list`*

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.

See *Hiding and renaming global symbols* on page 4-11 for more information on steering file syntax.

When you are specifying more than one steering file, the syntax can be either of the following:

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

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

Do not include spaces between the comma and the filenames.

`--entry` *`location`*

Specifies the unique initial entry point of the image. 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. The ARM 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).

Replace *`location`* with 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 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`.

`--[no_]exceptions`

Use `--exceptions` to enable the final image to contain exception tables. This is the default.

Use `--no_exceptions` to force the linker to generate 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 *Using command-line options to handle C++ exceptions* on page 3-32 for more information.

`--exceptions_tables=`*`action`*

Specifies how exception tables are generated for objects that do not already contain exception unwinding tables. Replace *action* with one of the following:

`nocreate`	The linker does not create missing exception tables. This is the default.
`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.

See *Using command-line options to handle C++ exceptions* on page 3-32 for more information.

`--[no_]export_all`

Use `--export_all` to dynamically export all global, non hidden symbols from the executable or DLL. This is the default for shared libraries and DLLs.

Use --no_export_all to prevent the exporting of symbols to the dynamic symbols table. This is the default for applications.

For more precise control over the exporting of symbols, use one or more steering files. If a steering file is used then this option has no effect. See *Hiding and renaming global symbols* on page 4-11 for more information.

`--fini` *`symbol`*

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 Chapter 6 *BPABI and System V Shared Libraries and Executables* for more information.

`--first` *`section-id`*

Places the selected input section first in its execution region. This can, for example, place the section containing the vector table first in the image. Replace *`section-id`* with 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, `armlink` generates an error message. For example: `--first init.o`

Note

When using scatter-loading, use `+FIRST` in the scatter-loading description file instead.

See *Ordering input sections by attribute* on page 3-9 for more information.

`--[no_]force_so_throw`

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

`--no_force_so_throw` is the default.

Use `--force_so_throw` option 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 Chapter 6 *BPABI and System V Shared Libraries and Executables* for more information.

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

`--fpu=`*`name`*

Enables code generation for a specific FPU architecture.

The link phase 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 follows the same format as that supported by the compiler. See *--fpu=name* on page 2-42 in the *Compiler Reference Guide* for more information.

`--init` *`symbol`*

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 Chapter 6 *BPABI and System V Shared Libraries and Executables* for more information.

`--[no_]inline`

`--inline` enables branch inlining to optimize small function calls in your image.

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 *Branch inlining* on page 3-22 for more information on controlling branch inlining.

`--keep` *`section-id`*

Specifies input sections that must not be removed by unused section elimination (see *Specifying an image memory map* on page 3-5).

All forms of the *`section-id`* argument can contain the `*` and `?` wildcards. You can specify multiple `--keep` options on the command line.

Replace *`section-id`* with 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`*, `armlink` 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`.

`--last` *`section-id`*

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. Replace *`section-id`* with 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.

Note

When using scatter-loading, use `+LAST` in the scatter-loading description file instead.

See *Ordering input sections by attribute* on page 3-9 for more information.

`--[no_]locals`

Use `--locals` to instruct the linker to add local symbols to the output symbol table when producing an executable image. This is the default.

Use `--no_locals` to instruct the linker not to add local symbols to the output symbol table. This is a useful optimization if you want to reduce the size of the output symbol table.

`--[no_]merge`

Instruct the linker to merge **`const`** strings that are placed in shareable sections by the compiler. This option can reduce the size of the image if there are similarities between **`const`** strings. The default is `--merge`.

`--linux_abitag` *`version-id`*

Enables you to specify the minimum compatible Linux kernel version for the executable file you are building.

See Chapter 6 *BPABI and System V Shared Libraries and Executables* for more information.

`--pad` *`num`*

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

`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.

`--pltgot=`*`type`*

Specifies the type of *Procedure Linkage Table* (PLT) and *Global Offset Table* (GOT) to use, corresponding to the different addressing modes of the BPABI.

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. This is the default when the options `--bpabi` or `--dll` are used.
`indirect`	The linker creates a GOT and possibly a PLT entry for the imported symbol. The reference refers to PLT or GOT entry.
`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.

`--pltgot_opts=`*`mode`*

Enables or disables weak references when generating PLT entries.

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

`weakrefs`	weak references produce a PLT entry. These references must be resolved at a later link stage.
`noweakrefs`	Generates a NOP, for a function call, or zero, for data. No PLT entry is generated. Weak references to imported symbols remain unresolved.

`--[no_]remove`

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. The default is `--remove`. See also *Unused section elimination* on page 3-12 for more information.

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

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

`--runpath` *`pathlist`*

Specifies a list of paths to be added to the search path that the Linux dynamic linker uses to search for Shared-Objects needed by this Image/Shared Object.

`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`*.

`--soname` *`name`*

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 executable file produced by the linker.

See Chapter 6 *BPABI and System V Shared Libraries and Executables* for more information.

`--[no_]startup` *`symbol`*

`--startup` *`symbol`* enables the linker to use alternative C libraries with a different startup symbol. By default, *`symbol`* is set to `__main`.

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

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 `--entry` *`location`* in *Controlling image contents* on page 2-16 for more information.

`--symver_script` *`file`*

Turns on implicit symbol versioning and enables you to specify *`file`* as a symbol version script.

See *Symbol versioning* on page 4-22 for more information.

`--symver_soname`

Turns on implicit symbol versioning and enables you to version symbols in order to force static binding. Where a symbol has no defined version, the linker uses the `SONAME` of the file being linked.

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 *Symbol versioning* on page 4-22 and the *Base Platform ABI for the ARM Architecture* (BPABI) for more information.

`--[no_]tailreorder`

`--tailreorder` 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. There are some restrictions to this option. The linker:

- can only move one tail calling section for each tail call target
- cannot move a tail calling section out of its execution region
- does not move tail calling sections before inline veneers.

See *Branch inlining* on page 3-22 for more information on handling tail calling sections.

`--no_tailreorder` is the default.

`--vfemode=`*`mode`*

Virtual Function Elimination (VFE) is a technique that enables the linker to identify more unused sections.

Use `--vfemode=`*`mode`* to specify how these, and *Runtime Type Information* (RTTI) objects, are eliminated.

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

- `on`
- `off`
- `force`
- `force_no_rtti`

See *Unused function elimination* on page 3-12 for more information.

2.2.9 Controlling veneer generation

These options control veneer generation:

`--[no_]inlineveneer`

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

`--inlineveneer` is the default.

`--[no_]piveneer`

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. See *PI to absolute veneers* on page 3-19 for more information.

`--piveneer` is the default.

`--[no_]veneershare`

Enables or disables veneer sharing. Veneer sharing can cause a significant decrease in image size.

`--veneershare` is the default.

See *Veneer generation* on page 3-20 for more information.

2.2.10 Specifying Byte Addressing mode

These options control Byte Addressing mode:

`--be8` 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.

`--be32` 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.

For more information on endian support, see:

- *Support for ARM architecture v6* on page 4-43 in the *Compiler User Guide*

- *ARM Architecture Reference Manual*.

2.2.11 Generating image-related information

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

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

`--[no_]callgraph`

`--callgraph` creates a static callgraph of functions. This is saved in the same directory as the generated image. The callgraph gives definition and reference information for all functions in the image.

Note

Any functions defined in the assembler files must have the appropriate `PROC`/`ENDP` and `FRAME PUSH`/`POP` directives if the linker is to calculate the function stack usage.

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).

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.

See the chapter describing the directives reference in the *Assembler Guide* for more information on how stack usage is determined.

`--callgraph_file=`*`filename`*

Controls the output name *`filename`* of the callgraph. For example:

`--callgraph_file=myfile`

The default *`filename`* is the same as the image. The output format is controlled by the `--callgraph_output=`*`fmt`* option.

`--callgraph_output=`*`fmt`*

Controls the output format of the callgraph, where *`fmt`* can be one of the following:

`html`	Outputs the callgraph in html format. This is the default.
`text`	Outputs the callgraph in plain text format.

`--cgfile=`*`type`*

Controls what is included in the callgraph, where *`type`* can be one of the following:

`all`	Includes symbols from all files. This is the default.
`user`	Includes only symbols from user defined objects and libraries.
`system`	Includes only symbols from system libraries.

`--cgsymbol=`*`type`*

Controls what is included in the callgraph, where *`type`* can be one of the following:

`all`	Includes both local and global symbols. This is the default.
`locals`	Includes only local symbols.
`globals`	Includes only global symbols.

`--cgundefined=`*`type`*

Controls what is included in the callgraph, where *`type`* can be one of the following:

`all`	Includes both function entries and calls to undefined weak references. This is the default.

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

`--feedback` *`file`*

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. For information on how to use the feedback file, see *Linker feedback* on page 3-14.

`--feedback_image` *`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. For 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.

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

`none`	Uses the scatter-loading file. 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-loading file. 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-loading 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. This is the default option.

`--info` *`topics`*

Prints information about specified topics, where *`topics`* is a comma-separated list of topic keywords. A topic keyword can be one of the following:

`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`.
`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. For more information on branch inlining see *Branch inlining* on page 3-22.
`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 a modifier, `--info_lib_prefix`, to display information about a specific library. For example, use the following options to identify the floating-point library used by the linker: `--info libraries --info_lib_prefix=f`.
`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`. For more information on handling tail calling sections see *Branch inlining* on page 3-22.
`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`.
`veneers`	Lists the linker-generated veneers. See *Veneer generation* on page 3-20 for more information.
`veneercallers`	Lists the linker-generated veneers with additional information about the callers to each veneer. Use with `--verbose` to list each call individually. See *Veneer generation* on page 3-20 for more information.

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 keywords in a list. For example, you can enter `--info sizes,totals` but not `--info sizes, totals`.

See *Getting information about images* on page 3-33 for more information.

`--[un]mangled`

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.

`--unmangled` is the default. This means that the linker unmangles C++ symbol names so that they are displayed as they appear in your source code.

If `--mangled` is selected, the linker does not unmangle C++ symbol names. Therefore, symbol names are displayed as they appear in the object symbol tables.

`--[no_]map` `--map` creates an image 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.

`--[no_]symbols`

`--symbols` lists 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.

`--[no_]list_mapping_symbols`

`--list_mapping_symbols` includes mapping symbols in the output produced by `--symbols`. For example:

$a	ARM code
$t	Thumb code
$d	data.

Mapping symbols are used to flag transitions between ARM code, Thumb code, and data. See *ELF for the ARM Architecture* (AAELF) for more information.

`--symdefs` *`file`*

Creates a file containing the global symbol definitions from the output image.

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.

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 *Accessing symbols in another image* on page 4-8 for more information.

`--[no_]xref` `--xref` lists all cross-references between input sections.

`--[no_]xrefdbg`

`--xrefdbg` lists all cross-references between input debug sections.

`--xref{from|to}` *`object(section)`*

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.

2.2.12 Controlling linker diagnostics

These options control how the linker emits diagnostics:

`--diag_style arm|ide|gnu`

Change the formatting of warning and error messages. `--diag_style arm` is the default, `--diag_style gnu` matches the format reported by `gcc`, and `--diag_style ide` matches the format reported by Microsoft Visual Studio.

`--diag_suppress` *`taglist`*

Disables all diagnostic messages that have the specified tag(s).

This option requires a comma-separated list of diagnostic message numbers that specifies the messages that must be suppressed. For example, to suppress the warning messages that have numbers `L6314W` and `L6305W`, use the following command:

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

`--diag_warning` *`taglist`*

Sets diagnostic messages that have the specified tag(s) to be displayed as warning messages, for example, where you want to downgrade an error message.

This option requires a comma-separated list of diagnostic message numbers that specifies the messages that must be downgraded.

`--errors` *`file`*

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 cleared.

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

`--list` *`file`* 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 path information, it is created in the output directory, that is, the directory where the output image is being written.

`--verbose` Prints detailed information about the link operation, including the objects that are included and the libraries from which they are taken. 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`.

Prefix letters in diagnostic messages

The RVCT tools automatically insert an identification letter to diagnostic messages, as described in Table 2-1. Using these prefix letters enables the RVCT tools to use overlapping message ranges.

Table 2-1 Identifying diagnostic messages

Prefix letter	RVCT tool
`C`	`armcc`
`A`	`armasm`
`L`	`armlink` or `armar`
`Q`	`fromelf`

The following rules apply:

- All the RVCT tools act on a message number without a prefix.
- A message number with a prefix is only acted on by the tool with the matching prefix.
- A tool does not act on a message with a non matching prefix.

Thus, the linker prefix `L` can be used with `--diag_error`, `--diag_remark`, and `--diag_warning`, or when suppressing messages, for example:

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

2.2.13 Using a via file

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

`--via` *`file`* 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 *Overview of via files* on page A-2 in the *Compiler Reference Guide* for more information on writing via files.

2.2.14 Miscellaneous

`--[no_]legacyalign`

By default, the linker assumes execution regions and load regions to be four-byte aligned. This 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. Use this option to ensure strict conformance with the ELF specification. See *Section placement* on page 3-8 for more information.

2.2.15 Controlling compatibility with legacy objects

The ABI in RVCT v3.1 is different to that in RVCT v1.2. Therefore, legacy RVCT v1.2 objects and libraries are not directly compatible with RVCT v3.1.

See *ABI for the ARM Architecture compliance* on page 1-3 in the *Libraries Guide*, and the section on compatibility with legacy objects and libraries in *RealView Compilation Tools Essentials Guide* for more information.

Chapter 3
Using the Basic Linker Functionality

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

- *Specifying the image structure* on page 3-2
- *Section placement* on page 3-8
- *Optimizations and modifications* on page 3-11
- *Using command-line options to create simple images* on page 3-26
- *Using command-line options to handle C++ exceptions* on page 3-32
- *Getting information about images* on page 3-33.

For information about advanced linker functionality, see the descriptions of:

- how to access symbols in Chapter 4 *Accessing Image Symbols*
- how to use scatter-loading in Chapter 5 *Using Scatter-loading Description Files*.

3.1 Specifying the image structure

The structure of an image is defined by the:

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

When describing a memory map:

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

3.1.1 Building blocks for objects and images

An executable file is constructed from a hierarchy of images, regions, output sections, and input sections:

- An image consists of one or more regions. Each region consists of one or more output sections.
- Each output section contains one or more input sections.
- Input sections are the code and data information in an object file.

Figure 3-1 on page 3-3 shows the relationship between regions, output sections, and input sections.

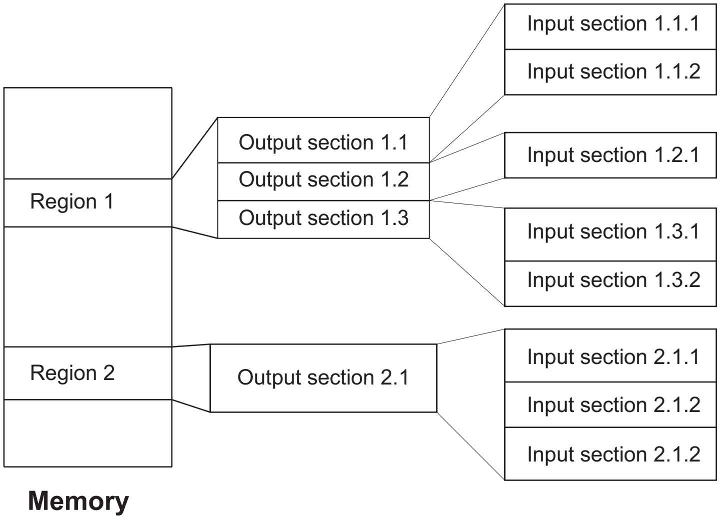


Figure 3-1 Building blocks for an image

Figure 3-1 shows the building blocks that make up the image:

Input sections

An input section contains code or initialized data, or describes a fragment of memory that is not initialized or that must be set to zero before the image can execute. These properties are represented by attributes such as RO, RW and ZI. See *Ordering input sections by attribute* on page 3-9 for more information. These attributes are used by `armlink` to group input sections into bigger building blocks called output sections and regions.

Output sections

An output section is a contiguous sequence of input sections that have the same RO, RW, or ZI attribute. An output section has the same attributes as its constituent input sections. Within an output section, the input sections are sorted according to the rules described in *Section placement* on page 3-8.

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

3.1.2 Load view and execution view of an image

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

The memory map of an image has the following distinct views (as shown in Figure 3-2).

Load view Describes each image region and section in terms of the address it is located at when the image is loaded into memory, that is, the location *before* the image starts executing.

Execution view Describes each image region and section in terms of the address it is located at *while* the image is executing.

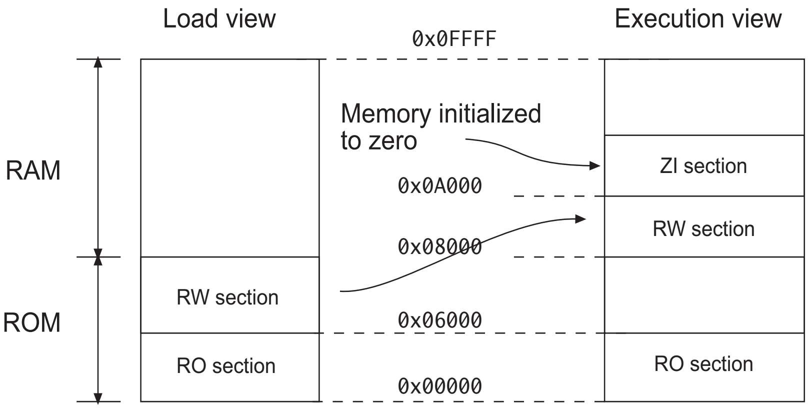


Figure 3-2 Load and execution memory maps

Table 3-1 compares the load and execution views.

Table 3-1 Comparing load and execution views

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

3.1.3 Specifying an image memory map

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

Grouping	How input sections are grouped into output sections and regions.
Placement	Where image regions are to be located in the memory maps.

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

Using command-line options

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

- `--ro-base`
- `--rw-base`
- `--ropi`
- `--rwpi`
- `--split`
- `--rosplit`

The options listed above provide a simplified notation that gives the same settings as a scatter-loading description for a simple image. For more information, see *Using command-line options to create simple images* on page 3-26.

Using a scatter-loading description file

A scatter-loading description file is used for more complex cases where you require complete control over the grouping and placement of image components. To use scatter-loading, specify `--scatter` *`filename`* at the command-line. This is described in full in Chapter 5 *Using Scatter-loading Description Files*.

3.1.4 Image entry points

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

Initial entry point

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

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

Entry points set by the ENTRY directive

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

You can specify multiple entry points in an image with the `ENTRY` directive. The directive marks the output code section with an `ENTRY` keyword that instructs the linker not to remove the section when it performs unused section elimination.

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

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

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

Specifying an initial entry point

You can specify an initial entry point with the `--entry` linker option. You can specify the `--entry` option only once. See the description in *armlink command syntax* on page 2-5 for more information.

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

The initial entry point must meet the following conditions:

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

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

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

  In both these situations, the linker issues a warning.

  Specify a unique entry point with `--entry` to fix this warning.

3.2 Section placement

The linker sorts all the input sections within a region according to their attributes. Input sections with identical attributes form a contiguous block within the region.

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

In general, the linker sorts the input sections in the following order when generating an image:

1. By attribute.
2. By input section name.
3. By their positions in the input list, except where overridden by `FIRST` or `LAST`. See *Using FIRST and LAST to place sections* on page 3-9.

Note

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

If an execution region contains 4MB of Thumb code, 16MB of Thumb-2 code or more than 32MB of ARM code, the linker might change the sort order to reduce the number of long branch veneers to a minimum. See*Veneer generation* on page 3-20 for more information.

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

3.2.1 Ordering input sections by attribute

Portions of the image are collected together into a minimum number of contiguous regions. `armlink` orders input sections by attribute as follows:

1. read-only code
2. read-only data
3. read-write code
4. read-write data
5. zero-initialized data.

Input sections that have the same attributes are ordered by name. Names are considered to be case-sensitive and are compared in alphabetical order using the ASCII collation sequence for characters.

Identically attributed and named input sections are ordered according to their relative positions in the input list.

These rules mean that the positions of identically attributed and named input sections included from libraries are not predictable. If more precise positioning is required, you can extract modules manually and include them in the input list.

3.2.2 Using FIRST and LAST to place sections

Within a region, all RO code input sections are contiguous and form an RO output section that must precede the output section containing all the RW input sections.

If you are not using scatter-loading, use the `--first` and `--last` linker options to place input sections.

If you are using scatter-loading, use the attributes `FIRST` and `LAST` in the scatter-loading description file to mark the first and last input sections in an execution region if the placement order is important.

However, `FIRST` and `LAST` must not violate the basic attribute sorting order as described in *Ordering input sections by attribute*. For example, `FIRST RW` is placed after any read-only code or read-only data.

3.2.3 Aligning sections

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

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

If you require strict conformance with the ELF specification then use the `--no_legacyalign` option. Padding might be inserted to ensure compliance and the linker faults base addresses that are not `0 mod Max(`*`input section alignment`*`)`.

It is possible to use `ALIGN` to expand the alignment of a region, for example, changing something that is normally four-byte aligned to be eight-byte aligned. You cannot reduce the natural alignment, for example, forcing two-byte alignment on something that is normally four-byte aligned. See *Creating regions on page boundaries* on page 5-46 for more information.

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

3.2.4 Ordering execution regions containing Thumb code

The Thumb branch range is 4MB. When an execution region contains Thumb code that exceeds 4MB, `armlink` attempts to order sections that are at a similar average call depth and to place the most commonly called sections centrally. This helps to minimize the number of veneers generated. See *Veneer generation* on page 3-20 for more information.

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

3.3 Optimizations and modifications

This section describes linker optimizations for duplicate sections, unused sections, debug sections, veneers and branching. It also describes how to use linker feedback for repeat compilations.

3.3.1 Common debug section elimination

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

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

3.3.2 Common group or section elimination

If there are inline functions or templates used in the C++ source, the ARM compiler generates complete objects for linking such that each object contains the out-of-line copies of inline functions and template functions that the object requires. When these functions are declared in a common header file, the functions might be defined many times in separate objects that are subsequently linked together. In order to eliminate duplicates, the compiler compiles these functions into separate instances of common code sections or groups.

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

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

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

This optimization is controlled by linker options:

- Use the `--bestdebug` option to use the largest Comdat group (likely to give the best debug view).
- Use the `--no_bestdebug` option to use the smallest Comdat group (likely to give the smallest code size). This is the default. In most cases, use of `--bestdebug` is unlikely.

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

For more information, see:

- *Inlining* on page 4-19 in the *Compiler User Guide*
- *--[no_]debug* on page 2-21 in the *Compiler Reference Guide.*

3.3.3 Unused section elimination

Unused section elimination removes code that is never executed, or data that is not referred to by the code, from the final image. This optimization can be controlled by the `--remove`, `--no_remove`, `--first`, `--last`, and `--keep` linker options. Use the `--info unused` linker option to instruct the linker to generate a list of the unused sections that have been eliminated.

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

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

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

Note

Unused section elimination is a property of all groups, not just common groups.

3.3.4 Unused function elimination

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

If a function is compiled in its own section then VFE is synonymous with unused section elimination (see *Unused section elimination*). However, where an input section contains more that one function, it can only be eliminated if *all* the functions are unused. The linker cannot remove unused functions from *within* a section.

In the rest of this section, it is assumed that functions are compiled in their own sections.

Unused section elimination efficiently removes unused functions from C code. However, in C++ applications, unused sections and RTTI objects are referenced by pointer tables. This means that the elimination algorithm used by the linker cannot guarantee to remove sections and RTTI objects reliably.

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

——— **Note** ———

Assembler source files do not require VFE annotations, provided that they do not reference the C++ libraries. This is because the linker assumes that no virtual function calls are made by object files that do not reference the C++ libraries. Similarly, C source files that are compiled with an old version of `armcc` can participate in VFE provided that they do not reference the C++ libraries.

VFE operates in four modes:

On Use the command-line option `--vfemode=on` to make the linker VFE aware. This is the default mode if you do not specify a VFE option on the command line.

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.

The compiler places the extra information in sections with names beginning `.arm_vfe`. These sections are not referenced by the rest of the code and so are ignored by the linker when it is not VFE aware. Such sections do not, therefore, increase the size of the final image but they do increase the size of object files produced by the compiler.

3.3.5 Linker feedback

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

When the `--inline` optimization is turned on (see *Branch inlining* on page 3-22), functions inlined by the linker are also emitted in the feedback file. These functions are also placed in their own sections.

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

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

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

Linker feedback example

To see how linker feedback works:

1. Create a file `fb.c` containing the code shown in Example 3-1 on page 3-15.

Example 3-1 Feedback example

```
#include <stdio.h>

void legacy()
{
    printf("This is a legacy function, that is no longer used.\n");
}

int cubed(int i)
{
    return i*i*i;
}

void main()
{
    int n = 3;
    printf("%d cubed = %d\n",n,cubed(n));
}
```

2. Compile the program (ignore the warning that the feedback file does not exist):

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

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

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

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

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

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

   Note

   See *Generating image-related information* on page 2-29 and *Controlling linker diagnostics* on page 2-36 for more information.

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

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

This shows that the functions are not used.

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

   ```
   armcc --asm -c --feedback fb.txt fb.c
   armlink --info sizes --list fbout2.txt fb.o -o fb.axf
   ```

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

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

Note

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

3.3.6 RW data compression

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

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

Choosing a compressor

`armlink` gathers information about the content of data sections before choosing the most appropriate compression algorithm to generate the smallest image. If compression is appropriate, the linker can only use one data compressor for all the compressible data sections in the image and different compressions might be tried on these sections to produce the best overall size. Compression is applied automatically if:

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

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

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

- using the `--datacompressor off` option to turn off compression
- specifying a compressor of your choosing.

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

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

How is compression applied?

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

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

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

When choosing a compressor be aware that:

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

The linker prefers compressor 0 or 1 where the data contains mostly zero-bytes (>75%). Compressor 2 is chosen where the data contains few zero-bytes (<10%). If the image is made up only of ARM code, then ARM decompressors are used automatically. If the image contains any Thumb code, Thumb decompressors are used. If there is no clear preference, all compressors are tested to produce the best overall size (see *Choosing a compressor* on page 3-17).

Note

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

Working with RW data compression

When working with RW data compression:

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

  See Chapter 2 *Embedded Software Development* in the *Developer Guide* for more information.

In RVCT v2.0 and earlier, only the `__main` section and the region tables had to be placed in a root region. In RVCT v2.1 and above, RW data compression requires that additional sections (such as `__dc*.o` sections) be placed in a root region.

If you are using a scatter-loading description file:

- Where coded, decompressor objects named `__dc*.o`, must be in a root region, for example:

  ```
  LR 0x8000
  {
      ER_ROOT +0
      {
          __main.o(*)
          * (Region$$Table)
          __scatter*.o(*)
          __dc*.o(*)
      }
  }
  ```

  Or, preferably, use `InRoot$$Sections` to place all library sections that must be in a root region, for example:

  ```
  LR 0x8000
  {
      ER_ROOT +0
      {
          * (InRoot$$Sections)
      }
  }
  ```

  For more information, see:

  — *Assigning sections to a root region* on page 5-43

  — Chapter 2 *Embedded Software Development* in the *Developer Guide*.

- Specify that a load or execution region must not be compressed by adding the `NOCOMPRESS` attribute. See *Formal syntax of the scatter-loading description file* on page 5-9 for more information.

3.3.7 PI to absolute veneers

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

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

3.3.8 Veneer generation

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

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

`armlink` supports the following veneer types:

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

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

If you are using ARMv4T, `armlink` generates veneers when a branch involves change of state between ARM and Thumb. In ARMv5 and above, the `BLX` instruction is used.

Veneer sharing

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

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

Veneer sharing makes it impossible to assign an owning object. Using `--no_veneershare`, therefore, provides a more consistent image layout. This comes at the cost of a significant increase in code size.

Veneer variants

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

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

Inline veneers

In this variant:

- the veneer must be inserted just before the target section to be in range
- an ARM-Thumb interworking veneer has a range of 256 bytes and so the function code must appear in the first 256 bytes immediately after the veneer code
- a Thumb-ARM interworking veneer has a range of zero bytes and so the function code must appear immediately after the veneer code
- an inline veneer is always position-independent.

These limitations mean that you cannot move inline veneers out of an execution region using `Veneer$$Code`. Use the command-line option `--no_inlineveneer` to prevent the generation of inline veneers.

Short branch veneers

In this variant:

- an ARM-Thumb short branch veneer has a range of 4MB
- a short branch veneer is always position-independent.

Long branch veneers

In this variant:

- both ARM-Thumb and Thumb-ARM interworking veneers have a range of 2^{32} bytes
- `armlink` combines long branch capability with the state change capability, therefore, all long branch veneers are also interworking veneers
- a long branch veneer is either absolute or position-independent.

When you are using veneers be aware of the following:

- All veneers cannot be collected into one input section because the resulting veneer input section might not be within range of other input sections. If the sections are not within addressing range, long branching is not possible.

- `armlink` generates position-independent variants of the veneers automatically. However, because such veneers are larger than non position-independent variants, `armlink` only does this where necessary, that is, where the source and destination execution regions are both position-independent and are rigidly related.

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

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

3.3.9 Reuse of veneers with overlay execution regions

`armlink` reuses veneers whenever possible. However, both the following conditions are enforced on reuse:

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

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

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

3.3.10 Branch inlining

`armlink` has global visibility of all your program code and so can perform some additional branch optimizations.

`armlink` uses branch inlining to optimize small function calls in your image. A small function is defined as any one-instruction function that can be inlined into the 4 bytes of a `BL` or `BLX` instruction. In this case, there is no branch and, therefore, the return address is redundant.

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.

Use the command-line options to control branch inlining:

`--no_branchnop`

The linker replaces 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.

Use the `--no_branchnop` option to disable this behavior.

`--inline`

Enables branch inlining. See *Controlling inlining* on page 3-24 for more information.

`--tailreorder`

Moves tail calling sections immediately before their target, if possible, to optimize function calls. See *Handling tail calling sections* on page 3-25 for more information.

If you enable branch inlining, `armlink` scans each function call in the image and then inlines where applicable. When `armlink` inlines a function, it removes the reference to the called function from the caller. `armlink` applies this optimization before any unused sections are eliminated so that any section that is always inlined can then be removed.

Use the `--info` command-line option to display information about branch inlining:

`--info inline`

Displays a message each time a function is inlined and gives the total number of inlines, for example:

```
Small function inlining results

Inlined function __Heap_DescSize from object h1_alloc.o at offset
0x5c in section .text from object malloc.o.
Inlined function __ieee_status from object istatus.o at offset
0x40 in section .text from object _printf_fp_dec.o.
.
Inlined total of 6 calls.
```

Controlling inlining

If you have enabled branch inlining, there are certain conditions that a function must meet in order to be inlined:

- `armlink` handles only the simplest cases and does not inline any instruction that reads or writes to the PC because this depends on the location of the function.
- If your image contains both ARM and Thumb code, functions that are called from the other state must be built for interworking. An ARM caller might inline a Thumb callee if an equivalent ARM instruction is available. However, a Thumb caller cannot inline an ARM callee. Also, `armlink` can inline up to two 16-bit Thumb instructions. However, an ARM caller can only inline a single 16-bit Thumb instruction.
- The action of the linker also depends on the size of the symbol representing a function and on the caller (ARM or Thumb) and the callee (ARM or Thumb) as shown in Table 3-2.

Table 3-2 Inlining small functions

Caller	Callee	Symbol size that can be inlined
ARM	ARM	4 to 8 bytes
ARM	Thumb	2 to 6 bytes
Thumb	Thumb	2 to 6 bytes
Thumb	ARM	4 to 8 bytes

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

  ```
  MOV pc, lr
  ```

  or

  ```
  BX lr
  ```

  A function that consists of just a return sequence can be inlined as a `NOP`.
- A conditional ARM instruction can only be inlined if either the condition on the `BL` matches the condition on the instruction being inlined, or the `BL` or instruction to be inlined is unconditional. For example, `BLEQ` can only inline an unconditional instruction like `ADD` or an instruction with a matching condition like `ADDEQ`.

  An unconditional ARM `BL` can inline any conditional or unconditional instruction that satisfies all the other criteria.

- A `BL` that is the last instruction of an IT block cannot inline a 16-bit Thumb instruction or a 32-bit `MRS`, `MSR`, or `CPS` instruction. This is because the IT block changes the behavior of the instructions within its scope so inlining the instruction changes the behavior of the program.

Handling tail calling sections

As described in *Controlling inlining* on page 3-24, the linker replaces any branch with a relocation that resolves to the next instruction with a `NOP`. This means that tail calling sections, that is, sections that finish with a branch instruction, might be optimized so that their target appears immediately after them in the execution region.

You can take advantage of this behavior by using the command-line option `--tailreorder` to move tail calling sections above their target. If this is possible, be aware that:

- `armlink` 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.
- `armlink` cannot move a tail calling section out of its execution region.
- `armlink` does not move tail calling sections before inline veneers.

Use the `--info` command-line option to display information about tail call optimization. For example, `--info tailreorder` gives information on any moved tail calling sections:

```
Tailcall reorder results
Tail calling Section !!!main from object __main.o placed before .text from kernel.o
Tail calling Section .text from object rt_raise.o placed before .text from sys_exit.o
Tail calling Section .text from object plibspace.o placed before .text from libspace.o
Tail calling Section .text from object aeabi_idiv0.o placed before .text from rt_div0.o
......
```

3.4 Using command-line options to create simple images

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

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

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

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

See *Type 2, one load region and non contiguous execution regions* on page 3-28 for more information.

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

See *Type 3, two load regions and non contiguous execution regions* on page 3-30 for more information.

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

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

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

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

3.4.1 Type 1, one load region and contiguous execution regions

An image of this type consists of a single load region in the load view and three execution regions placed contiguously in the memory map. This approach is suitable for systems that load programs into RAM, for example, an OS bootloader or a desktop system (see Figure 3-3 on page 3-27).

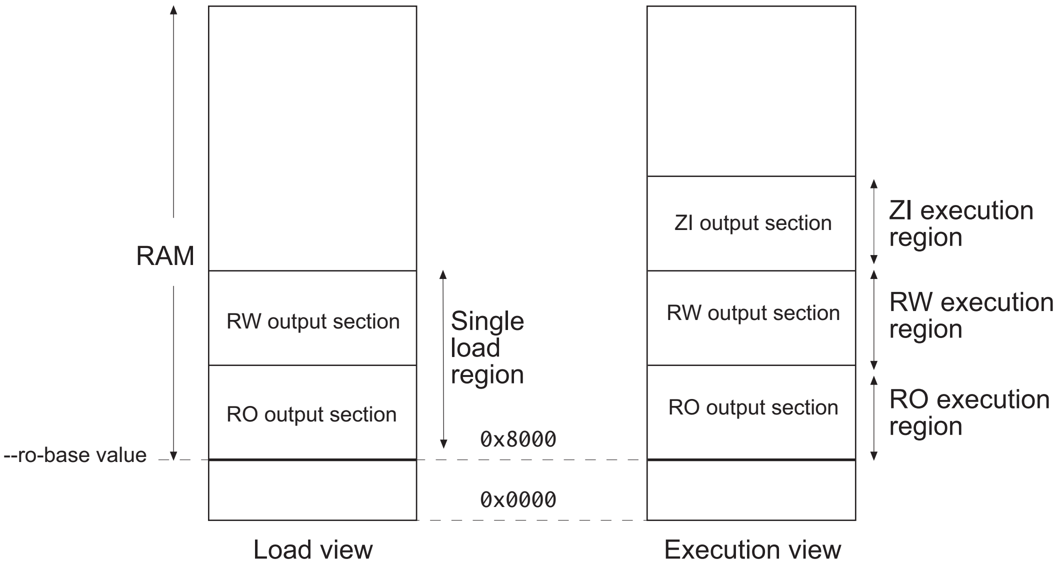


Figure 3-3 Simple type 1 image

Use the following command for images of this type:

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

Load view

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

Execution view

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

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

3.4.2 Type 2, one load region and non contiguous execution regions

An image of this type consists of a single load region, and three execution regions in execution view. The RW execution region is not contiguous with the RO execution region. This approach is used, for example, for ROM-based embedded systems (see Figure 3-4), where RW data is copied from ROM to RAM at startup.

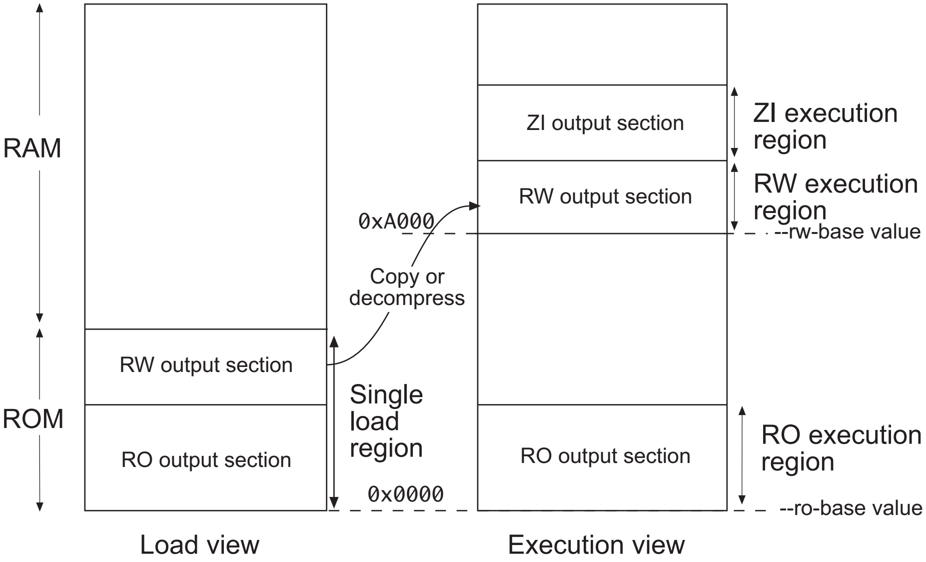


Figure 3-4 Simple type 2 image

Use the following command for images of this type:

```
armlink --ro-base 0x0 --rw-base 0xA000
```

Load view

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

Execution view

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

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

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

Use `armlink` options `--ro-base` *`address`* to specify the load and execution address for the RO output section, and `--rw-base` *`exec_address`* to specify the execution address of the RW output section. If you do not use the `--ro-base` option to specify the address, the default value of `0x8000` is used by `armlink`. For an embedded system, `0x0` is typical for the `--ro-base` value. If you do not use the `--rw-base` option to specify the address, the default is to place RW directly above RO (as in *Type 1, one load region and contiguous execution regions* on page 3-26).

Note

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

3.4.3 Type 3, two load regions and non contiguous execution regions

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

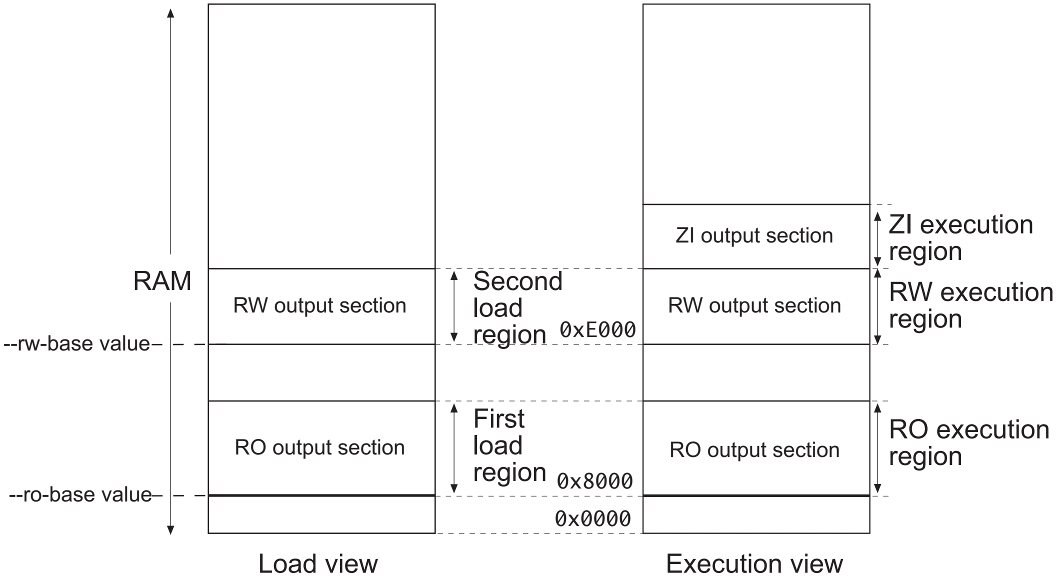


Figure 3-5 Simple type 3 image

Use the following command for images of this type:

```
armlink --split --ro-base 0x8000 --rw-base 0xE000
```

Load view

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

Execution view

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

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

The execution address of the RW region is also the same as its load address, so the contents of the RW output section are not moved from their load address to their execution address. However, the ZI output section is created before execution begins and placed after the RW region.

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

`--split` Splits the default single load region (that contains both the RO and RW output sections) into two load regions (one containing the RO output section and one containing the RW output section) so that they can be placed separately using `--ro-base` and `--rw-base`.

`--ro-base` *`address`*

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

`--rw-base` *`address`*

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

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

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

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

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

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

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

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

Note

Be aware of the following:

- With the default settings, that is, `--exceptions --exception_tables=nocreate`, it is not safe to throw an exception through C or assembly code (unless the C code is compiled with the option `--exceptions`).
- The linker cannot generate cleanup code for automatic variables in C++ code that is compiled without exceptions support, for example, any code compiled with RVCT v1.2, or code compiled with RVCT 3.0 using the `--no_exceptions` option.

3.6 Getting information about images

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

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

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

See *Generating image-related information* on page 2-29 for more information on the topic keywords you can specify for the `--info` command-line option.

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

Example 3-2 Image information

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

3712        1580        19        44      10200       7436   Object Totals
0              0        16         0          0          0   (incl. Generated)
0              0         3         0          0          0   (incl. Padding)
21376        648       805         4        300      10216   Library Totals
0              0         6         0          0          0   (incl. Padding)

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

Code (inc. data)   RO Data    RW Data    ZI Data      Debug
25088       2228       824         48      10500      17652   Grand Totals
25088       2228       824         48      10500      17652   ELF Image Totals
25088       2228       824         48          0          0   ROM Totals

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

Total RO  Size (Code + RO Data)           25912 (  25.30kB)
Total RW  Size (RW Data + ZI Data)        10548 (  10.30kB)
Total ROM Size (Code + RO Data + RW Data)   25960 (  25.35kB)
```

Code (inc. Data)

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

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

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

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

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

Object Totals

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

(incl. Generated)

`armlink` might generate image contents, for example, interworking veneers, and input sections such as region tables. If the `Object Totals` row includes this type of data, it is shown in this row. In Example 3-2 on page 3-33 there are 19 bytes of RO data in total, of which 16 bytes is linker-generated RO data.

Library Totals

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

(incl. Padding)

`armlink` inserts padding, if required, to force section alignment (see *Aligning sections* on page 3-9). If the `Object Totals` row includes this type of data, it is shown in the associated `(incl. Padding)` row. Similarly, if the `Library Totals` row includes this type of data, it is shown in its associated row. In Example 3-2 on page 3-33, there are 19 bytes of RO data in the object total, of which 3 bytes is linker-generated padding, and 805 bytes of RO data in the library total, with 6 bytes of padding.

Grand Totals

Shows the true size of the image. In Example 3-2 on page 3-33 there are 10200 bytes of ZI data (in `Object Totals`) and 300 of ZI data (in `Library Totals`) giving a total of 10500 bytes.

ELF Image Totals

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

In Example 3-2 on page 3-33, RW data compression is not enabled. If data is compressed, the RW value changes. See *RW data compression* on page 3-17 for more information.

ROM Totals

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

3.6.1 Using image-related information

You can use the `--map` option to create an image map. This includes the address and size of each load region, execution region, and input section in the image, and shows how RW data compression is applied. See *Generating image-related information* on page 2-29 for more information.

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

Chapter 4
Accessing Image Symbols

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

4.1 ARM/Thumb synonyms

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

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

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

4.2 Accessing linker-defined symbols

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

These symbolic addresses can be imported and used as relocatable addresses by your assembly language programs, or referred to as **`extern`** symbols from your C or C++ source code. See *Importing linker-defined symbols* on page 4-7 for more information.

Note

Linker-defined symbols are only generated when your code references them.

4.2.1 Region-related symbols

Region-related symbols are generated when the linker creates an image. Table 4-1 shows the symbols that the linker generates for every execution region present in the image.

Table 4-1 Region-related linker symbols

Symbol	Description
`Load$$`*`region_name`*`$$Base`	Load address of the region
`Image$$`*`region_name`*`$$Base`	Execution address of the region
`Image$$`*`region_name`*`$$Length`	Execution region length in bytes (multiple of 4) excluding ZI length
`Image$$`*`region_name`*`$$Limit`	Address of the byte beyond the end of the non ZI part of the execution region
`Image$$`*`region_name`*`$$RO$$Base`	Execution address of the RO output section in this region
`Image$$`*`region_name`*`$$RO$$Length`	Length of the RO output section in bytes (multiple of 4)
`Image$$`*`region_name`*`$$RO$$Limit`	Address of the byte beyond the end of the RO output section in the execution region
`Image$$`*`region_name`*`$$RW$$Base`	Execution address of the RW output section in this region
`Image$$`*`region_name`*`$$RW$$Length`	Length of the RW output section in bytes (multiple of 4)
`Image$$`*`region_name`*`$$RW$$Limit`	Address of the byte beyond the end of the RW output section in the execution region

Table 4-1 Region-related linker symbols (continued)

Symbol	Description
`Image$$`*`region_name`*`$$ZI$$Base`	Execution address of the ZI output section in this region
`Image$$`*`region_name`*`$$ZI$$Length`	Length of the ZI output section in bytes (multiple of 4)
`Image$$`*`region_name`*`$$ZI$$Limit`	Address of the byte beyond the end of the ZI output section in the execution region

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

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

Note

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

Using scatter-loading description files

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

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

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

Placing the stack and heap above the ZI region

One common use of region-related symbols is to place a heap directly above the ZI region. Example 4-1 on page 4-5 shows how to create a retargeted version of `__user_initial_stackheap()` in assembly language. The example assumes that you are using the default one region memory model from the ARM C libraries. See *__user_initial_stackheap()* on page 2-87 in the *Libraries and Floating Point Support Guide* for more information. See also Chapter 6 *Handling Processor Exceptions* in the *Developer Guide* for an example of how to do this in C.

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

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

4.2.2 Section-related symbols

The output section symbols shown in Table 4-2 are generated if you use command-line options to create a simple image. A simple image has three output sections (RO, RW, and ZI) that produce the three execution regions. For every input section present in the image, the linker generates the input symbols shown in Table 4-3 on page 4-6.

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

Output section symbols

If you are using a scatter-loading description file, the output section symbols in Table 4-2 are undefined. If your code accesses these symbols, you must treat it as a weak reference.

The standard implementation of `__user_initial_stackheap()` uses the value in `Image$$ZI$$Limit`. Therefore, if you are using a scatter-loading description file you might have to re-implement `__user_initial_stackheap()` to set the heap and stack boundaries. For more information, see Chapter 5 *Using Scatter-loading Description Files*.

Table 4-2 Image-related symbols

Symbol	Section type	Description
`Image$$RO$$Base`	Output	Address of the start of the RO output section.
`Image$$RO$$Limit`	Output	Address of the first byte beyond the end of the RO output section.
`Image$$RW$$Base`	Output	Address of the start of the RW output section.

Table 4-2 Image-related symbols (continued)

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

Input section symbols

If your code refers to the input-section symbols, it is assumed that you expect all the input sections in the image with the same name to be placed contiguously in the image memory map. If your scatter-loading description places these input sections non contiguously, the linker diagnoses an error because the use of the base and limit symbols over non contiguous memory usually produces unpredictable and undesirable effects.

Table 4-3 Section-related symbols

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

4.2.3 Importing linker-defined symbols

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

```
extern unsigned int symbol_name;
```

or:

```
extern void *symbol_name;
```

If you declare a symbol as an `int`, then you must use the address-of operator to obtain the correct value as shown in Example 4-2.

Example 4-2 Importing linker-defined symbols

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

4.3 Accessing symbols in another image

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

This can be used, for example, if you have one image that always resides in ROM and multiple images that are loaded into RAM. The images loaded into RAM can access global functions and data from the image located in ROM.

4.3.1 Creating a symdefs file

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

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

Note

If *`filename`* does not exist, the file is created containing all the global symbols. If *`filename`* exists, the existing contents of *`filename`* are used to select the symbols that are output when the linker rewrites the file. If you do not want this behavior, ensure that any existing symdefs file is deleted before the link step.

Outputting a subset of the global symbols

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

When *`filename`* exists, the linker uses its contents to restrict the output to a subset of the global symbols. To restrict the output symbols:

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

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

4.3.2 Reading a symdefs file

A symdefs file can be considered as an object file with symbol information but no code or data. To read a symdefs file, add it to your file list as you do for any object file. The linker reads the file and adds the symbols and their values to the output symbol table. The added symbols have `ABSOLUTE` and `GLOBAL` attributes.

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

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

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

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

4.3.3 Symdefs file format

The symdefs file contains symbols and their values. Unlike other object files, however, it does not contain any code or data.

The file consists of an identification line, optional comments, and symbol information as shown in Example 4-3.

Example 4-3 Symdefs file format

```
#<SYMDEFS># ARM Linker, RVCT3.1 [Build num]: Last Updated: Date
;value type name, this is an added comment
0x00008000 A __main
0x00008004 A __scatterload
0x000080e0 T main
0x0000814d T _main_arg
0x0000814d T __argv_alloc
0x00008199 T __rt_get_argv
...
   # This is also a comment, blank lines are ignored
...
0x0000a4fc D __stdin
0x0000a540 D __stdout
0x0000a584 D __stderr
0xfffffffd N __SIG_IGN
```

Identifying string

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

The identifying string is followed by linker version information, and date and time of the most recent update of the symdefs file. The version and update information are not part of the identification string.

Comments

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

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

Symbol information

The symbol information is provided by the address, type, and name of the symbol on a single line:

Symbol value	The linker writes the absolute address of the symbol in fixed hexadecimal format, for example, `0x00008000`. If you edit the file, you can use either hexadecimal or decimal formats for the address value.
Type flag	A single letter to show symbol type:
	A ARM code
	T Thumb code
	D Data
	N Number.
Symbol name	The symbol name.

4.4 Hiding and renaming global symbols

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

Use the `armlink` command-line option `--edit` *`file-list`* to specify the steering file (see the description of the `--edit` option in *armlink command syntax* on page 2-5). When you are specifying more than one steering file, the syntax can be either of the following:

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

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

Do not include spaces between the comma and the filenames.

4.4.1 Steering file format

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

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

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

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

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

4.4.2 Steering file commands

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.

IMPORT

The `IMPORT` command specifies that a symbol is defined in a shared object at runtime.

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 undefined global symbol is to be renamed. Wildcards must have a corresponding wildcard in *`replacement_pattern`*. The characters matched by the *`replacement_pattern`* wildcard are substituted for the *`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.

EXPORT

The `EXPORT` command specifies that a symbol can be accessed by other shared objects or executables.

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.

replacement_pattern

Is a string, optionally including wildcard characters (either `*` or ?), to which the defined global symbol is to be renamed. Wildcards must have a corresponding wildcard in *replacement_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.

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. Wildcards 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 symbol name that already exists, even if the target symbol name is being renamed itself. Only one wildcard character (either `*` or `?`) is permitted in `RENAME`. For example, given an image containing the symbols `func1`, `func2`, and `func3`:

```
EXPORT func1 AS func2    ;invalid, func2 exists
RENAME func3 AS b2
EXPORT func1 AS func3    ;invalid, func3 exists, even though it is renamed to b2
```

The linker processes the steering file before doing any replacements. You cannot, therefore, use `RENAME A AS B` on line 1 and then `RENAME B AS A` on line 2.

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, that must be matched to a defined global symbol.
`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 4-4 on page 4-17. Create an `ed.txt` file containing the line `RESOLVE MP3* AS MyMP3*`, and issue 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`.

Example 4-4 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()
{
}
```

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 wildcards 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.

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 4-5. This example produces a partially linked object with all global symbols hidden, except those beginning with `os_`.

Example 4-5 Using the HIDE command

```
steer.txt

HIDE *            ; Hides all global symbols
SHOW os_*         ; Shows all symbols beginning with 'os_'
```

Link this example with the command:

```
armlink --partial input_object.o --edit steer.txt -o partial_object.o
```

This example 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.

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 *HIDE* on page 4-19 for further information.

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

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

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

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

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

Example 4-6 shows the legacy function `foo()` modified to result in a call to `ExtraFunc()` and a call to `foo()`. For more information, see the *ARM ELF specification*, `aaelf.pdf`, in *`install_directory`*`\Documentation\Specifications\...`.

Example 4-6 Using $Super$$ and $Sub$$

```
extern void ExtraFunc(void);
extern void $Super$$foo(void):

/* this function is called instead of the original foo() */
void $Sub$$foo(void)
{
  ExtraFunc();    /* does some extra setup work */
  $Super$$foo();  /* calls the original foo() function */
}
```

4.6 Symbol versioning

The linker conforms to the BPABI and supports symbol versions.

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

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

4.6.1 Version

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

Note

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

4.6.2 Default version

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

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

4.6.3 Creating versioned symbols

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

Embedded symbols

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

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

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

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

- *Adding symbol versions* on page 2-26 in *Compiler User Guide*
- how to write ARM and Thumb assembly language in the *Assembler Guide*.

Example 4-7 places the symbols `foo@ver1`, `foo@@ver2`, and `bar@@ver1` into the object symbol table:

Example 4-7 Creating versioned symbols, embedded symbols

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

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

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

Steering file

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

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

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

A steering file can be provided in addition to the embedded symbol method. If you choose to do this then your script file must match your embedded symbols and use the *Backus-Naur Form* (BNF) notation:

```
version_definition ::=

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

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

```
symbol_association ::=

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

where:

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

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

Example 4-8 shows a steering file that corresponds to the embedded symbols example (Example 4-7 on page 4-23) with the addition of dependency information so that `ver2` depends on `ver1`:

Example 4-8 Creating versioned symbols, steering file

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

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

Errors & warnings

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

Filename

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

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

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

Chapter 5
Using Scatter-loading Description Files

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

5.1 About scatter-loading

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

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

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

The scatter-loading mechanism enables you to specify the memory map of an image to the linker using a description in a text file. Scatter-loading gives you complete control over the grouping and placement of image components. Scatter-loading can be used for simple images, but it is generally only used for images that have a complex memory map, that is, where multiple regions are scattered in the memory map at load and execution time.

5.1.1 Symbols defined for scatter-loading

When the linker creates an image using a scatter-loading description file, it creates some region-related symbols. These are described in *Region-related symbols* on page 4-3. The linker creates these special symbols only if your code references them.

Undefined symbols

Be aware, the following symbols are *not* defined when a scatter-loading description file is used:

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

See *Accessing linker-defined symbols* on page 4-3 for more information.

If you use a scatter-loading description file but do not specify any special region names and do not re-implement `__user_initial_stackheap()`, the library generates an error message:

```
Error: L6915E: Library reports error: scatter-load file declares no heap or
stack regions and __user_initial_stackheap is not defined.
```

For more information see:

- Chapter 2 *The C and C++ Libraries* in the *Libraries and Floating Point Support Guide* for more information on library memory models
- Chapter 2 *Embedded Software Development* in the *Developer Guide*.

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

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

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

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

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

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

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

Note

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

5.1.3 When to use scatter-loading

The command-line options to the linker give some control over the placement of data and code, but complete control of placement requires more detailed instructions than can be entered on the command line. Situations where scatter-loading descriptions are either necessary or very useful are:

Complex memory maps

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

Different types of memory

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

Memory-mapped I/O

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

Functions at a constant location

A function can be placed at the same location in memory even though the surrounding application has been modified and recompiled.

Using symbols to identify the heap and stack

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

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

Note

If you are compiling for the Cortex-M3 processor, this has a fixed memory map and so you can use a scatter-loading description file to define both stack and heap. An example of this is supplied as `Cortex-M3.scat` in the main examples directory `install_directory\RVDS\Examples`.

5.1.4 Scatter-loading command-line option

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

`--scatter` *`description_file`*

This instructs the linker to construct the image memory map as described in *`description_file`*. The format of the description file is given in *Formal syntax of the scatter-loading description file* on page 5-9.

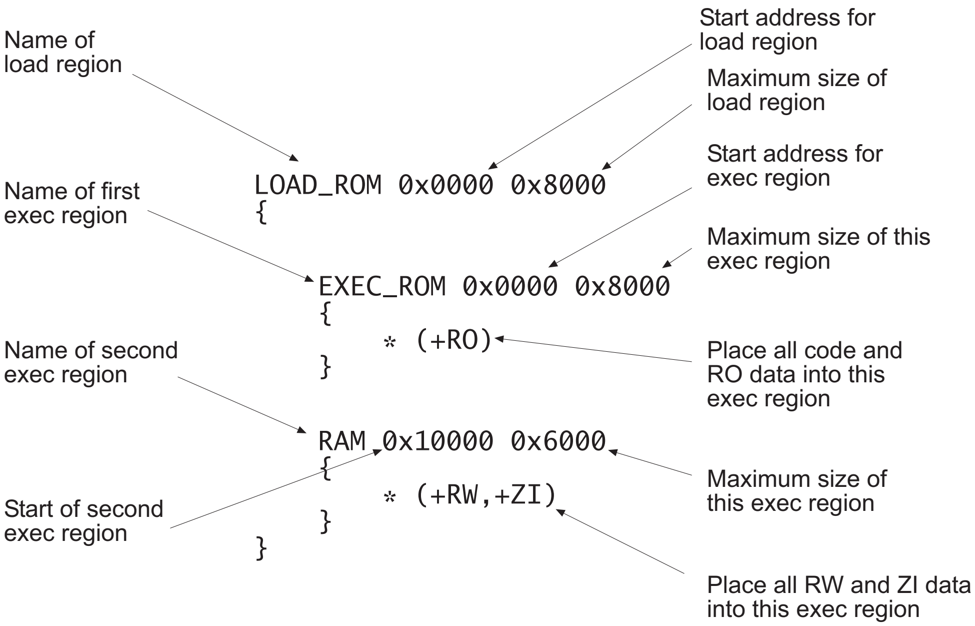
For additional information on scatter-loading description files, see:

- *Examples of specifying region and section addresses* on page 5-32
- *Equivalent scatter-loading descriptions for simple images* on page 5-48
- Chapter 2 *Embedded Software Development* in the *Developer Guide*.

5.1.5 Images with a simple memory map

The scatter-loading description in Figure 5-1 on page 5-6 loads the segments from the object file into memory corresponding to the map shown in Figure 5-2 on page 5-6. The maximum size specifications for the regions are optional but, if they are included, these enable the linker to check that a region has not overflowed its boundary.

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



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

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

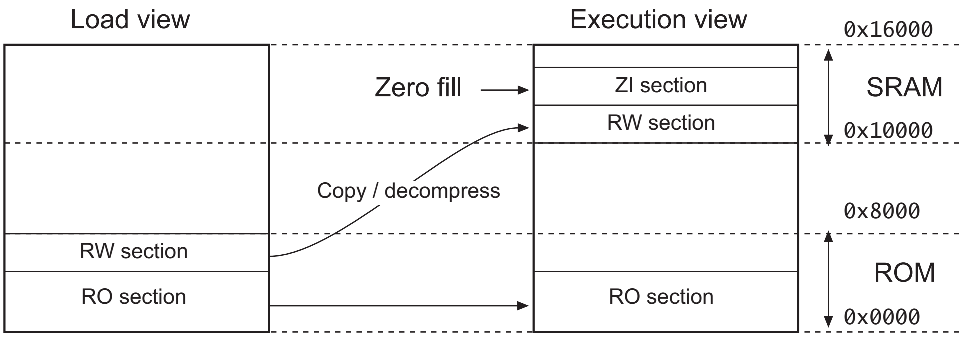


Figure 5-2 Simple scatter-loaded memory map

5.1.6 Images with a complex memory map

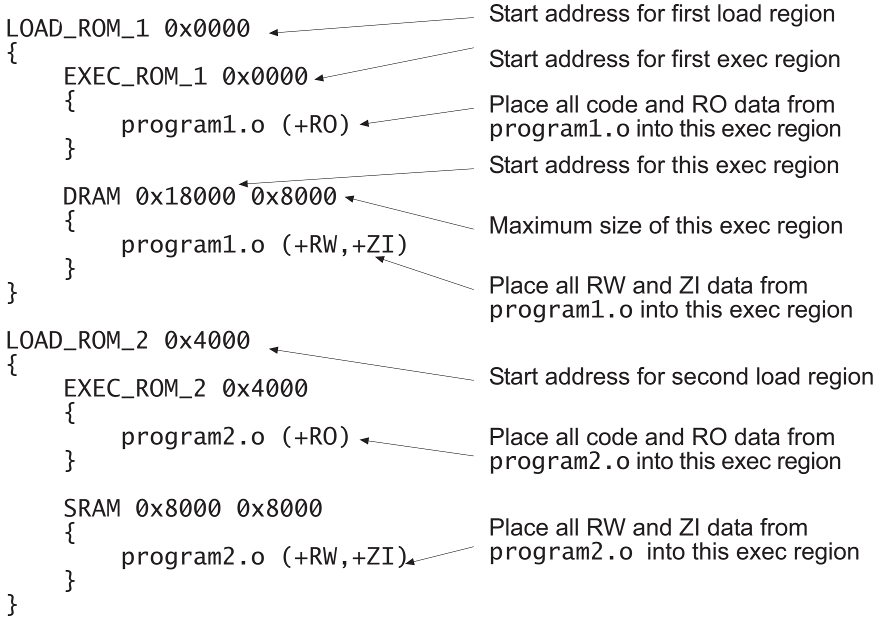
The scatter-loading description in Figure 5-3 loads the segments from the `program1.o` and `program2.o` files into memory corresponding to the map shown in Figure 5-4 on page 5-8.

Unlike the simple memory map shown in Figure 5-2 on page 5-6, this application cannot be specified to the linker using only the basic command-line options.

Caution

The scatter-loading description in Figure 5-3 specifies the location for code and data for `program1.o` and `program2.o` only. If you link an additional module, for example, `program3.o`, and use this description file, the location of the code and data for `program3.o` is not specified.

Unless you want to be very rigorous in the placement of code and data, it is advisable to use the `*` or `.ANY` specifier to place leftover code and data. See *Placing regions at fixed addresses* on page 5-34 for more information.



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

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

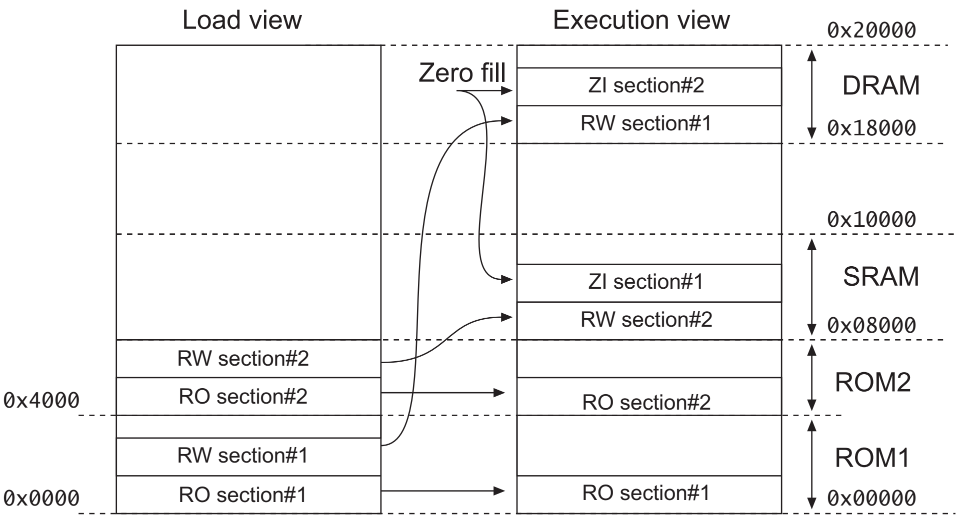


Figure 5-4 Complex scatter-loaded memory map

5.2 Formal syntax of the scatter-loading description file

A scatter-loading description file is a text file that describes the memory map of the target embedded product to the linker. The file extension for the description file is not significant if you are using the linker from the command line. The description file enables you to specify the:

- load address and maximum size of each load region
- attributes of each load region
- execution regions derived from each load region
- execution address and maximum size of each execution region
- input sections for each execution region.

The description file format reflects the hierarchy of load regions, execution regions, and input sections.

Note

How input sections are assigned to regions is completely independent of the order in which selection patterns are written in the scatter-loading description file. The best match between selection patterns and either file/section names or section attributes wins. See *Resolving multiple matches* on page 5-24.

5.2.1 BNF notation and syntax

Table 5-1 summarizes the *Backus-Naur Form* (BNF) symbols that are used to describe a formal language.

Table 5-1 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`.

Table 5-1 BNF syntax (continued)

Symbol	Description
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`.

5.2.2 Overview of the syntax of scatter-loading description files

Note

In the BNF definitions in this section, line returns and spaces have been added to improve readability. They are not required in the scatter-loading definition and are ignored if present in the file.

A `scatter_description` is defined as one or more *`load_region_description`* patterns:

```
Scatter_description ::=

  load_region_description+
```

A *`load_region_description`* is defined as a load region name, optionally followed by attributes or size specifiers, and one or more execution region descriptions:

```
load_region_description ::=
    load_region_name (base_address | ("+" offset)) [attributes] [max_size]
    "{"
        execution_region_description+
    "}"
```

An *`execution_region_description`* is defined as an execution region name, a base address specification, optionally followed by attributes or size specifiers, and one or more input section descriptions:

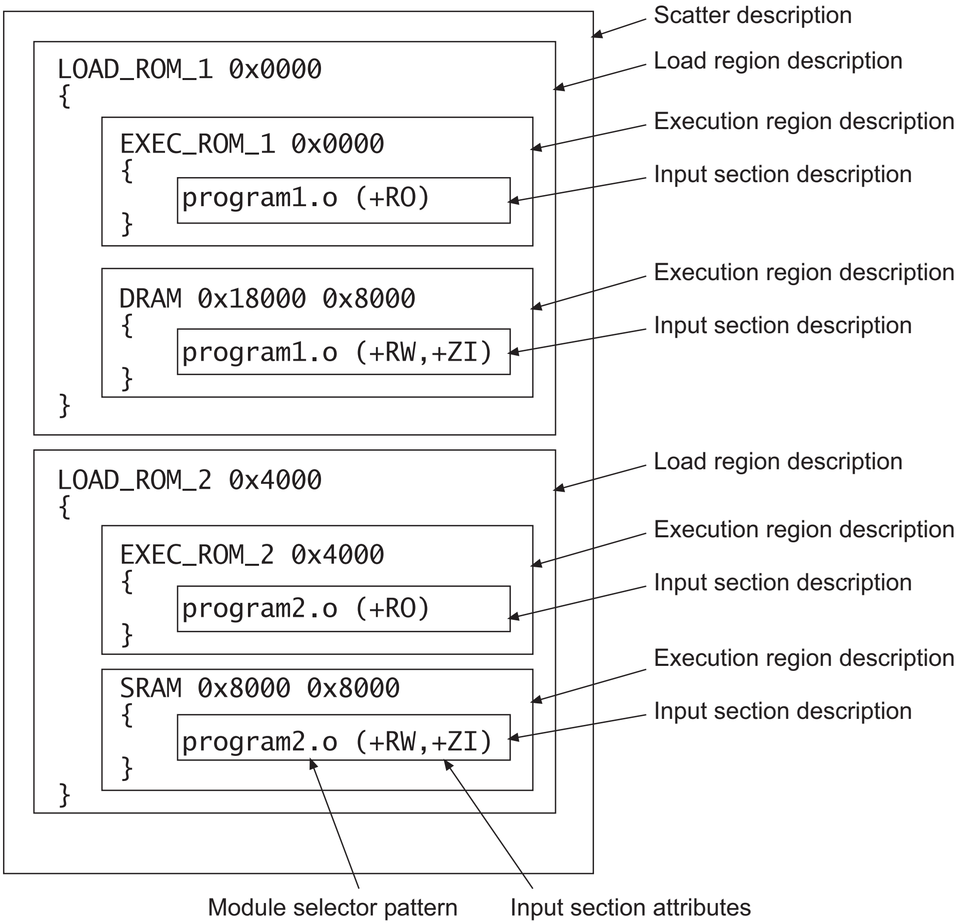
```
execution_region_description ::=
    exec_region_name (base_address | "+" offset) [attribute_list][max_size | "-" length]
    "{"
        input_section_description*
    "}"
```

An *input_section_description* is defined as a source module selector pattern optionally followed by input section selectors:

```
input_section_description ::=
    module_select_pattern
        [ "(" input_section_selector ( "," input_section_selector )* ")" ]

input_section_selector ::=
    ("+" input_section_attr | input_section_pattern | input_symbol_pattern)
```

Figure 5-5 shows the contents and organization of a typical scatter-loading description file.



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

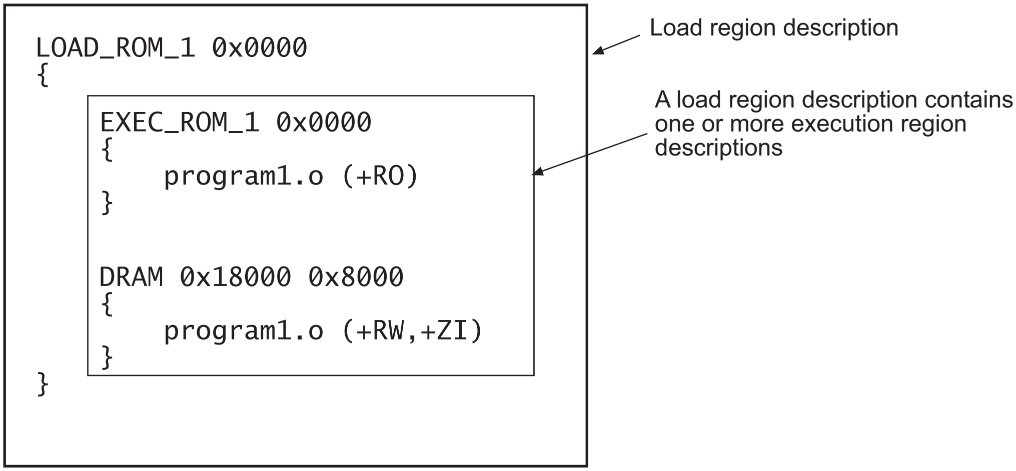
Figure 5-5 Components of a scatter-loading file definition

5.2.3 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 (optional)
- a maximum size (optional)
- a list of execution regions (used to identify the type and location of modules in the execution view).

Figure 5-6 shows the components of a typical load region description.



```
LOAD_ROM_1 0x0000
{
    EXEC_ROM_1 0x0000
    {
        program1.o (+RO)
    }

    DRAM 0x18000 0x8000
    {
        program1.o (+RW,+ZI)
    }
}
```

Load region description

A load region description contains one or more execution region descriptions

Figure 5-6 Components 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	The linker generates a `Load$$`*`exec_region_name`*`$$base` symbol for each execution region. This symbol holds the load address of the execution region (see *Execution region description* on page 5-15).

The `load_region_name`, however, is used only to identify each region, that is, it is not used to generate `Load$$region_name` symbols.

Note

An image created for use by a debugger requires a unique base address for each region because the debugger must load regions at their load addresses. Overlapping load region addresses result in part of the image being overwritten.

A loader or operating system, however, can correctly load overlapping position-independent regions. One or more of the position-independent regions is automatically moved to a different address.

`base_address`	Specifies the address where objects in the region are to be linked. *`base_address`* must be word-aligned.
`+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 after the base address of the load region.
`attribute_list`	Specifies the properties of the load region contents:

`ABSOLUTE`	Absolute address.
`PI`	Position-independent.
`RELOC`	Relocatable.
`OVERLAY`	Overlaid.
`NOCOMPRESS`	Must not be compressed.
`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.

You can specify only one of the attributes `ABSOLUTE`, `PI`, `RELOC`, and `OVERLAY`. The default load region attribute is `ABSOLUTE`.

Load regions that have one of `PI`, `RELOC`, or `OVERLAY` attributes can have overlapping address ranges. The linker faults overlapping address ranges for `ABSOLUTE` load regions.

The `OVERLAY` keyword enables you to have multiple execution regions at the same address. ARM does not provide an overlay mechanism in RVCT. Therefore, to use multiple execution regions at the same address, you must provide your own overlay manager.

RW data compression is enabled by default. The `NOCOMPRESS` keyword enables you to specify that a load region must not be compressed in the final image.

max_size Specifies the maximum size of the load region. 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 5-15.

5.2.4 Execution region description

An execution region has:

- a name
- a base address (either absolute or relative)
- an optional maximum size specification
- attributes that specify the properties of the execution region
- one or more input section descriptions (the modules placed into this execution region).

Figure 5-7 shows the components of a typical execution region description.

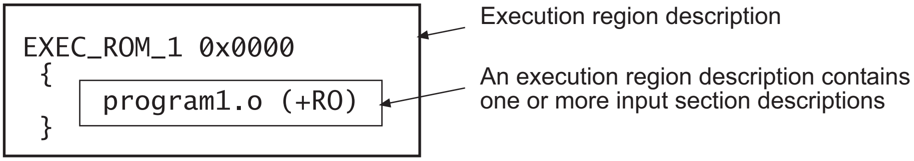


Figure 5-7 Components of an execution region description

The 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.
+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 there is no preceding execution region (that is, 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 the `+offset` form is used and the encompassing load region has the `RELOC` attribute, the execution region inherits the `RELOC` attribute. However, if a fixed `base_address` is used, future occurrences of `offset` do not inherit the `RELOC` attribute.

`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.
`PI`	Position-independent.
`OVERLAY`	Overlaid.
`FIXED`	Fixed address. Both the load address and execution address of the region is specified by the base designator (the region is a root region). See *Creating root execution regions* on page 5-32. The base designator must be either an absolute base address, or an offset of `+0`.
`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.
`EMPTY`	Reserves an empty block of memory of a given length in the execution region, typically used by a heap or stack. See *Reserving an empty region* on page 5-43 for further information.
`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. In certain situations, for example, simulation, this is preferable to spending a long time in a zeroing loop.
`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.
`FILL`	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.
`NOCOMPRESS`	Must not be compressed.
`UNINIT`	Must not be zero initialized.

`max_size`	An optional number that instructs the linker to generate an error if the region has more than *`max_size`* bytes allocated to it.
`-length`	If the length is given as a negative value, the *`base_address`* is taken to be the end address of the region. Typically used with `EMPTY` to represent a stack that grows down in memory. See *Reserving an empty region* on page 5-43 for more information.
`input_section_description`	Specifies the content of the input sections. See *Input section description* on page 5-19.

When specifying the properties of the execution region:

- An execution region must be one of the attributes `ABSOLUTE`, `PI`, or `OVERLAY`. If no attribute is specified, `ABSOLUTE` is the default attribute of the execution region.
- Execution regions that use the *`+offset`* form of the base designator either inherit the attributes of the preceding execution region, (or of the containing load region if this is the first execution region in the load region), or have the `ABSOLUTE` attribute.
- 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.

- The attribute `RELOC` cannot be explicitly specified for execution regions. The region can only be `RELOC` by inheriting the attribute from a load region.
- It is not possible for an execution region that uses the `+offset` form of the base designator to have its own attributes (other than the `ABSOLUTE` attribute that overrides inheritance). Use the combination `+0 ABSOLUTE` to set a region to `ABSOLUTE` without changing the start location.
- Execution regions that are specified as `PI` or `OVERLAY` (or that have inherited the `RELOC` attribute) are permitted to have overlapping address ranges. The linker faults overlapping address ranges for `ABSOLUTE` and `FIXED` execution regions.
- RW data compression is enabled by default. The `NOCOMPRESS` keyword enables you to specify that an execution region must not be compressed in the final image.
- `UNINIT` specifies that any ZI output section in the execution region must not be zero-initialized. Use this to create execution regions containing uninitialized data or memory-mapped I/O.
- Consecutive execution regions with the `OVERLAY` attribute and a base offset of `+0` are given the same base address.

5.2.5 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 5-8 shows the components of a typical input section description.

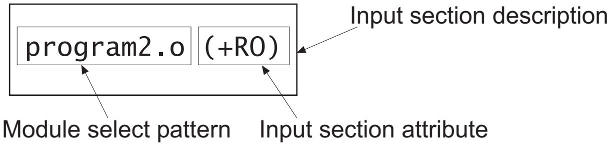


Figure 5-8 Components of an input section description

The 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)
```

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 wildcards to simplify searching. For example, use `*libname.lib` to match `C:\lib dir\libname.lib`.

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 *do not care* assignments.

Use `.ANY`*`num`* to give a specific priority ordering where *`num`* is a positive integer suffix from 0 upwards. The highest priority being given to the highest integer.

Example 5-1 .ANY with an integer suffix

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

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`.

The following attribute selector patterns describe the placement order of a section within the execution region:

First and last sections

`FIRST` and `LAST` can be used 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 only be one `FIRST` or `LAST` attribute in an *`input_section_selector`* list and it must follow a single *`input_section_attr`*. For example:

`*(section, +FIRST)`

This is pattern is correct

`*(+FIRST, section)`

This pattern is incorrect and produces an error message: `Error: L6234E: FIRST must follow a single selector`.

Special module selector

The special module selector pattern `.ANY` enables you to assign input sections to execution regions without considering their parent module. Use one or more `.ANY` patterns to fill up the execution regions with *do not care* assignments. In most cases, using a single `.ANY` is equivalent to using the `*` module selector.

Modified selectors

You cannot have two `*` selectors in a scatter-loading description 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.

Unassigned sections

The input section descriptions having the `.ANY` module selector pattern are resolved after all other (non-`.ANY`) input section descriptions have been resolved. All sections not assigned to an execution region are assigned to a `.ANY` region.

If more than one `.ANY` pattern is present, the linker takes the section with the largest size not assigned to an execution region and assigns the section to the most specific `.ANY` execution region that has enough free space. When `armlink` makes this choice, `.ANY(.text)` is judged to be more specific than `.ANY(+RO)`.

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

For example:

- If you have two equally specific execution regions where one has a size limit of `0x2000` and the other has no limit, then all the sections are assigned to the second unbounded `.ANY` region.
- If you have two equally specific execution regions where one has a size limit of `0x2000` and the other has a size limit of `0x3000`, then the first sections to be placed are assigned to the second `.ANY` region of size limit `0x3000`

until the remaining size of the second `.ANY` is reduced to `0x2000`. From this point, sections are assigned alternately between both `.ANY` execution regions.

`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.

Order of input section descriptors is not significant.

5.2.6 Resolving multiple matches

If a section matches more than one execution region, matches are resolved as described below. However, if a unique match cannot be found, the linker faults the scatter-loading description. Each section is selected by a *`module_select_pattern`* and an *`input_section_selector`*.

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 ARM-supplied C libraries
- `*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 the assembly file area named `BLOCK_42`.

  Note

  The compiler produces areas that can be identified by input section patterns such as `.text`, `.data`, `.constdata`, and `.bss`. These names, however, might change in the future and you must avoid using them.

  If you want to match a specific function or **`extern`** data from a C or C++ file, either:

  — compile the function or data in a separate module and match the module object name

  — use `#pragma arm section` or `__attribute__` to specify the name of the section containing the code or data of interest. See *Pragmas* on page 4-49 in the *Compiler Reference Guide* for more information on pragmas.

The following variables are used to describe multiple matches:

- *m1* and *m2* represent module selector patterns
- *s1* and *s2* represent input section selectors.

In the case of multiple matches, the linker determines the region to assign the input section to on the basis of the `module_select_pattern` and `input_section_selector` pair that is the most specific.

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*.

The sequence `armlink` uses 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:
   - *s1* is a literal input section name that is, it contains no pattern characters, and *s2* matches input section attributes other than `+ENTRY`
   - *m1* is more specific than *m2*
   - *s1* is more specific than *s2*.

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.

Example 5-2 shows multiple execution regions and pattern matching.

Example 5-2 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
    }
}
```

5.2.7 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

ARM recommends that you use forward slashes in path names to ensure they are understood on Windows and Unix platforms.

5.2.8 Expression evaluation in scatter-loading files

Scatter files frequently contain numeric constants that are specified as expressions. The linker also provides an assertion function called `ScatterAssert` that takes an expression as a parameter. An error message is generated if this expression does not evaluate to true. See *ScatterAssert function* on page 5-28 for more information.

Expression usage

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.

Expression rules

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 *Builtin functions* 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.

Builtin functions

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 5-2.

Table 5-2 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 4-3 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 on page 5-29.

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 5-7 on page 5-31 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 5-3.

Table 5-3 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.

Symbol related function

The symbol related function, `defined(`*`global_symbol_name`*`)` returns zero if *`global_symbol_name`* is not defined and nonzero if it is defined.

Examples

Example 5-3 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 5-4 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 5-5 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 5-6 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 5-6 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 5-7 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
```

5.3 Examples of specifying region and section addresses

This section describes input and executions sections, regions and preprocessing directives. For examples on accessing data and functions at fixed addresses, see Chapter 2 *Embedded Software Development* in the *Developer Guide*.

5.3.1 Selecting veneer input sections in scatter-loading descriptions

Veneers are used to switch between ARM and Thumb code or to perform a longer program jump than can be specified in a single instruction. See *Veneer generation* on page 3-20. Use a scatter-loading description file to place linker-generated veneer input sections. At most, one execution region in the scatter-loading description file can have the `*(Veneer$$Code)` section selector.

If it is safe to do so, the linker places veneer input sections into the region identified by the `*(Veneer$$Code)` section selector. It might not be possible for a veneer input section to be assigned to the region because of address range problems or execution region size limitations. If the veneer cannot be added to the specified region, it is added to the execution region containing the relocated input section that generated the veneer.

Note

Instances of `*(IWV$$Code)` in scatter-loading description files from earlier versions of ARM tools are automatically translated into `*(Veneer$$Code)`. Use `*(Veneer$$Code)` in new descriptions.

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

5.3.2 Creating root execution regions

If you specify an initial entry point for an image, or if the linker creates an initial entry point because you have used only one `ENTRY` directive, you must ensure that the entry point is located in a root region. A root region is a region having the same load and execution address. If the initial entry point is not in a root region, the link fails and the linker gives an error message such as:

```
Entry point (0x00000000) lies within non root region ER_ROM
```

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

- specify `ABSOLUTE`, either explicitly or by permitting it to default, as the attribute for the execution region and use the same address for the first execution region and the enclosing load region. To make the execution region address the same as the load region address, either:
  - specify the same numeric value for both the base address for the execution region and the base address for the load region
  - specify a `+0` offset for the first execution region in the load region.

    If an offset of zero (`+0`) is specified for all subsequent execution regions in the load region, they are all root regions.

  See Example 5-8.
- use the `FIXED` execution region attribute to ensure that the load address and execution address of a specific region are the same. See Example 5-9 and Figure 5-9 on page 5-34.

You can use the `FIXED` attribute to place any execution region at a specific address in ROM. See *Placing regions at fixed addresses* on page 5-34 for more information.

Example 5-8 Specifying the same load and execution address

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

Example 5-9 Using the FIXED attribute

```
LR_1 0x040000              ; load region starts at 0x40000
{                          ; start of execution region descriptions
    ER_RO 0x040000         ; load address = execution address
    {
        * (+RO)            ; RO sections other than those in init.o
    }
```

```
    ER_INIT 0x080000 FIXED ; load address and execution address of this
                           ; execution region are fixed at 0x80000
    {
        init.o(+RO)        ; all RO sections from init.o
    }
    ...                    ; rest of scatter description
}
```

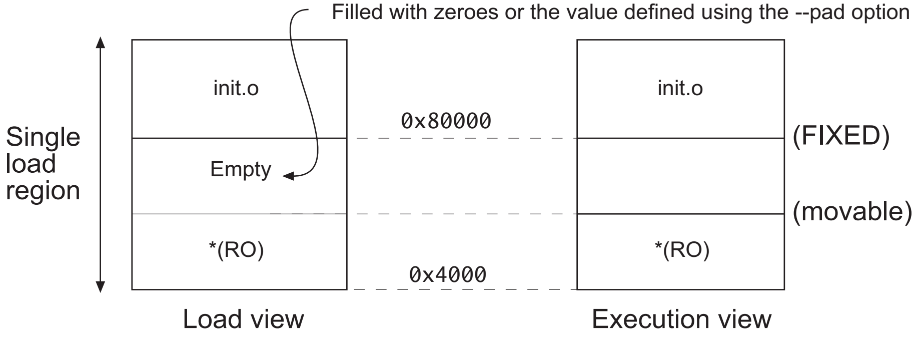


Figure 5-9 Memory Map for fixed execution regions

5.3.3 Placing regions at fixed addresses

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

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

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

——— **Note** ———

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

You cannot specify component objects that have been partially linked. For example, if you partially link the objects `obj1.o`, `obj2.o`, and `obj3.o` together to produce `obj_all.o`, the resulting component object names are discarded in the resulting object. Therefore, you cannot refer to one of the objects by name, for example, `obj1.o`. You can only refer to the combined object `obj_all.o`.

Placing functions and data at specific addresses

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

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

- An execution region can be created at the desired address with a section description that selects just one section.
- For a specially named section the linker can get the placement address from the section name. These specially named sections are called `__at` sections. See *Using __at sections to place sections at a specific address* on page 5-37 for more information.

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

- Place the function or data item in its own source file.
- Use the `--split_sections` compiler option to produce an object file for each function. See *--split_sections* on page 2-83 in the *Compiler Reference Guide*.

  This option increases code size slightly for some functions because it reduces the potential for sharing addresses, data, and string literals between functions. However, this can help to reduce the final image size overall by enabling the linker to remove unused functions when you specify `armlink --remove`.
- Use `__attribute__((section(”name”)))` to create multiple named sections. See *__attribute__((section))* on page 4-44 in the *Compiler Reference Guide*.
- Use the `AREA` directive from assembly language. In assembly code, the smallest locatable unit is an `AREA`. See the *Assembler Guide* for more information.

Placing a named section explicitly using scatter-loading

The scatter-loading description file in Example 5-10 places:

- initialization code at address `0x0` followed by the remainder of the RO code and all of the RO data except for the RO data in the object `data.o`
- all global RW variables in RAM at `0x400000`
- a table of `RO-DATA` from `data.o` fixed at address `0x1FF00`.

Example 5-10 Section placement

```
LR1 0x0 0x10000
{
    ER1 0x0 0x2000               ; Root Region, containing init code
    {                            ; place init code at exactly 0x0
        init.o (Init, +FIRST)
        * (+RO)                  ; rest of code and read-only data
    }
    RAM  0x400000                ; RW & ZI data to be placed at 0x400000
    {
        * (+RW +ZI)
    }
    DATABLOCK 0x1FF00 FIXED 0xFF  ; execution region fixed at 0x1FF00
    {                            ; maximum space available for table is 0xFF
        data.o(+RO-DATA)         ; place RO data between 0x1FF00 and 0x1FFFF
    }
}
```

Note

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

- If your loader can handle multiple load regions, place the RO code or data in its own load region.
- If you do not require the function or data to be at a fixed location in ROM, use `ABSOLUTE` instead of `FIXED`. The loader then copies the data from the load region to the specified address in RAM. `ABSOLUTE` is the default attribute.
- To place a data structure at the location of memory-mapped I/O, use two load regions and specify `UNINIT`. `UNINIT` ensures that the memory locations are not initialized to zero. See Chapter 2 *Embedded Software Development* in the *Developer Guide* for more information.

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

Placing a code or data object in its own source file and then placing the object file sections uses standard coding techniques. However, you can also use `__attribute__((section("name")))` and a scatter-loading description file to place named sections. Create a module, for example, `adder.c` and name a section explicitly as shown in Example 5-11.

Example 5-11 Naming a section

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

Use a scatter-loading description file to specify where the named section is placed, see Example 5-12. If both code and data sections have the same name, the code section is placed first.

Example 5-12 Placing a section

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

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

Using __at sections to place sections at a specific address

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

`.ARM.__at_`*`address`*

Where:

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

In the compiler, variables can be assigned to `__at` sections by either explicitly naming the section using the attribute `section()` or by using the attribute `__at` which sets up the name of the section for you. For example, the following statements place variable in a section called `.ARM.__at_0x8000`:

```
int variable __attribute__((section(".ARM.__at_0x8000"))) = 10;
int variable __attribute__((__at(0x8000))) = 10;
```

See *Variable attributes* on page 4-40 in the *Compiler Reference Guide*.

Restrictions

- `__at` section address ranges must not overlap, unless the overlapping sections are placed in different overlay regions
- `__at` sections are not permitted in position independent execution regions
- you must not reference the linker-defined symbols `$$Base`, `$$Limit` and `$$Length` of an `__at` section
- `__at` sections must not be used in System V and BPABI executables and BPABI DLLs
- `__at` sections must have an address that is a multiple of their alignment
- `__at` sections ignore any `+FIRST` or `+LAST` ordering constraints.

Manual placement

The standard scatter-loading rules are used to determine which execution regions are used to place `__at` sections.

Example 5-13 shows the placement of read-only sections `.ARM.__at_0x2000` and the read-write section `.ARM.__at_0x4000`. Load and execution regions are not created automatically in manual mode. An error is produced if an `__at` section cannot be placed in an execution region.

Example 5-13 Manual placement of __at sections in a scatter file

```
LR1 0x0
{
    ER_RO 0x0 0x2000
    {
        *(+RO)              ; .ARM.__at_0x0 is selected by +RO
    }
    ER_RO2 0x2000
    {
```

```
        *(.ARM.__at_0x2000)  ; .ARM.__at_0x2000 is selected by .ARM.__at_0x2000
    }
    ER2 0x4000
    {
        *(+RW +ZI)           ; .ARM.__at_0x4000 is selected by +RW
    }
}
```

Automatic placement

This mode is enabled by using the linker command-line option, `--autoat`. See *Specifying memory map information for the image* on page 2-12 for more information.

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

All linker `--autoat` created execution regions have the `UNINIT` scatter-loading attribute. If you require a ZI `__at` section to be zero-initialized then it must be placed within a compatible region.

A compatible region is one where:

- The `__at` address lies within the execution region base and limit, where limit is the base address + maximum size of execution region. If no maximum size is set, the limit is assumed to be infinite.
- The execution region meets at least one of the following conditions:
  - it has a selector that matches the `__at` section by the standard scatter-loading rules
  - it has at least one section of the same type (RO, RW or ZI) as the `__at` section
  - it does not have the `EMPTY` attribute.

  Note

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

Example 5-14 on page 5-40 show the sections `.ARM.__at_0x0` type RO, `.ARM.__at_0x2000` type RW, `.ARM.__at_0x4000` type ZI and `.ARM.__at_0x8000` type ZI.

Example 5-14 Automatic placement of __at sections

```
LR1 0x0
{
    ER_RO 0x0 0x2000
    {
        *(+RO)      ; .ARM.__at_0x0 lies within the bounds of ER_RO
    }
    ER_RW 0x2000 0x2000
    {
        *(+RW)      ; .ARM.__at_0x2000 lies within the bounds of ER_RW
    }
    ER_ZI 0x4000 0x2000
    {
        *(+ZI)      ; .ARM.__at_0x4000 lies within the bounds of ER_ZI
    }
}

; the linker creates a load and execution region for the __at section
; .ARM.__at_0x8000 because it lies outside all candidate regions.
```

Placing a key in flash memory

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

Assuming a device has flash memory from `0x8000` to `0x10000` and a key is required in address `0x9000`. To do this with an `__at` section, you must define a section `.ARM.__at_0x9000` that contains the value of the key. For example:

```
int key __attribute__((__at(0x9000))) = 10;
```

Example 5-15 shows a scatter file with manual placement of the flash execution region.

Example 5-15 Manual placement of flash execution regions

```
ER_FLASH 0x8000 0x2000
{
    *(+RO)                    ; other code, read-only data, and padding if reqd
    *(.ARM.__at_0x9000)       ; key
}
```

Example 5-16 on page 5-41 shows a scatter file with automatic placement of the flash execution region. Use the linker command-line option `--autoat` to enable automatic placement.

Example 5-16 Automatic placement of flash execution regions

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

Mapping a structure over a peripheral register

To place an uninitialized variable over a peripheral register, a ZI `__at` section can be used. Assuming a register is available for use at `0x10000000`. Define a ZI `__at` section called `.ARM.__at_0x10000000`.

Example 5-17 shows the a scatter file with the manual placement of the ZI `__at` section.

Example 5-17 Manual placement of ZI __at sections

```
ER_PERIPHERAL 0x10000000 UNINIT
{
    *(.ARM.__at_10000000)
}
```

Using automatic placement, assuming that there is no other execution region near `0x10000000`, the linker automatically creates a region with the `UNINIT` attribute at `0x10000000`.

5.3.4 Using overlays to place sections

You can use the `OVERLAY` attribute in a scatter-loading description file to place multiple execution regions at the same address. Example 5-18 defines a static section in RAM followed by a series of overlays. Here, only one of these sections is instantiated at runtime.

Example 5-18 Specifying a root region

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

If the length of the static area is unknown, use a zero relative offset to specify the start address of an overlay so that it is placed immediately after the end of the static section, for example:

```
OVERLAY_A_RAM +0 OVERLAY
```

In this case, consecutive overlay regions with the same `+offset` are placed at `+offset` bytes from the previous non-overlay region or start of load region. Do this to avoid unused RAM (where the static area is small) or to prevent overwriting the static area with an overlay (where the static area is large).

5.3.5 Assigning sections to a root region

In RVCT v2.1 and earlier, the only library sections that must be placed in a root region, are `__main` and the region tables. However, with the implementation of RW data compression there are more sections that must be placed in a root region. The linker can place all these sections automatically with `InRoot$$Sections`.

Use a scatter-loading description file to specify a root section in the same way as a named section. Example 5-19 uses the section selector `InRoot$$Sections` to place all sections that must be in a root region in a region called `ER_ROOT`.

Example 5-19 Specifying a root region

```
LR_FLASH 0x0
{
    ER_ROOT 0x0                        ; root region at 0x0
    {
        vectors.o (Vectors, +FIRST)    ; vector table
        * (InRoot$$Sections)           ; all library sections that must be
                                       ; in a root region
    }
    ...                                ; rest of scatter description...
}
```

5.3.6 Reserving an empty region

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

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

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

If the length is given as a negative value, the address is taken to be the end address of the region. This must be an absolute address and not a relative one. For example, the execution region definition `STACK 0x800000 EMPTY -0x10000` shown in Example 5-20 on page 5-44 defines a region called `STACK` that starts at address `0x7F0000` and ends at address `0x800000`.

Note

The dummy ZI region that is created for an `EMPTY` execution region is not initialized to zero at runtime.

If the address is in relative (+*n*) form and the length is negative, the linker generates an error.

Example 5-20 Reserving a region for the stack

```
LR_1 0x80000                          ; load region starts at 0x80000
{
    STACK 0x800000 EMPTY -0x10000     ; region ends at 0x800000 because of the
                                      ; negative length. The start of the region
                                      ; is calculated using the length.
    {
                                      ; Empty region used to place stack
    }
    HEAP +0 EMPTY 0x10000             ; region starts at the end of previous
                                      ; region. End of region calculated using
                                      ; positive length
    {
                                      ; Empty region used to place heap
    }
    ...                               ; rest of scatter description...
}
```

Figure 5-10 is a diagrammatic representation of this example.

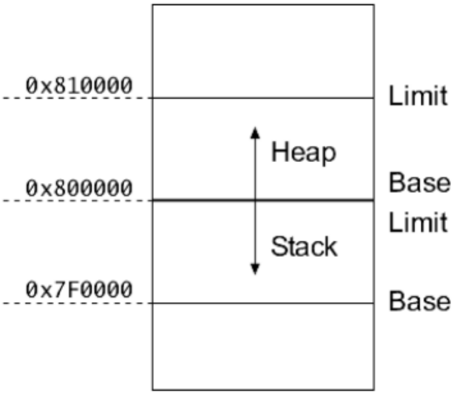


Figure 5-10 Reserving a region for the stack

In this example, the linker generates the symbols:

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

Note

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

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

5.3.7 Placing ARM libraries

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

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

Example 5-21 shows how to place library code.

Note

In Example 5-21, forward slashes are used in path names to ensure they are understood on Windows and Unix platforms.

Example 5-21 Placing ARM library code

```
ROM1 0
{
    * (InRoot$$Sections)
    * (+RO)
    ROM2 0x1000
    {
        *armlib/c_* (+RO)         ; all ARM-supplied C library functions
    }
```

```
}
ROM3 0x2000
{
    *armlib/h_* (+RO)                  ; just the ARM-supplied __ARM_*
                                       ; redistributable library functions
}
RAM1 0x3000
{
    *armlib* (+RO)                     ; all other ARM-supplied library code
                                       ; e.g. floating-point libraries
}
RAM2 0x4000
{
    * (+RW, +ZI)
}
```

5.3.8 Creating regions on page boundaries

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

Example 5-22 assumes a page size of 65536 and produces an ELF file with each region starting on a new page.

Example 5-22 Creating regions on page boundaries

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

5.3.9 Using preprocessing directives

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

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

Most typically the command is `#! armcc -E`.

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

You can add preprocessing directives to the top of the scatter-loading description file. For example:

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

The linker parses the preprocessed scatter-loading description file where these are treated as comments and ignored.

Consider the following simple example:

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

Use the directives:

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

If the scatter-loading description file contains:

```
LOAD_FLASH AN_ADDRESS    ; start address
```

Following preprocessing, this evaluates to:

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

Following evaluation, the linker parses the scatter-loading file to produce the load region:

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

See *Specifying memory map information for the image* on page 2-12 for more information on using `--predefine` command-line option.

5.4 Equivalent scatter-loading descriptions for simple images

The command-line options `--reloc`, `--ro-base`, `--rw-base`, `--ropi`, `--rwpi`, and `--split` create the simple image types described in *Using command-line options to create simple images* on page 3-26. You can create the same image types by using the `--scatter` command-line option and a file containing one of the corresponding scatter-loading descriptions.

5.4.1 Type 1, one load region and contiguous execution regions

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

`--ro-base` *`address`* specifies the load and execution address of the region containing the RO output section. Example 5-23 shows the scatter-loading description equivalent to using `--ro-base 0x040000`.

Example 5-23 Single load region and contiguous execution regions

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

The description shown in Example 5-23 creates an image with one load region called `LR_1`, whose load address is `0x040000`.

The image has three execution regions, named `ER_RO`, `ER_RW`, and `ER_ZI`, that contain the RO, RW, and ZI output sections respectively. RO, RW are root regions. ZI is created dynamically at runtime. The execution address of `ER_RO` is `0x040000`. All three execution regions are placed contiguously in the memory map by using the *`+offset`* form of the base designator for the execution region description. This enables an execution region to be placed immediately following the end of the preceding execution region.

The `--reloc` option is used to make relocatable images. Used on its own, `--reloc` makes an image similar to simple type 1, but the single load region has the `RELOC` attribute.

ropi example variant

In this variant, the execution regions are placed contiguously in the memory map. However, `--ropi` marks the load and execution regions containing the RO output section as position-independent.

Example 5-24 shows the scatter-loading description equivalent to using `--ro-base 0x010000 --ropi`.

Example 5-24 Position-independent code

```
LR_1 0x010000 PI       ; The first load region is at 0x010000.
{
    ER_RO +0           ; The PI attribute is inherited from parent.
                       ; The default execution address is 0x010000, but the code can be moved.
    {
        * (+RO)        ; All the RO sections go here.
    }
    ER_RW +0 ABSOLUTE  ; PI attribute is overridden by ABSOLUTE.
    {
        * (+RW)        ; The RW sections are placed next. They cannot be moved.
    }
    ER_ZI +0           ; ER_ZI region placed after ER_RW region.
    {
        * (+ZI)        ; All the ZI sections are placed consecutively here.
    }
}
```

Shown in Example 5-24, `ER_RO`, the RO execution region, inherits the `PI` attribute from the load region `LR_1`. The next execution region, `ER_RW`, is marked as `ABSOLUTE` and uses the *`+offset`* form of base designator. This prevents `ER_RW` from inheriting the `PI` attribute from `ER_RO`. Also, because the `ER_ZI` region has an offset of `+0`, it inherits the `ABSOLUTE` attribute from the `ER_RW` region.

5.4.2 Type 2, one load region and non contiguous execution regions

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

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

Example 5-25 shows the scatter-loading description equivalent to using `--ro-base 0x010000 --rw-base 0x040000`.

Example 5-25 Single load region and multiple execution regions

```
LR_1 0x010000       ; Defines the load region name as LR_1
{
    ER_RO +0        ; The first execution region is called ER_RO and starts at end of previous region.
                    ; Since there is no previous region, the address is 0x010000.
    {
        * (+RO)     ; All RO sections are placed consecutively into this region.
    }
    ER_RW 0x040000  ; Second execution region is called ER_RW and starts at 0x040000.
    {
        * (+RW)     ; All RW sections are placed consecutively into this region.
    }
    ER_ZI +0        ; The last execution region is called ER_ZI.
                    ; The address is 0x040000 + size of ER_RW region.
    {
        * (+ZI)     ; All ZI sections are placed consecutively here.
    }
}
```

This description creates an image with one load region, named `LR_1`, with a load address of `0x010000`.

The image has three execution regions, named `ER_RO`, `ER_RW`, and `ER_ZI`, that contain the RO, RW, and ZI output sections respectively. The RO region is a root region. The execution address of `ER_RO` is `0x010000`.

The `ER_RW` execution region is not contiguous with `ER_RO`. Its execution address is `0x040000`.

The `ER_ZI` execution region is placed immediately following the end of the preceding execution region, `ER_RW`.

rwpi example variant

This is similar to images of type 2 with `--rw-base` with the RW execution region separate from the RO execution region. However, `--rwpi` marks the execution regions containing the RW output section as position-independent.

Example 5-26 shows the scatter-loading description equivalent to using `--ro-base 0x010000 --rw-base 0x018000 --rwpi`.

Example 5-26 Position-independent data

```
LR_1 0x010000           ; The first load region is at 0x010000.
{
    ER_RO +0            ; Default ABSOLUTE attribute is inherited from parent. The execution address
                        ; is 0x010000. The code and ro data cannot be moved.
    {
        * (+RO)         ; All the RO sections go here.
    }
    ER_RW 0x018000 PI   ; PI attribute overrides ABSOLUTE
    {
        * (+RW)         ; The RW sections are placed at 0x018000 and they can be moved.
    }
    ER_ZI +0            ; ER_ZI region placed after ER_RW region.
    {
        * (+ZI)         ; All the ZI sections are placed consecutively here.
    }
}
```

`ER_RO`, the RO execution region, inherits the `ABSOLUTE` attribute from the load region `LR_1`. The next execution region, `ER_RW`, is marked as `PI`. Also, because the `ER_ZI` region has an offset of `+0`, it inherits the `PI` attribute from the `ER_RW` region.

Similar scatter-loading descriptions can also be written to correspond to the usage of other combinations of `--ropi` and `--rwpi` with type 2 and type 3 images.

5.4.3 Type 3, two load regions and non contiguous execution regions

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

Relocate and split load regions using the following linker options:

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

`--ro-base` *`address1`*

Specifies the load and execution address of the region containing the RO output section.

`--rw-base` *`address2`*

Specifies the load and execution address for the region containing the RW output section.

`--split`

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

Example 5-27 shows the scatter-loading description equivalent to using `--ro-base 0x010000` `--rw-base 0x040000` `--split`.

In this example:

- This description creates an image with two load regions, named `LR_1` and `LR_2`, that have load addresses `0x010000` and `0x040000`.
- The image has three execution regions, named `ER_RO`, `ER_RW` and `ER_ZI`, that contain the RO, RW, and ZI output sections respectively. The execution address of `ER_RO` is `0x010000`.
- The `ER_RW` execution region is not contiguous with `ER_RO`. Its execution address is `0x040000`.
- The `ER_ZI` execution region is placed immediately following the end of the preceding execution region, `ER_RW`.

Example 5-27 Multiple load regions

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

```
    }
    ER_ZI +0      ; The address is 0x040000 + size of ER_RW region.
    {
        * (+ZI)   ; All ZI sections are placed consecutively into this region.
    }
}
```

Relocatable load regions example variant

This type 3 image also consists of two load regions in load view and three execution regions in execution view. However, `--reloc` is used to specify that the two load regions now have the `RELOC` attribute.

Example 5-28 shows the scatter-loading description equivalent to using `--ro-base 0x010000` `--rw-base 0x040000` `--reloc --split`.

Example 5-28 Relocatable load regions

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

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

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

Chapter 6

BPABI and System V Shared Libraries and Executables

This chapter describes how the ARM linker, `armlink`, supports BPABI and System V shared libraries and executables. It contains the following sections:

- *The Base Platform ABI* on page 6-2
- *Using BPABI shared libraries and executables* on page 6-3
- *Using SVr4 shared libraries and executables* on page 6-7.

6.1 The Base Platform ABI

The BPABI governs the format and content of executable and shared object files generated by static linkers. It supports platform-specific executable files using post linking and provides a base standard that is used to derive a platform ABI. The standard defines three platform families based on the shared object model:

- Bare metal
- DLL-like
- System V release 4 (SVr4).

The linker conforms to the BPABI and so enables you to:

- link a collection of objects and libraries into a:
  - Bare metal executable image
  - BPABI DLL or SVr4 shared object
  - BPABI or SVr4 executable file.
- link a collection of objects against shared libraries
- partially link a collection of objects into an object that can be used as input to a subsequent link step.

6.1.1 Getting more information

For information about the base standard, software interfaces, and standards supported by ARM, see `*install_directory*\Documentation\Specifications\...`.

For information on the latest published versions, see `http://www.arm.com`.

See ARM *Application Note 178 (ARM DAI 0178) Building Linux Applications Using RVDS 3.1 and the GNU Tool and Libraries*.

For more information on the generic System V ABI, see `http://www.sco.com`.

6.2 Using BPABI shared libraries and executables

The linker enables you to build BPABI shared libraries and to link objects against shared libraries.

6.2.1 About the BPABI

The BPABI standardizes the interface between executable files, including DLLs and *Dynamic Shared Objects* (DSOs), and their execution environment or platform. The ABI defines four platform families depending on how they manage dynamically-loaded executables:

- a bare platform, corresponding to a fully static link, with no dynamically loaded binaries
- a platform using DLLs loaded and used by multiple processes in a single address space
- a platform using DLLs by mapping their segments at the same address in different virtual address spaces
- a platform like System V, where a dynamic shared object can be mapped at a different address in each different virtual address space.

Each specific platform builds on the underlying models provided by the BPABI to add their individual requirements, such as application standards and build parameters. These are used by a post-processing tool, for example a post-linker, specific to your platform to transform a BPABI executable or shared object into a complete executable or shared object for that execution environment.

The BPABI also supports different addressing modes corresponding to the different ways in which code must reference static data under the different platform families. Code can address static data directly, or indirectly through a separate dynamic relocation, and the addressing can be absolute, relative to the PC, or relative to a static base held in register `r9`.

See the BPABI for more information on the platform categories, addressing modes and dynamic linking issues.

6.2.2 Symbol import and export

Executables and DLLs can export symbols that they define for other Executables and DLLs to use. They can also import symbols from other DLLs.

The recommended method of specifying imported symbols in a BPABI executable or DLL is to use the `__declspec(dllimport)` attribute in the function declaration, as the generated code accesses the reference in the correct manner. However, you can also use the `IMPORT` directive in a linker steering file to import symbols when you do not have access to a DLL defining the symbol.

To mark symbols for dynamic export, you can use the `EXPORT` attribute in a steering file or the `__declspec(dllexport)` attribute in the symbol declaration. The `--export_all` switch can be used to dynamically export all global non hidden symbols from a BPABI executable or DLL. If you do not specify any steering file commands, the linker exports all global non hidden symbols by default.

The linker imports undefined symbol references when it finds a matching definition in the dynamic symbol table of a shared object that you specify on the command line. These symbols are then considered to be imported from the shared object, or in the case of a BPABI, from the DLL.

A non hidden symbol is one that has the `DYNAMIC` or `PROTECTED` visibility attribute in assembler source, or where the C source code contains the `__declspec(dllimport)` or `__declspec(dllexport)` attributes.

For more information see:

- *Hiding and renaming global symbols* on page 4-11
- *__declspec attributes* on page 4-24 in the *Compiler Reference Guide*.

6.2.3 Symbol visibility

The linker supports symbol visibility. Symbol visibility affects the static linker and the dynamic linker. This section describes the effect of each visibility on armlink. See the ARM ELF specification for more information.

6.2.4 Symbol Versioning

The linker supports symbol versions. This provides more useful information for the symbol table:

- Symbols with local scope in a versioned shared object are not referred to from the outside.
- A global versioned symbol has no version, and so the usual symbol matching applies.
- A versioned symbol with the `HIDDEN` visibility attribute set is a deprecated versioned symbol. The static linker ignores this.

- A versioned symbol where the `HIDDEN` visibility attribute is not set is the default symbol.

Symbol versioning information is added to the symbol table when a shared object is loaded that contains symbol versioning tables, and references to versioned symbols are matched. You can use a version script file to specify a list of exported symbols using the `--symver_script` option. See `--symver_script` in *Controlling image contents* on page 2-16 for more information.

6.2.5 Memory map

For both BPABI executable and DLL images, the image layout corresponds to the following scatter-file description:

```
LR_1 <read-only base address>
{
    ER_RO  +0
    {
        *(+RO)
    }
}
LR_2 <read-write base address>
{
    ER_RW  +0
    {
        *(+RW)
    }
    ER_ZI  +0
    {
        *(+ZI)
    }
}
```

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 must specify the `--ro-base` and `--rw-base` switches, as the default values might not be suitable for your platform. These addresses need not reflect the actual addresses to which the image is relocated at run time.

For more information see:

- `--ropi` in *Specifying memory map information for the image* on page 2-12
- `--rwpi` in *Specifying memory map information for the image* on page 2-12
- `--ro-base` in *Specifying memory map information for the image* on page 2-12

- `--rw-base` in *Specifying memory map information for the image* on page 2-12.

6.3 Using SVr4 shared libraries and executables

The linker enables you to build SVr4 shared libraries and to link objects against shared libraries.

6.3.1 Building an ARM Linux executable

Use the `--sysv` command-line option to generate an SVr4 formatted ELF executable file that can be used on ARM Linux.

Note

If you use `--sysv`, the linker ignores any scatter file that you specify on the command line.

The base of an executable is `0x8000` by default. When shared objects are given on the command line the linker uses these to resolve references and create a dynamic executable.

When the linker finds a shared object on the command line, it is included in the list of libraries to be added to the executable file. See *Library searching, selection, and scanning* on page 7-3 for more information.

If you are working on ARM Linux, be aware of the following:

- The Linux kernel performs all copying and zero initialization of the executable file.
- RW data is not compressed.
- An executable is always entered in ARM state.

Building a shared object

A shared object provides an extension of the static and dynamic linking described in *Building an ARM Linux executable* on page 6-7. The base address of the load region is set at `0` and is then relocated by the Linux dynamic linker.

If your shared object contains any exported RW data, you are required to use position independent code and data. In this case, you must compile or assemble your files using `--apcs /fpic`, and link the files into a shared object using the `--fpic` linker option.

Use the `--shared` command-line option to build an SVr4 shared object.

Note

A shared object usually has no entry point. However, it is possible to set an entry point. This must be done if the object you are building is the dynamic linker.

Using the Linux ABI tag

To comply with the Linux Standard Base Specification v1.2, an executable file must contain a section named `.note.ABI-tag` of type `SHT_NOTE`, structured as a note section as documented in the ELF specification.

You can use the command-line option `--linux_abitag` to specify the *minimum* compatible kernel version for the executable file you are building, for example:

```
armlink ... --sysv --linux_abitag 2.2.5 main.o
```

This links `main.o` into a static executable that is defined as being compatible with Linux kernel v2.2.5 or later. If you specify any shared objects on the command line that demand a *newer* kernel, the kernel requirements in the output file are incremented to match.

For more information on using the Linux ABI tag and Standard Base Specification, see `http://www.linuxbase.org`.

6.3.2 Accessing symbols

The symbol tables provide a way to determine those symbols in shared objects that are referenced by other non shared objects included in the link stage. Where a reference to a symbol exists, it is defined as having been imported from the shared object.

The linker supports symbol versions. This provides more useful information for the symbol table:

- Symbols with local scope in a versioned shared object are not referred to from the outside.
- A global versioned symbol has no version, and so the usual symbol matching applies.
- A versioned symbol with the `HIDDEN` visibility attribute set is a deprecated versioned symbol. The static linker ignores this.
- A versioned symbol where the `HIDDEN` visibility attribute is not set is the default symbol.

Symbol versioning information is added to the symbol table when a shared object is loaded that contains symbol versioning tables (and references to versioned symbols are matched). You can use a version script file to specify a list of exported symbols, for example:

```
armlink file_1.o file_2.o --sysv --shared -o libfoo.so --symver_script ver_script.txt
```

Symbol resolution

The linker resolves symbols in shared and non shared objects in the same way. When an undefined reference matches a definition in a shared object, the linker imports the reference by placing it in the dynamic symbol table.

Importing and exporting symbols

The linker imports undefined symbol references when it finds a matching definition in the dynamic symbol table of a shared object that you specify on the command line. These symbols are then considered to be exported symbols.

When building a shared object, only those symbols marked for export by steering file directives, by the presence of `__declspec(dllexport)` in the source file, or by the `--no_hide_all` option are exported. If you do not specify any steering file commands, the linker exports all global (non hidden) symbols by default. A non hidden symbol is one that has the `DYNAMIC` or `PROTECTED` visibility attribute in assembler source, or where the C source code contains `__declspec(dllimport)` or `__declspec(dllexport)`. To mark all symbols as non hidden, use the `--no_hide_all` option.

When building an executable file, only the symbols required to execute the image correctly on a Linux platform are exported. That is, the linker imports any symbol found in a shared object. Steering file commands can be used to define additional symbols to be inserted into the dynamic symbol table.

For Linux executables and shared libraries, the standard approach to importing and exporting symbols is to use the `--no_hide_all` compiler or assembler option to mark all symbols as candidates for dynamic import or export.

To mark individual symbols as candidates for dynamic import and export, use a steering file. Use the `EXPORT` directive within the file to specify exported symbols.

Note

Be aware that `armlink` generates an error for any undefined references that remain.

See *Steering file commands* on page 4-12 for more information on using `EXPORT`.

6.3.3 Exception tables

In a static image that does not use shared libraries, the linker automatically discards exception tables if it decides that the image cannot throw an exception.

When linking an image that uses shared objects, use the `--force_so_throw` command-line option to specify that all shared objects might throw exceptions and so force the linker to keep the exception tables, regardless of whether the image can throw an exception or not.

6.3.4 Thread Local Storage

In the current release, the linker supports *Thread Local Storage* (TLS) for SVr4 images and shared libraries only. For more information on the linker implementation, see the *Addenda to, and Errata in, the ABI for the ARM Architecture* (ABI-addenda).

6.3.5 Using a dynamic linker

A shared object or executable file contains all the information necessary for the dynamic linker to load and run the file correctly:

- Every shared object contains a `SONAME` that identifies the object. You can specify this name on the command line using the `--soname` *`name`* option.
- The linker identifies dependencies to other shared objects using the shared objects specified on the command line. These shared object dependencies are encoded in `DT_NEEDED` tags. In the current release, the linker orders these tags to match the order of the libraries on the command line.
- The dynamic linker searches for dependencies using a pre-determined search path. Use the `--runpath` *`pathlist`* command-line option to add the paths in *`pathlist`* to the search path. The search path is encoded using the `DT_RUNPATH` tag.
- When building a shared library using `--sysv --shared` the linker does not include the ARM C libraries initialization function `__cpp_initialize__aeabi_` by default. Instead the linker sets the `DT_INIT_ARRAY` tags if appropriate, so that the dynamic linker can initialize the library.

  If you prefer to use the `__cpp_initialize__aeabi_` function to initialize your shared library then you must add `--ref_cpp_init` to the command line and set `--init=__cpp_initialize_aeabi_`.
- If you specify the `--fini` *`symbol`* command-line option, the linker uses the specified symbol name to define finalization code. The dynamic linker executes this code when it unloads the executable file or shared object.

  There is no assumption that a symbol named `_fini` marks this code.
- If you specify the `--fini` *`symbol`* command-line option, the linker uses the specified symbol name to define finalization code. The dynamic linker executes this code when it unloads the executable file or shared object.

  There is no assumption that a symbol named `_fini` marks this code.

Use the `--dynamiclinker` *`name`* command-line option to specify the dynamic linker to use to load and relocate the file at runtime. If you are working on Linux platforms, the linker assumes that the default dynamic linker is `/lib/ld-linux.so.2`.

Chapter 7
Using the ARM Librarian

This chapter describes the use of libraries with the ARM linker, `armlink`, and the librarian, `armar`. It contains the following sections:

- *About libraries* on page 7-2
- *Library searching, selection, and scanning* on page 7-3
- *The ARM librarian* on page 7-7.

7.1 About libraries

An object file can refer to external symbols that are, for example, functions or variables. The linker attempts to resolve these references by matching them to definitions found in other object files and libraries. The linker recognizes a collection of ELF files stored in an `ar` format file as a library. The contents of each ELF file forms a single member in the library.

If you use `--sysv` to generate an SVr4 formatted ELF executable file, the linker treats a shared object as a library. Similarly, a shared object or DLL is treated as a library when you are generating a BPABI-compatible executable file. However, a shared object or DLL differs from an archive in that:

- symbols are imported from a shared object or DLL
- code or data for symbols is extracted from an archive into the file being linked.

The rest of this chapter describes archives.

7.2 Library searching, selection, and scanning

The differences between the way the linker adds object files to the image and the way it adds libraries to the image are:

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

  Note

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

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

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

Note

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

Therefore, the linker creates a list of libraries as follows:

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

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

7.2.1 Searching for ARM libraries

You can specify the search paths used to find the ARM standard libraries by:

- Using the environment variable `RVCT31LIB`. This is the default.
- Adding the `--libpath` option to the `armlink` command line with a comma-separated list of parent directories.

  This list must end with the parent directory of the ARM library directories `armlib` and `cpplib`. The `RVCT31LIB` environment variable holds this path.

  Note

  The linker command-line option `--libpath` overrides the paths specified by the `RVCT31LIB` variable.

The linker combines each parent directory, given by either `--libpath` or the `RVCT31LIB` variable, with each subdirectory request from the input objects and identifies the place to search for the ARM library. The names of ARM subdirectories within the parent directories are placed in each compiled object by using a symbol of the form `Lib$$Request$$`*`sub_dir_name`*.

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

Selecting ARM library variants

There are different variants of the ARM libraries based on the attributes of their member objects. The variant of the ARM library is coded into the library name. The linker must select the best-suited variant from each of the directories identified during the library search.

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

The final list contains all the libraries that the linker scans in order to resolve references.

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

7.2.2 Searching for user libraries

You can specify user libraries by:

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

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

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

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

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

Note

The search paths used for the ARM standard libraries specified by the `RVCT31LIB` environment variable or the linker command-line option `--libpath` are not searched for user libraries (see *Searching for ARM libraries* on page 7-4).

7.2.3 Scanning the libraries

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

When all the directories have been searched, and the most compatible library variants have been selected and added to the list of libraries, each of the libraries is scanned to load the required members:

1. For each currently unsatisfied non weak reference, the linker searches sequentially through the list of libraries for a matching definition. The first definition found is marked for step 2.

   The sequential nature of the search ensures that the linker chooses the library that appears earlier in the list if two or more libraries define the same symbol. This enables you to override function definitions from other libraries, for example, the ARM C libraries, by adding your libraries to the input file list. However you must be careful to consistently override all the symbols in a library member or the behavior is unpredictable.

2. Library members marked in step 1 are loaded. As each member is loaded it might satisfy some unresolved references, possibly including weak ones. Loading a library might also create new unresolved weak and non weak references.
3. The process in steps 1 and 2 continues until all non weak references are either resolved or cannot be resolved by any library.

If any non weak reference remains unsatisfied at the end of the scanning operation, the linker generates an error message.

7.3 The ARM librarian

The ARM librarian, `armar`, enables sets of ELF object files or libraries to be collected together and maintained in libraries. Such a library can then be passed to the linker in place of several object files. However, linking with an object library file does not necessarily produce the same results as linking with all the object files collected into the object library file. This is because the linker processes the input list and libraries differently:

- Each object file in the input list appears in the output unconditionally, although unused areas are eliminated if the `armlink --remove` option is specified.
- A member of a library file is only included in the output if it is referred to by an object file or a previously processed library file.

For more information on how the linker processes its input files, see Chapter 2 *The Linker Command Syntax*.

7.3.1 Librarian command-line options

The syntax of the `armar` command when used to add or modify files in the library is:

```
armar --help [--create] [--diag_style arm|ide|gnu] [--[no_]project=[filename]]
[--reinitialize_workdir] [--workdir=directory] [-c] [-d] [-m] [-q] [-r] [-u]
[--vsn] [-v] [--via option_file] [{-a|-b|-i} pos_name] library [file_list]
```

The syntax of the `armar` command when used to extract files or library information is:

```
armar [--help] [--diag_style arm|ide|gnu] [-C] [--entries] [-p] [-t] [-s]
[--sizes] [-T] [--vsn] [-v] [--via option_file] [-x] [--zs] [--zt] library
[file_list]
```

where:

`-a`	Places new files in *`library`* after the file *`pos_name`*. The effect of this option is negated if you include `-b` (or `-i`) on the same command line.
`-b`	Places new files in *`library`* before the file *`pos_name`*. This option takes precedence if you include `-a` on the same command line.
`-i`	Places new files in *`library`* before the member *`pos_name`* (equivalent to `-b`). This option takes precedence if you include `-a` on the same command line.

pos_name	The name of an existing library member to be used for relative positioning. This name must be supplied with options `-a`, `-b`, and `-i`.
`-C`	Instructs the librarian not to replace existing files with like-named files when performing extractions. This option is useful when `-T` is also used to prevent truncated filenames from replacing files with the same prefix.
`-c`	Suppresses the diagnostic message normally written to standard error when a library is created.
`--create`	Creates a new library even if *library* already exists.
`-d`	Deletes one or more files from *library*.
`--diag_style arm|ide|gnu`	Change the formatting of warning and error messages. `--diag_style arm` is the default, `--diag_style gnu` matches the format reported by `gcc`, and `--diag_style ide` matches the format reported by Microsoft Visual Studio.
`--entries`	Lists all entry points defined in *library*. The format for the listing is: `ENTRY at offset` *num* `in section` *name* `of` *member*
file_list	A list of files to process. Each file is fully specified by its path and name. The path can be absolute, relative to drive and root, or relative to the current directory. Only the filename at the end of the path is used when comparing against the names of files in the library. If two or more path operands end with the same filename, the results are unspecified. You can use the wildcards `*` and `?` to specify files. If one of the files is a library, `armar` copies all members from the input library to the destination library. The order of entries on the command line is preserved. Therefore, supplying a library file is logically equivalent to supplying all of its members in the order that they are stored in the library.
`--help`	Gives help information on the `armar` command.
`--[no_]project=[`*filename*`]`	Enables or disables the use of a project template file. See *--[no_]project=filename* on page 2-74 in the *Compiler Reference Guide* for more information.

`--reinitialize_workdir`

Enables you to reinitialize the project template working directory. See *--reinitialize_workdir* on page 2-77 in the *Compiler Reference Guide* for more information.

`--workdir=`*`directory`*

Enables you to provide a working directory for a project template. See *--workdir=directory* on page 2-95 in the *Compiler Reference Guide* for more information.

`library`

Path name of the library file.

`-m`

Moves files. If `-a`, `-b`, or `-i` with *`pos_name`* is specified, files are moved to the new position. Otherwise, move files to the end of library.

`-n`

Suppresses the archive symbol table. This is used when the library is not an object library.

`-p`

Prints the contents of files in *`library`* to `stdout`.

`-q`

An alias for `-r`.

`-r`

Replaces, or adds, files in *`library`*. If *`library`* does not exist, a new library file is created and a diagnostic message is written to standard error.

If *`file_list`* is not specified and the library exists, the results are undefined. Files that replace existing files do not change the order of the library.

If the `-u` option is used, then only those files with dates of modification later than the library files are replaced.

If the `-a`, `-b`, or `-i` option is used, then *`pos_name`* must be present and specifies that new files are to be placed after (`-a`) or before (-b or `-i`) *`pos_name`*. Otherwise the new files are placed at the end.

`-t`

Prints a table of contents of *`library`*. The files specified by *`file_list`* are included in the written list. If *`file_list`* is not specified, all files in the library are included in the order of the archive.

`-s`

Regenerates the archive symbol table.

`--sizes`

Lists the `Code`, `RO Data`, `RW Data`, `ZI Data`, and `Debug` sizes of each member in *`library`*, for example:

```
Code    RO Data    RW data    ZI Data    Debug    Object Name
 464          0          0          0     8612    app_1.o
3356          0          0      10244    11848    app_2.o
3820          0          0      10244    20460    TOTAL
```

`-T`	Enables filename truncation of extracted files whose library names are longer than the file system can support. By default, extracting a file with a name that is too long is an error. A diagnostic message is written and the file is not extracted.
`-u`	Updates older files. When used with the `-r` option, files within `library` are replaced only if the corresponding file has a modification time that is at least as new as the modification time of the file within library.
`--via option_file`	Instructs the librarian to take options from `option_file`. See *Overview of via files* on page A-2 in the *Compiler Reference Guide* for more information on writing via files.
`-v`	Gives verbose output. The output depends on what other options are used: `-d, -r, -x` Write a detailed file-by-file description of the library creation, the constituent files, and maintenance activity. `-p` Writes the name of the file to the standard output before writing the file itself to the `stdout`. `-t` Includes a long listing of information about the files within the library. `-x` Prints the filename preceding each extraction.
`--vsn`	Prints the version number on standard error.
`-x`	Extracts the files in `file_list` from `library`. The contents of `library` are not changed. If no file operands are given, all files in `library` are extracted. If the filename of a file extracted from the library is longer than that supported in the destination directory, the results are undefined.
`--zs`	Shows the symbol table.
`--zt`	Lists member sizes and entry points in `library`. See `--sizes` and `--entries` for output format.

——— Note ———

The options `-a`, `-b`, `-C`, `-i`, `-m`, `-T`, `-u`, and `-v` are not required for normal operation.

Options relating to library order (for example, `-a`, `-b`, `-i`, and `-m`) are not relevant, because the ARM tools cannot use a library that does not have a symbol table. If there is a symbol table, the order is irrelevant. However, see the rules about precedence if you include `-a` and `-b` (or `-i`) on the same command line.

Options relating to updating a library (`-C` and `-u`) are unlikely to be used because, in practice, the libraries are rebuilt rather than updated.

7.3.2 Ordering command-line options

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

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

Note

In the current release of RVCT, `armar` command-line options must be preceded by a `-`. This is a change from previous releases.

7.3.3 Examples of armar usage

Syntax examples are shown in Example 7-1 to Example 7-8 on page 7-13.

Example 7-1 Create a new library and add all object files

```
armar --create mylib *.o
```

Example 7-2 List the table of contents (verbose)

```
armar -tv mylib
```

Example 7-3 List the symbol table

```
armar --zs mylib
```

Example 7-4 Add (or replace) files

```
armar -r mylib obj1.o obj2.o obj3.o ...

armar -ru mylib k*.o
```

Example 7-5 Add (or replace) files in specified position

```
armar -r -a obj2.o mylib obj3.o obj4.o ...
```

Example 7-6 Extract a group of files

```
armar -x mylib k*.o
```

Example 7-7 Delete a group of files

```
armar -d mylib sys_*
```

Example 7-8 Merge libraries and add (or replace) files

```
armar -r mylib my_lib.a other_lib.a obj1.o obj2.o obj3.o
```

Chapter 8
Using fromelf

This chapter describes the ARM `fromelf` software utility provided with RVCT. It contains the following sections:

- *About fromelf* on page 8-2
- *fromelf command syntax* on page 8-3
- *Examples of fromelf usage* on page 8-14.

8.1 About fromelf

`fromelf` translates ELF image files produced by the ARM linker into other formats suited to ROM tools and to loading directly into memory. You can also use `fromelf` to display various information about an ELF object or to generate text files containing the information.

`fromelf` outputs the following image formats:

- Plain binary
- Motorola 32-bit S-record
- Intel Hex-32
- Byte oriented (Verilog Memory Model) hexadecimal
- ELF. You can resave as ELF, for example, you can convert a debug ELF image to a no-debug ELF image.

`fromelf` can also display information about the input file, for example, disassembly output or symbol listings, to either `stdout` or a text file.

8.1.1 Image structure

`fromelf` can translate a file from ELF to other formats. It cannot change the image structure or addresses, other than altering the base address of Motorola S-record or Intel Hex output with the `--base` option. You cannot change a scatter-loaded ELF image into a non scatter-loaded image in another format. Any structural or addressing information must be provided to the linker at link time.

8.1.2 Ordering command-line options

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

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

8.2 fromelf command syntax

This section describes the syntax and options for the `fromelf` command.

The command for invoking `fromelf` is:

```
fromelf [--base n] [--debugonly] [--diag_style arm|ide|gnu]
[--diag_suppress taglist] [--expandarrays] [--fieldoffsets
[--select select_options ]] --help [--no_debug] [--no_linkview]
[--no_symbolversions] [memory_config] [--privacy ] [project-template-options]
[--strip option] [--text | code_output_format] [--vsn] [--output output_file]
input_file
```

8.2.1 `--base` *n*

Specifies the base address of the output for Motorola S-record and Intel Hex file formats. This option is available only if `--m32`, `--m32combined`, `--i32`, or `--i32combined` is specified as the output format.

You can specify the base address as either:

- a decimal value, for example, `--base 0`
- a hexadecimal value, for example, `--base 0x8000`.

All addresses encoded in the output file start at the base address *n*. If you do not specify a `--base` option, the base address is taken from the load region address.

Note

If multiple load regions are present, the `--base` value is used for each output file. That is, it overrides all load region addresses.

8.2.2 *code_output_format*

Selects the binary or ELF output file options. *code_output_format* can be one of:

`--bin`	Plain binary. You can split output from this option into multiple files with the *memory_config* option.
`--elf`	ELF format (resaves as ELF). This can be used to convert a debug ELF image into a no-debug ELF image.
`--i32`	Intel Hex-32 format. This option generates one output file for each load region in the image. You can specify the base address of the output with the `--base` option.

`--i32combined`	Intel Hex-32 format. This option generates one output file for an image containing multiple load regions. You can specify the base address of the output with the `--base` option.
`--m32`	Motorola 32-bit format (32-bit S-records). This option generates one output file for each load region in the image. You can specify the base address of the output with the `--base` option.
`--m32combined`	Motorola 32-bit format (32-bit S-records). This option generates one output file for an image containing multiple load regions. You can specify the base address of the output with the `--base` option.
`--vhx`	Byte oriented (Verilog Memory Model) hexadecimal format. This format is suitable for loading into the memory models of *Hardware Description Language* (HDL) simulators. You can split output from this option into multiple files with the *`memory_config`* option.

If you use `fromelf` to convert an ELF image containing multiple load regions to a binary format using any of the `--bin`, `--m32`, `--i32`, or `--vhx` options, `fromelf` creates an output directory named *`output_file`* and generates one binary output file for each load region in the input image. `fromelf` places the output files in the *`output_file`* directory.

Note

For multiple load regions, the name of the first non-empty execution region in the corresponding load-region is used for the file name.

If you convert an ELF image containing multiple load regions using either the `--m32combined` or `--i32combined` option, `fromelf` creates an output directory named *`output_file`*, generates one binary output file for all load regions in the input image, and then places the output file in the *`output_file`* directory.

ELF images contain multiple load regions if, for example, they are built with a scatter-loading description file that defines more than one load region.

See also

- *--base n* on page 8-3
- *--output output_file* on page 8-9.

8.2.3 `--continue_on_error`

This option downgrades all recoverable errors to warnings so that `fromelf` can process broken ELF files.

8.2.4 `--decode_build_attributes`

Prints the contents of the build attributes section in human-readable form for standard build attributes or raw hexadecimal form for nonstandard build attributes.

8.2.5 `--debugonly`

Use this option with `--elf` to remove the content of any code or data sections. Use this so that the output file contains only information required for debug, for example, debug sections, symbol table, and string table.

Section headers are retained because they are required to act as targets for symbols.

This option affects only ELF output files.

8.2.6 `--diag_style arm|ide|gnu`

Change the formatting of warning and error messages. `--diag_style arm` is the default, `--diag_style gnu` matches the format reported by `gcc`, and `--diag_style ide` matches the format reported by Microsoft Visual Studio.

8.2.7 `--diag_suppress` *`taglist`*

Disables all diagnostic messages that have the specified tag(s).

This option requires a comma-separated list identifying the number of the message to be suppressed (more than one tag can be specified). For example, to suppress the warning messages that have numbers `1293` and `187`, use the following command:

```
fromelf --diag_suppress 1293,187 ...
```

The `fromelf` prefix `Q` can be used when suppressing messages, for example:

```
fromelf --diag_suppress Q1293,Q187 ...
```

Using the prefix letter is optional. However, if a prefix letter is included, it must match the `fromelf` identification letter. If another prefix is found, the message number is ignored.

8.2.8 `--dump_build_attributes`

Prints the contents of the build attributes section in raw hexadecimal form, that is, in the same form as data.

8.2.9 `--expandarrays`

Prints data addresses, including arrays that are expanded both inside and outside structures.

This option can only be used in conjunction with `--text -a`.

8.2.10 `--extract_build_attributes`

Prints the build attributes only in human-readable form for standard build attributes or raw hexadecimal form for nonstandard build attributes.

8.2.11 `--fieldoffsets`

Prints a list of assembly language `EQU` directives that equate C++ class or C structure field names to their offsets from the base of the class or structure. The input ELF file can be a relocatable object or an image.

Use `-o` to redirect the output to a file. Use the `INCLUDE` command from `armasm` to load the produced file and provide access to C++ classes and C structure members by name from assembly language. See the *Assembler Guide* for more information on `armasm`.

Note

This option:

- is not available if the source file does not have debug information
- cannot be used together with a *`code_output_format`*.

This option outputs all structure information. To output a subset of the structures, use `--select` *`select_options`*.

If you do not require a file that can be input to `armasm`, use the `--text -a` options to format the display addresses in a more readable form. The `-a` option only outputs address information for structures and static data in images because the addresses are not known in a relocatable object.

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.

8.2.12 `--help`

Shows help and usage information. If this option is specified, other command-line options are ignored. Calling `fromelf` without any parameters produces the same help information.

8.2.13 *`input_file`*

Specifies the ELF file to be translated.

`fromelf` accepts only ARM executable ELF files and ARM object ELF files (`.o`). If *`input_file`* is a scatter-loaded image that contains more than one load region and the output format is one of `--bin`, `--m32`, `--i32`, or `--vhx`, `fromelf` creates a separate file for each load region.

If *`input_file`* is a scatter-loaded image that contains more than one load region and the output format is either `--m32combined` or `--i32combined`, `fromelf` creates a single file containing all load regions.

8.2.14 *`memory_config`*

Outputs multiple files for multiple memory banks.

The format of *`memory_config`* is `--`*`width`*`x`*`banks`* where:

`banks`	specifies the number of memory banks in the target memory system.
`width`	is the width of memory in the target memory system (8-bit, 16-bit, 32-bit, or 64-bit).

Valid configurations are:

```
--8x1
--8x2
--8x4
--16x1
--16x2
```

```
--32x1
--32x2
--64x1
```

`fromelf` uses the last specified configuration if more than one configuration is specified.

If the image has one load region, `fromelf` generates *`banks`* files with the following naming conventions:

- If there is one memory bank (*`banks`*`=`1) the output file is named by the `-o` *`output_file`* argument.
- If there are multiple memory banks (*`banks`*`>`1), `fromelf` generates *`banks`* number of files starting with *`output_file0`* and finishing with *`output_file banks`*`-1`. For example:

  ```
  fromelf --vhx --8x2 test.axf -o test
  ```

  generates two files named `test0` and `test1`.

If the image has multiple load regions, `fromelf` creates a directory named *`output_file`* and generates bank files for each load region named *`load region0`* to *`load region banks`*`-1`.

The memory width specified by *`width`* controls the size of the chunk of information read from the image and written to a file. The first chunk read is allocated to the first file (*`output_file0`*), the next chunk is allocated to the next file. After a chunk is allocated to the last file, allocation begins again with the first file (that is, the allocation is modulo based on the number of files). For example:

For a *`memory_config`* of `--8x4`

```
byte0 -> file0
byte1 -> file1
byte2 -> file2
byte3 -> file3
byte4 -> file0
…
```

For a *`memory_config`* of `--16x2`

```
halfword0 -> file0
halfword1 -> file1
halfword3 -> file0
…
```

8.2.15 `--no_debug`

Do not put debug information in the output files. This is the default for binary images. If `--no_debug` is specified, it affects all output formats. It overrides the `--text -g` option.

Note

This option can have unexpected effects if `--elf` is not specified on the command line. The following command produces a text file because no output format has been specified:

```
fromelf --no_debug image -o image_nodb.axf
```

To get ELF format output use the options:

```
fromelf --no_debug --elf image.axf -o image_ndb.axf
```

8.2.16 `--no_linkview`

Discards the section-level view from the ELF image and retain only the segment-level view (load time view). Discarding the link-view section level eliminates:

- the section header table
- the section header string table
- the string table
- the symbol table
- all debug sections.

All that is left in the output is the program header table and the program segments. According to the ELF specification, these are all that a program loader can rely upon being present in an ELF file.

Note

This option can have unexpected effects if `--elf` is not specified on the command line. To get ELF format output use the options:

```
fromelf --no_linkview --elf image.axf -o image_nlk.axf
```

8.2.17 `--no_symbolversions`

Turns off the decoding of symbol version tables. See *Symbol versioning* on page 4-22 and the BPABI for more information.

8.2.18 `--output` *`output_file`*

Specifies the name of the output file, or the name of the output directory if multiple output files are created. Specifying the output file is optional with the `--text` output option but is mandatory with all other outputs.

See also

- *code_output_format* on page 8-3
- *--text* on page 8-12.

8.2.19 `--privacy`

Changes section names to a default value and also sets `--strip symbols`. In addition, all local symbols, except mapping and build attribute symbols, lose their names.

For example, code section names are changed to **`.text`**.

8.2.20 *`Project template options`*

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

`--[no_]project=[`*`filename`*`]`

Enables or disables the use of a project template file.

`--reinitialize_workdir`

Enables you to reinitialize the project template working directory.

`--workdir=`*`directory`*

Enables you to provide a working directory for a project template.

For more information on each of these options, see:

- *--[no_]project=filename* on page 2-74 in the *Compiler Reference Guide*
- *--reinitialize_workdir* on page 2-77 in the *Compiler Reference Guide*
- *--workdir=directory* on page 2-95 in the *Compiler Reference Guide*.

8.2.21 `--select` *`select_options`*

Use this option with either `--fieldoffsets` or `--text -a` to select only those fields that match the patterns in the option list for output or display.

Use special characters to select multiple fields:

- Join options in the list together with a comma (`,`) as in: `a*,b*,c*`.
- The wildcard character `*` can be used to match any name.
- The wildcard character ? can be used to match any single letter.

- Specify the fields to include by prefixing a `+` to the *`select_options`* string. This is the default.
- Specify the fields to exclude by prefixing a `~` to the *`select_options`* string.

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

8.2.22 `--show_cmdline`

Shows how `fromelf` has processed the command line. The commands are shown normalized, and the contents of any via files are expanded.

8.2.23 `--strip` *`option`* [, *option*]

Protects IP in images and objects that are delivered to third parties.

Where *option* is one of:

`all`	For object modules, this option removes all debug, comments, notes and symbols from the ELF file. For executables, this options works the same as `--nolinkview`. See *--no_linkview* on page 8-9 for more information.
`debug`	Removes all debug sections from the ELF file. It is synonymous with the existing `--no_debug` option.
`comment`	Removes the **`.comment`** section from the ELF file. It is synonymous with the existing `--no_comment_section` option.
`filesymbols`	The STT_FILE symbols are removed from the ELF file.
`notes`	Removes the **`.notes`** section from the ELF file.
`pathnames`	Removes the path information from all symbols with type STT_FILE. For example, an STT_FILE symbol with the name `C:\work\myobject.o` is renamed to `myobject.o`.
`symbols`	For objects, this option removes all local symbols from the ELF file that are not used as relocation targets. For executables, this option removes all static symbols. If any of these static symbols are used as a static relocation target, then these relocations are lost. In all cases, STT_FILE symbols are removed.

Note

Stripping the symbols, pathnames or filesymbols might make it harder to debug.

`--strip` has no effect when not outputting `--elf`.

8.2.24 `--text`

Prints image information in text format. You can decode an ELF image or ELF object file using this option. This is the default, that is, if no code output format is specified, `--text` is assumed.

If *`output_file`* is not specified with the `-o` option, the information is displayed on `stdout`.

Use one or more of the following options to specify what is displayed:

`-a`	Prints the global and static data addresses (including addresses for structure and union contents). This option can only be used on files containing debug information. Use the `--select` option to output a subset of the data addresses. If you want to view the data addresses of arrays, expanded both inside and outside structures, use the `--expandarrays` option with this text category.
`-c`	Disassembles code.
`-d`	Prints contents of the data sections.
`-e`	Decodes exception table information for objects. Use with `-c` when disassembling images.
`-g`	Prints debug information.
`-r`	Prints relocation information.
`-s`	Prints the symbol and versioning tables.
`-t`	Prints the string table(s).
`-v`	Prints detailed information on each segment and section header of the image.
`-y`	Prints dynamic segment contents.
`-z`	Prints the code and data sizes. See *Getting information about images* on page 3-33 for more information.

These options are only recognized when the `--text` output format is selected.

8.2.25 `--vsn`

Displays `fromelf` version information.

8.3 Examples of fromelf usage

This section contains examples of using `fromelf` to change image format or extract information from an ELF file.

Note

If you are using a wildcard character on Unix platforms, for example, `*`, ? or `~`, you must enclose the options in quotes to prevent the shell expanding the selection.

For example, enter `'*, ~*.*'` instead of `*, ~*.*`.

8.3.1 Producing a plain binary file

To convert an ELF file to a plain binary (`.bin`) file, use:

```
fromelf --bin -o outfile.bin infile.axf
```

8.3.2 Disassembly

To produce a listing to `stdout` that contains the disassembled version of an ELF file, use:

```
fromelf -c infile.axf
```

To produce a plain text output file that contains the disassembled version of an ELF file and the symbol table, use:

```
fromelf -c -s -o outfile.lst infile.axf
```

8.3.3 Listing field offsets as assembly language EQUs

To produce an output listing to `stdout` that contains all the field offsets from all structures in the file `inputfile.o`, use:

```
fromelf --fieldoffsets inputfile.o
```

To produce an output file listing to `outputfile.a` that contains all the field offsets from structures in the file `inputfile.o` that have a name starting with `p`, use:

```
fromelf --fieldoffsets --select p* -o outputfile.a inputfile.o
```

To produce an output listing to `outputfile.a` that contains all the field offsets from structures in the file `inputfile.o` with names of `tools` or `moretools`, use:

```
fromelf --fieldoffsets --select tools.*, moretools.* -o outputfile.a inputfile.o
```

To produce an output file listing to `outputfile.a` that contains all the field offsets of structure fields whose name starts with `number` and are within structure field `top` in structure `tools` in the file `inputfile.o`, use:

```
fromelf --fieldoffsets --select tools.top.number* -o outputfile.a inputfile.o
```

8.3.4 Listing addresses of static data

To list to `stdout` all the global and static data variables and all the structure field addresses, use:

```
fromelf --text -a --select * infile.axf
```

Selecting only structures

To produce a text file containing all of the structure addresses in `inputfile.axf` but none of the global or static data variable information, use:

```
fromelf --text -a --select *.* -o strucaddress.txt infile.axf
```

Selecting only nested structures

To produce a text file containing addresses of the nested structures only, use:

```
fromelf --text -a --select *.*.* -o strucaddress.txt infile.axf
```

Selecting only variables

To produce a text file containing all of the global or static data variable information in `inputfile.axf` but none of the structure addresses, use:

```
fromelf --text -a --select *, ~*.* -o strucaddress.txt infile.axf
```

8.3.5 Converting debug to no debug

To produce a new output file equivalent to using the `--no_debug` option from an ELF file originally produced with the `--debug` option, use:

```
fromelf --no_debug --elf -o outfile.axf infile.axf
```