RealView Compilation Tools
Linker and Utilities Guide

Copyright © 2002-2004 ARM Limited. All rights reserved.

Release Information

<table>
<thead>
<tr>
<th>Date</th>
<th>Issue</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>August 2002</td>
<td>A</td>
<td>Release 1.2</td>
</tr>
<tr>
<td>January 2003</td>
<td>B</td>
<td>Release 2.0</td>
</tr>
<tr>
<td>September 2003</td>
<td>C</td>
<td>Release 2.0.1 for RVDS v2.0</td>
</tr>
<tr>
<td>January 2004</td>
<td>D</td>
<td>Release 2.1 for RVDS v2.1</td>
</tr>
</tbody>
</table>

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.

Confidentiality Status

This document is Open Access. This document has no restriction on distribution.

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

Preface
About this book .................................................................................................................. vi
Feedback .......................................................................................................................... ix

Chapter 1 Introduction
1.1 About the linker and utilities ...................................................................................... 1-2

Chapter 2 The armlink Command Syntax
2.1 About armlink ........................................................................................................... 2-2
2.2 armlink command syntax ....................................................................................... 2-8

Chapter 3 Using the Basic Linker Functionality
3.1 Specifying the image structure .................................................................................. 3-2
3.2 Section placement ...................................................................................................... 3-8
3.3 Optimizations and modifications ............................................................................... 3-11
3.4 Using command-line options to create simple images .............................................. 3-19
3.5 Using command-line options to handle exceptions ............................................... 3-25

Chapter 4 Accessing Image Symbols
4.1 ARM/Thumb synonyms ......................................................................................... 4-2
4.2 Accessing linker-defined symbols ......................................................... 4-3
4.3 Accessing symbols in another image .................................................... 4-7
4.4 Hiding and renaming global symbols .................................................... 4-10
4.5 Using $Super$ and $Sub$ to override symbol definitions .................... 4-16

Chapter 5 Using Scatter-loading Description Files
5.1 About scatter-loading ............................................................................ 5-2
5.2 Formal syntax of the scatter-loading description file ......................... 5-8
5.3 Examples of specifying region and section addresses ....................... 5-23
5.4 Equivalent scatter-loading descriptions for simple images .............. 5-33

Chapter 6 Creating and Using Libraries
6.1 About libraries ..................................................................................... 6-2
6.2 Library searching, selection, and scanning ......................................... 6-3
6.3 The ARM librarian .............................................................................. 6-6

Chapter 7 Using fromELF
7.1 About fromELF .................................................................................... 7-2
7.2 fromelf command syntax ................................................................... 7-3
7.3 Examples of fromELF usage ................................................................. 7-10

Glossary
Preface

This preface introduces the RealView® Compilation Tools v2.1 Linker and Utilities Guide. It contains the following sections:

- About this book on page vi
- Feedback on page ix.
About this book

This book provides reference information for RealView Compilation Tools (RVCT) v2.1. It describes the command-line options to the linker and other ARM® tools in 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 v2.1 Essentials Guide.

Using this book

This book is organized into the following chapters and appendixes:

Chapter 1 Introduction
   Read this chapter for an introduction to the linker and related utilities in RVCT version 2.1.

Chapter 2 The armlink 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 details on creating simple images.

Chapter 4 Accessing Image Symbols
   Read this chapter for details on accessing symbols in images.

Chapter 5 Using Scatter-loading Description Files
   Read this chapter for details on using a scatter-loading file to place code and data in memory.

Chapter 6 Creating and Using Libraries
   Read this chapter for an explanation of the procedures involved in creating and accessing library objects.

Chapter 7 Using fromELF
   Read this chapter for a description of the fromELF utility and how you can use it to change image format.
Glossary  An alphabetically arranged glossary defines the special terms used in this book.

This book assumes that you have installed your ARM software 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\Specifications\... You might have to change this if you have installed your ARM software in a different location.

Typographical conventions

The following typographical conventions are used in this book:

**monospace** Denotes text that can be entered at the keyboard, such as commands, file and program names, and source code.

**monospace** Denotes a permitted abbreviation for a command or option. The underlined text can be entered instead of the full command or option name.

**monospace italic** Denotes arguments to commands and functions where the argument is to be replaced by a specific value.

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

**italic** Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.

**bold** Highlights interface elements, such as menu names. Also used for emphasis in descriptive lists, where appropriate, and for ARM processor signal names.

Further reading

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

ARM Limited periodically provides updates and corrections to its documentation. See http://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:

- *RealView Compilation Tools v2.1 Essentials Guide* (ARM DUI 0202)
- *RealView Compilation Tools v2.1 Developer Guide* (ARM DUI 0203)
- *RealView Compilation Tools v2.1 Assembler Guide* (ARM DUI 0204)

For general information on software interfaces and standards supported by ARM, see `install_directory\Documentation\Specifications\...`.

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

- *ARM Reference Peripheral Specification* (ARM DDI 0062)
- the ARM datasheet or technical reference manual for your hardware device.

**Other publications**

For a comprehensive introduction to ARM architecture see:


This book is not intended to be an introduction to the ARM assembly language, C, or C++ programming languages. Other books provide general information about programming.
Feedback

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

Feedback on the product

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.
Preface
Chapter 1
Introduction

This chapter introduces the ARM® linker, armlink, and the utility programs, armar and fromelf provided with RealView® Compilation Tools (RVCT) v2.1. It contains the following section:

- About the linker and utilities on page 1-2.
1.1 About the linker and utilities

RVCT consists of a suite of tools, together with supporting documentation and examples, that enable you to write applications for the ARM family of Reduced Instruction Set Computer (RISC) processors. You can use RVCT to build C, C++, and ARM assembly language programs.

The RVCT toolkit consists of the following major components:
- command-line development tools
- utilities
- supporting software.

This book describes the ARM linker, `armlink`, and the command-line utility tools provided with RVCT. See ARM publications on page viii for a list of the other books in the RVCT documentation suite that give information on the ARM assembler, compiler, and supporting software.

This section describes:
- The linker
- `fromELF` on page 1-5
- `armar` on page 1-5
- Deprecated features on page 1-6.

1.1.1 The linker

The ARM linker, `armlink`, combines the contents of one or more object files with selected parts of one or more object libraries to produce an Executable and Linking Format (ELF) executable image, or a partially linked ELF object.

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 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.

The linker can link ARM code and Thumb® code, and automatically generates interworking veneers to switch processor state when required. The linker also automatically generates long branch veneers, where required, to extend the range of branch instructions.
The linker supports command-line options that enable you to specify separate locations for 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 optimize ROM size. This means that the size of the data in ROM needed to initialize the RAM is reduced.

The linker can perform common section elimination and 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 call graph and list the stack usage over it
- use the call graph to identify functions with no stack-frame information and, where required, add the details manually
- control the contents of the symbol table in output images.

In addition to unused common sections, the linker can perform elimination of common groups or sections. The Comdat (ELF name for Common) group elimination process uses the same criteria as the common section removal mechanism.

The linker does not generate output formats other than ELF. To convert ELF images to other formats, such as plain binary for loading into ROM, use the fromELF utility. See fromELF on page 1-5.

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

Compatibility with legacy objects and libraries

The Application Binary Interface (ABI) in RVCT v2.0 was different to that in ARM Developer Suite™ (ADS) v1.2 and RVCT v1.2. Therefore, legacy ADS v1.2 and RVCT v1.2 objects and libraries are not directly compatible with RVCT v2.1 or v2.0. However, some restricted compatibility is provided with the --apcs /adsabi compiler option. For more details, see the chapter on using the ARM compiler in RealView Compilation Tools v2.1 Compiler and Libraries Guide.

When linking objects and libraries, you must be aware of the following:

- There is no compatibility between C++ objects in RVCT v2.1 (or v2.0) and legacy C++ objects.
Introduction

- You must link using the RVCT v2.1 or v2.0 linker, not the linker of older ARM tools. This is because older linkers cannot process objects produced by the RVCT v2.x compilers.

- You must link with the RVCT v2.1 or v2.0 C and C++ runtime libraries supplied by ARM, because these are compatible with objects compiled with the --apcs /adsabi compiler option.

If you have your own libraries, it is recommended that you rebuild these libraries with RVCT v2.1 to avoid any incompatibilities.

**Stack alignment**

The ABI for the ARM Architecture (base standard) [BSABI] requires that code maintains eight-byte stack alignment at its interfaces. The linker is very strict in checking alignment in object files and ensures that any code that requires eight-byte alignment of the stack is only called, directly or indirectly, by code that preserves eight-byte alignment of the stack.

Use assembly language directives to specify alignment:

- The REQUIRE8 directive specifies that the current file requires eight-byte alignment of the stack. It sets the REQ8 build attribute to inform the linker.

- The PRESERVE8 directive specifies that the current file preserves eight-byte alignment of the stack. It sets the PRES8 build attribute to inform the linker.

The RVCT v2.x compilers set PRESERVE8 automatically, and set REQUIRE8 when required.

The linker generates an error message if a stack alignment conflict is detected:

Error L6238E: object_name.o(section_name) contains invalid call from '-PRES8' function to 'REQ8' function_name.

A similar warning message is generated where the address of an external symbol is referenced:

Warning L6306W: '-PRES8' section object_name.o(section_name) should not use the address of 'REQ8' function_name.

This error typically occurs when:

- you attempt to link SDT or ADS objects with RVCT v2.x objects.

Legacy SDT and ADS objects that do not have these attributes are treated as not preserving eight-byte alignment. This is regardless of whether they do preserve eight-byte alignment or not.
assembly language code that does not preserve eight-byte stack alignment calls compiled C/C++ code that requires eight-byte stack alignment.

To correct this problem:

- Rebuild all your object files and libraries using RVCT v2.1, if possible. If you have assembly language files, the RVCT 2.1 assembler is able to mark them as PRES8 automatically (or not PRES8) as required (see the chapter describing the directives reference in *RealView Compilation Tools v2.1 Assembler Guide*).

- Where you cannot rebuild legacy code, use `fromelf -c` to disassemble the object code and then check whether it preserves eight-byte alignment.

  If the code does preserve eight-byte alignment, use the linker option `--diag_suppress 6238` to suppress the error message. However, if you suppress the error message for an object that does not preserve eight-byte alignment, your code might fail at runtime.

For full details on how to use PRESERVE8 and REQUIRE8 see the chapter describing the directives reference in *RealView Compilation Tools v2.1 Assembler Guide*.

### 1.1.2 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.

The utility can also produce textual information about the input file and disassemble code.

See Chapter 7 *Using fromELF* for more information.

### 1.1.3 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 6-6 for more information.
1.1.4 Deprecated features

Be aware of the following changes in RVCT v2.1:

- The linker accepts object files in ARM Object Format (AOF) format and libraries in ARM Library Format (ALF) format. However, these legacy Software Development Toolkit (SDT) formats are obsolete and will not be supported in the future.

- The use of single dashes for keywords, for example armlink -help, is deprecated and will not be supported in the future. Use double dashes when working with the compilation tools, for example armlink --help.

Controlling warnings of old syntax and deprecated options

By default, the compilation tools warn against the use of deprecated options (for example, -help instead of --help). You can change this behavior by setting the environment variable RVCT21_CLWARN to one of the following values:

- 0  Warn against old syntax and deprecated options.
- 1  Accept old syntax without a warning, but warn against deprecated options. This is the default.
- 2  Accept old syntax and deprecated options without a warning.
Chapter 2
The armlink Command Syntax

This chapter describes the full command syntax for the ARM® linker, armlink. It contains the following sections:

- *About armlink* on page 2-2
- *armlink command syntax* on page 2-8.
2.1 About armlink

The ARM linker, armlink, enables you to:

- link a collection of objects and libraries (in either ARM or Thumb® code) into an executable image
- partially link a collection of objects into an object that can be used as input to a subsequent link step
- specify where the code and data are to be located in memory
- produce debug and reference information about the linked files.

Objects consist of input sections that contain code, initialized data, or the locations of memory that must be set to zero. Input sections can be Read-Only (RO), Read-Write (RW), or Zero-Initialized (ZI). These attributes are used by armlink to group input sections into bigger building blocks called output sections, regions, and images.

Load regions typically exist in the system memory map at reset, for example in ROM, or after the image is loaded into the target by a debugger. As part of executing the image, some regions might have to be moved from their load addresses to their execution addresses. The memory map of an image, therefore, has the following distinct views:

**Load view** The memory view when the program and data are first loaded.

**Execution view** The memory view after code is moved to its normal execution location.

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.

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

This section describes:

- Input to armlink on page 2-3
- Output from armlink on page 2-3
- Summary of armlink options on page 2-4.
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. See ARM publications on page viii for more information.
- One or more libraries created by armar as described in Chapter 6 Creating and Using Libraries.
- A symbol definitions file.

Note

armlink accepts object files in ARM Object Format (AOF) format and libraries in ARM Library Format (ALF) format. However, these legacy Software Development Toolkit (SDT) formats are obsolete and will not be supported in the future.

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 partially-linked 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 the fromELF utility to convert an executable image in ELF executable format to other file formats. See Chapter 7 Using fromELF for more information.

Constructing an executable image

When you use armlink 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 and virtual functions
- 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

- relocates relocatable values
- generates an executable image.

**Constructing a partially-linked object**

When you use `armlink` to construct a partially-linked object, it:

- eliminates duplicate copies of debug sections
- minimizes the size of the symbol table
- leaves unresolved references unresolved
- 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 Summary of armlink options

This section gives a summary of `armlink` command-line options. The options are arranged into functional groups.

**Accessing help and information**

To get information on the available command-line options use:

```bash
--help
```

To get the tool version number use:

```bash
--vsn
```

**Specifying the output type and the output filename**

Name the output file using the following option:

```bash
--output
```

Use the following option to create a partially-linked object instead of an executable image:

```bash
--partial
```
To specify ELF output format use:

```
--elf
```

ELF is the default, and only, output format. Use the fromELF utility to convert ELF output to other formats. See Chapter 7 Using fromELF for more information.

**Specifying memory map information for the image**

Use the following options to specify simple memory maps:

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

Alternatively, for more complex images, use the options:

```
--pad
--scatter
```

If you use the `--scatter` option, you must provide a scatter-loading description file and a reimplementation of the `__user_initial_stackheap()` function.

Scatter-loading is described in Chapter 5 Using Scatter-loading Description Files.

The memory map options cannot be used for partial linking because they specify the memory map of an executable image. See RealView Compilation Tools v2.1 Developer Guide for more information.

**Controlling image contents**

These options control miscellaneous factors affecting the image contents:

```
--datacompressor list | --datacompressor id | --datacompressor off
--debug | --nodebug
--edit
--entry
--exceptions | --noexceptions
--exceptions_tables
--first
--keep
--last
--libpath
--locals | --nolocals
--nobestdebug | --bestdebug
```
The armlink Command Syntax

--remove | --noremove
--scanlib | --noscanlib
--userlibpath
--vfemode

Specifying Byte Addressing mode

These options control Byte Addressing mode:

--BE32
--BE8

Generating image-related information

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

--callgraph
--feedback
--info
--map
--symbols
--symdefs
--xref
--xreffdbg
--xreffrom
--xrefto

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

For --callgraph, the information is saved as an HTML file. An HTML file named output_name.htm is saved in the same directory as the generated image.

Controlling armlink diagnostics

These options control how armlink emits diagnostics:

--diag_style
--diag_suppress
--errors
--list
--mangled | --unmangled
--match crossmangled
--strict
--unresolved
--verbose
Use --verbose to output diagnostics to stderr.

**Using a via file**

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

```
--via
```

See the section on via files in *RealView Compilation Tools v2.1 Compiler and Libraries Guide* for more information.
The armlink Command Syntax

2.2 armlink command syntax

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

--- Note ---
For command-line arguments that use parentheses, you might have to escape the parentheses characters with a backslash (\) character on Sun Solaris or Red Hat Linux systems.

Specify command-line keywords using double dashes -- (for example, --partial). The single-dash command-line options used in previous versions of ADS and RVCT are still supported for backwards-compatibility but are deprecated.

The complete linker command syntax is:

armlink [-BE32|--BE8] [-callgraph] [--datacompressor off|list|id]
[--debug|--nodebug] [--diag_style arm|ide] [--diag_suppress taglist]
[--edit file-list] [--elf] [--entry location ] [--errors file]
[--exceptions_tables=action] [--feedback file] [--first section-id] [--help]
[--info topics] [--keep section-id] [--last section-id] [--libpath pathlist]
[--list file] [--locals|--nolocals] [--mangled|--unmangled] [--map]
[--match crossmangled] [--nobestdebug|--bestdebug]
[--remove (RO/Rw/ZI(DBG))|--noremove] [--ropi] [--rosplit] [--rw-base address]
[--strict] [--symbols] [--symdefs file] [--unresolved symbol]
[--xref] [--xrefdbg] [--xreffrom object(section)] [--xrefto object(section)]
[input-file-list]

The rest of this chapter describes these options in more detail:

- Accessing help and information on page 2-9
- Specifying an input file list on page 2-9
- Specifying the output type and the output filename on page 2-10
- Specifying memory map information for the image on page 2-10
- Controlling image contents on page 2-12
- Specifying Byte Addressing mode on page 2-21
- Generating image-related information on page 2-21
- Controlling armlink diagnostics on page 2-24
- Using a via file on page 2-26.
2.2.1 Accessing help and information

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

--help  This option prints a summary of some commonly used command-line options.

--vsn   This option displays the armlink version information and license details.

2.2.2 Specifying an input file list

This option defines the input files passed to the linker:

\textit{input-file-list}

This is a space-separated list of objects, libraries, or symdefs files.

The symdef file, can be included in the list to provide global symbol values for a previously generated image file. See Accessing symbols in another image on page 4-7 for more information.

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

- Specify particular members to be extracted from a library and added to the image as individual objects. For example, specify mstring.lib(strcmp.o) in the input file list.
- 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 mstring.lib in the input file list. The standard C or C++ libraries are added to this list implicitly by armlink when it scans the default library directories and selects the closest matching library variants available. Members from the libraries in this list are added to the image only when they resolve an unresolved non-weak reference. For more information see Library searching, selection, and scanning on page 6-3.

armlink 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 were objects. For example, if you are working on Windows, the following command unconditionally adds all a*.o objects and stdio.o from mylib:

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

3. The standard C or C++ libraries are added to the list of libraries that are later used to resolve any remaining non-weak unresolved references.
2.2.3 Specifying the output type and the output filename

Name the output file using the following options:

--output file

This option 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, armlink 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  This option creates a partially-linked object instead of an executable image.

--elf  This option generates the image in ELF format. This is the only output format supported by armlink. This is the default.

2.2.4 Specifying memory map information for the image

Use the following options to specify simple memory maps:

--ro-base address

This option 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, nor a scatter file specified, the default RO base address is 0x8000.

--rw-base address

This option sets the execution addresses of the region containing the RW output section at address. address must be word-aligned.

--ropi  This option makes the load and execution region containing the RO output section position-independent. If this option is not used the region is marked as absolute. Usually each read-only input section must be read-only position-independent. If this option is selected, armlink:

• checks that relocations between sections are valid
• ensures that any code generated by armlink itself, such as interworking veneers, is read-only position-independent.
--rwpi  This option makes the load and execution region containing the RW and ZI output section position-independent. If this option is not used the region is marked as absolute. This option requires a value for --rw-base. If --rw-base is not specified, --rw-base 0 is assumed. Usually each writable input section must be read-write position-independent.

If this option is selected, armlink:

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

--reloc  This option makes relocatable ELF images (see the ARM ELF specification for more information).

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 run time.

If the image is loaded at its link-time address, the relocatable image produced by armlink 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 output regions on page 3-20 for details).

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

- One containing the RO output section. The default load address is 0x8000, but a different address can be specified with the --ro-base option.
- One 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.

--rosplit  This option splits the default load region into two RO output sections, one for code and one for data.
--pad num

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

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.

--scatter file

This option 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 is mutually exclusive with the use of any of the memory map options --reloc, --ro-base, --rw-base, --ropi, --rwpi, --rosplit, or --split.

Note

You must reimplement the stack and heap initialization function __user_initial_stackheap() if you use this option.

2.2.5 Controlling image contents

These options control miscellaneous factors affecting the image contents:

--datacompressor list|id

These options 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.
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.

If you do not specify a data compression algorithm, armlink chooses the most appropriate one for you. For more information see RW data compression on page 3-14.

--datacompressor off

This option turns off RW data compression. RW data compression is enabled by default to optimize ROM size.

--debug

This option includes debug information in the output file. The debug information includes debug input sections and the symbol and string table. This is the default.

--nodebug

This option turns off the inclusion of debug information in the output file. The ELF image is smaller, but you cannot debug it at the source level. armlink 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 and 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, armlink 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 --nodebug if a fromelf --fieldoffsets step is required. If your image is produced without debug information:

• fromELF cannot translate the image into other file formats
• fromELF cannot produce a meaningful disassembly listing.

--nobestdebug

This option selects sections without reference to debug information, that is it chooses the smallest sections. This is the default.

Use the option --bestdebug to select the Comdat group or common section with the best debug view.
--- Note ---

--nobestdebug is the default, so that the code and data of the final image is the same regardless of whether you generate debug tables during compilation with -g. For details, see generating debug information in the chapter describing how to use the ARM compiler in RealView Compilation Tools v2.1 Compiler and Libraries Guide.

--exceptions

This option enables the final image to contain exception tables. This is the default.

See Using command-line options to handle exceptions on page 3-25 for more information.

--noexceptions

This option forces the linker to generate an error message if any exceptions sections are present in the image after unused sections have been eliminated. If required, you can downgrade this error to a warning and then suppress the warning. Use this option to ensure that your code is exceptions free.

--exceptions_tables=action

Use this option to specify how exception tables are generated for legacy objects. Replace action with one of:

nocreate The linker does not create exception tables for legacy objects. 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 exceptions on page 3-25 for more information.

--edit file-list

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

- Hide global symbols. Use this option to hide specific global symbols in object files. The hidden symbols are not publicly visible.
The armlink Command Syntax

- Rename global symbols. Use this option to resolve symbol naming conflicts.

See **Hiding and renaming global symbols** on page 4-10 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**

This option specifies the unique initial entry point of the image. The image can contain multiple entry points, but the initial entry point specified using this command 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. ARM debuggers use this entry address to initialize the 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**
  - This option specifies an image entry point as the address of **symbol**, for example:
    ```
    --entry reset_handler
    ```
- **offset+object**(section)
  - This option 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**.
The armlink Command Syntax

--first section-id
This option places the selected input section first in its execution region. This can, for example, place the section containing the vector table first in the image. 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. armlink warns that --first and --last are ignored if a scatter-loading description file is used.

Using --first cannot override the basic attribute sorting order for output sections in regions that places RO first, RW second, and ZI last. If the region has an RO section, an RW or a ZI section cannot be placed first. If the region has an RO or RW section, a ZI section cannot be placed first.

Two different sections cannot both be placed first in the same execution region, so only one instance of this option is permitted.

--keep section-id
Specifies input sections that are not to be removed by unused section elimination. See Specifying an image memory map on page 3-5. All forms of section-id argument to --keep can contain the * and ? wildcards. Replace section-id with one of the following:

symbol This option specifies that the input section defining symbol is to be retained during unused section elimination. If multiple definitions of symbol exist, then all input sections that define symbol are treated similarly, for example:
--keep int_handler
To keep all sections that define a symbol ending in `_handler`, use:

```bash
--keep *_handler
```

There can be multiple occurrences of this command on the command line.

**object** *(section)*

This option specifies that `section` from `object` is to be retained during unused section elimination. The input section and object names are matched without case-sensitivity. For example, to keep the `vect` section from the `vectors.o` object use:

```bash
--keep vectors.o(vect)
```

To keep all sections from the `vectors.o` object where the first three characters of the name of the section is `vec`, use:

```bash
--keep vectors.o(vec*)
```

There can be multiple occurrences of this command on the command line.

**object**

This option specifies that the single input section from `object` is to be retained during unused section elimination. The object name is matched without case-sensitivity. If you use this short form and there is more than one input section in `object`, `armlink` generates an error message. For example:

```bash
--keep dspdata.o
```

To keep the single input section from each of the objects that has a name starting with `dsp`, use:

```bash
--keep dsp*.o
```

There can be multiple occurrences of this option on the command line.

**--last section-id**

This option places the selected input section last in its execution region. For example, this can force an input section that contains a checksum to be placed last in the RW section. 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 more than one section cannot be placed last. For example:
  ```bash
  --last checksum
  ```
The armlink Command Syntax

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.

---

Using --last cannot override the basic attribute sorting order for output sections in regions that places RO first, RW second, and ZI last. If the region has a ZI section, an RW section cannot be placed last. If the region has an RW or ZI section, an RO section cannot be placed last.

Two different sections cannot both be placed last in the same execution region, so only one instance of this option is permitted.

--libpath pathlist

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

  The default path for the parent directory containing the ARM libraries is specified by the RVCT21LIB environment variable. Any paths specified here override the path specified by RVCT21LIB.

  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 6-3 for more information on including libraries.

--locals

- This option instructs the linker to add local symbols to the output symbol table when producing an executable image. This is the default.
The armlink Command Syntax

--nolocals This option instructs the linker not to add local symbols to the output symbol table when producing an executable image. This is a useful optimization if you want to reduce the size of the output symbol table.

--remove
--remove (RO/RW/ZI/DBG)

This option performs unused section elimination on the input sections to remove unused 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. See also Unused section elimination on page 3-12.

Note
You must take care to avoid reset code or exception handlers accidentally being removed when using --remove. Use the --keep option to identify exception handlers or use the ENTRY directive to label them as entry points.

You can use section attribute qualifiers for more precise control of the unused section elimination process. If a qualifier is used, it can be one or more of the following:

RO        Remove all unused sections of type RO.
RW        Remove all unused sections of type RW.
ZI        Remove all unused sections of type ZI.
DBG       Remove all unused sections of type DEBUG.

The qualifiers can appear in any case and order, but must be enclosed in parentheses (), and must be separated by a slash /.

The default is --remove (RO/RW/ZI/DBG).

If no section attribute qualifiers are specified, all unused sections are eliminated. --remove is equivalent to --remove (RO/RW/ZI/DBG).

--noremove This option does not perform unused section elimination on the input sections. This retains all input sections in the final image even if they are unused.

--scanlib This option enables scanning of default libraries (the standard ARM C and C++ libraries) to resolve references. This is the default.

--noscanlib This option prevents the scanning of default libraries in a link step.

--userlibpath pathlist
This option specifies a list of paths that are used to search for user libraries.
The armlink Command Syntax

`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 6-3 for more information on including user libraries.

`--vfemode=mode`

This option specifies how unused virtual functions, and Runtime Type Information (RTTI) objects, are eliminated in the linker. Depending on the mode specified, the linker uses extra information about virtual functions and RTTI objects supplied by the compiler to analyze more accurately how such functions are used. Unused functions are then eliminated.

Replace `mode` with one of:

- **on**
  - Makes the linker Virtual Function Elimination (VFE) aware.
  - In this mode, `armlink` chooses `force` or `off` mode based on the content of object files. This is the default.
  - This is the same as specifying no VFE option on the command line.

- **off**
  - Use this option 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**
  - Makes the linker VFE aware and forces the VFE algorithm to be applied. If some of the object files do not contain VFE information, the linker continues with the elimination but displays a warning to alert you to possible errors.

- **force_no_rtti**
  - When operating in default mode (using `--vfemode=on`), the linker might remove both unused virtual functions and RTTI objects. Use this option to make the linker VFE aware and force the removal of all RTTI objects whilst retaining all the virtual functions.

See also Unused virtual function elimination on page 3-12 for more details.
2.2.6 Specifying Byte Addressing mode

These options control Byte Addressing mode:

--BE32
This option specifies standard big-endian mode, that is, identical to big-endian images prior to ARMv6. This produces big-endian code and data. This is the default Byte Addressing mode for all pre-ARMv6 big-endian images.

--BE8
This option specifies ARMv6 Byte Invariant Addressing mode. This means that the linker reverses the endianness of the instructions to give little-endian code and big-endian data. This is the default Byte Addressing mode for ARMv6 big-endian images.

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

2.2.7 Generating image-related information

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

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

Note
Any assembler files must contain PROC/ENDP and FRAME PUSH/POP directives if the linker is to calculate the function stack sizes.

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
The 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.

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.

Use frame directives 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. For full details on how stack usage is determined, see the chapter describing the directives reference in RealView Compilation Tools v2.1 Assembler Guide.

--feedback file

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

When you next compile the file, use the compiler option --feedback file to specify the feedback file to use. Unused functions are placed in their own sections for future elimination by the linker. For full details on how to use this file see the chapter describing how to use the ARM compiler in RealView Compilation Tools v2.1 Compiler and Libraries Guide.

--info topics

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

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

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

sizes Sizes gives a list of 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.

totals Totals gives totals of the Code and Data (RO Data, RW Data, ZI Data, and Debug Data) sizes for input objects and libraries.
The armlink Command Syntax

veneers  Gives details of armlink-generated veneers. For more information on veneers see Veneer generation on page 3-17.

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

exceptions  Gives details of exception table generation and optimization.

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

Where 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

--map  This option creates an image map. The image map contains the address and the size of each load region, execution region, and input section in the image, including linker-generated input sections.

--symbols  This option lists each local and global symbol used in the link step, and its value. This includes linker-generated symbols.

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

By default, all global symbols are written to the symdefs file. If file already exists, the linker restricts its output to the symbols listed in the existing symdefs 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 4-7 for more information.

--xref
This option lists all cross-references between input sections.

--xrefdbg
This option lists all cross-references between input debug sections.

--xreffrom object(section)
This option lists cross-references from input section in object to other input sections. This is a useful subset of the listing produced by using --xref if you are interested in references from a specific input section. You can have multiple occurrences of this option to list references from more than one input section.

--xrefto object(section)
This option lists cross-references to input section in object from other input sections. This is a useful subset of the listing produced by using --xref if you are interested in references to a specific input section. You can have multiple occurrences of this option in order to list references to more than one input section.

2.2.8 Controlling armlink diagnostics

These options control how armlink emits diagnostics:

--diag_style arm|ide
This option specifies that all warning and error messages are output in a format that is more compatible with IDEs, such as Microsoft Visual Studio.

The default is arm, for example:
Error: L6223E: Ambiguous selectors for foo.o ....
Finished: 0 information, 1 warning and 1 error messages.

Specifying --diag_style ide results in:
Armlink: Error: L6223E: Ambiguous selectors for foo.o ....
Armlink: Finished: 0 information, 1 warning and 1 error messages.
--diag_suppress taglist

This option disables all diagnostic messages that have the specified tag(s).

This option requires a taglist, identifying the number of the message to be suppressed, and more than one tag can be specified. For example, to suppress the warning messages that have numbers L6314W and L6305W, use the following command:

armlink --diag_suppress 6314,6305 ...

--errors file

Redirects the diagnostics from the standard error stream to file.

--list file

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

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.

--match crossmangled

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

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

Libraries and matching 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

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

--mangled
This option instructs the linker to display mangled C++ symbol names in diagnostic messages, and in listings produced by the --xref, --xreffrom, --xrefto, and --symbols options. If this option is selected, the linker does not unmangle C++ symbol names. The symbol names are displayed as they appear in the object symbol tables.

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

If this option is selected, the linker unmangles C++ symbol names so that they are displayed as they appear in your source code. This is the default.

--unresolved symbol
This option 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.

--verbose
This option prints detailed information about the link operation, including the objects that are included and the libraries from which they are taken. As this output is typically quite long, you may wish to use this command with the --errors file command to redirect the information to file.

Use --verbose to output diagnostics to stderr.

2.2.9 Using a via file

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

--via file
This option reads a further list of input filenames and linker options from file.

You can enter multiple --via options on the armlink command line. The --via options can also be included within a via file.

See RealView Compilation Tools v2.1 Compiler and Libraries Guide for more information on writing via files.
Chapter 3
Using the Basic Linker Functionality

This chapter describes the basic functionality available in the ARM® linker, arm{link}.

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.

This chapter 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 3-19
• Using command-line options to handle exceptions on page 3-25.
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.

This section describes:
- Building blocks for objects and images
- Load view and execution view of an image on page 3-4
- Specifying an image memory map on page 3-5
- Image entry points on page 3-6.

3.1.1 Building blocks for objects and images

An image, as stored in 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.
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. Input sections can have the attributes RO, RW, or ZI. These three 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 to 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** This 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** This 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](image)

Table 3-1 compares the load and execution views.

<table>
<thead>
<tr>
<th>Load</th>
<th>Description</th>
<th>Execution</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load address</td>
<td>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.</td>
<td>Execution address</td>
<td>The address where a section or region is located while the image containing it is being executed.</td>
</tr>
<tr>
<td>Load region</td>
<td>A region in the load address space.</td>
<td>Execution region</td>
<td>A region in the execution address space</td>
</tr>
</tbody>
</table>

Table 3-1 Comparing load and execution views
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, arm-link 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 arm-link:

**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`
- `--reloc`
- `--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 3-19.

**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 exception vectors.

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 2-8 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 the following warning:
L6305W: Image does not have an entry point. (Not specified or not set due to multiple choices)

**Specifying multiple entry points**

An embedded image can have multiple entry points. For example, an embedded image typically has entry points that are used by the Reset, IRQ, FIQ, SVC, UNDEF, and ABORT exceptions to transfer control when the image is running.

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 *RealView Compilation Tools v2.1 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* on page 3-6 for more information.
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.

When generating an image, the linker sorts the input sections in the following order:

1. By attribute.
2. By input section name.
3. By their positions in the input list, except where overridden by a --first or --last option. This is described in Using FIRST and LAST to place sections on page 3-9.

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.

This section describes:
- Ordering input sections by attribute
- Using FIRST and LAST to place sections on page 3-9
- Aligning sections on page 3-9
- Ordering execution regions containing Thumb code on page 3-10.

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. other initialized data
5. zero-initialized (uninitialized) 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 pseudo-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. This means that an input section can be first (or last) in the execution region if the output section it is in is the first (or last) output section in the region. For example, in an execution region containing RO input sections, the `FIRST` input section must be an RO input section. Similarly, if the region contains any ZI input sections, the `LAST` input section must be a ZI input section.

Within each output section, input sections are sorted alphabetically according to their names, and then by their positions in the input order.

### 3.2.3 Aligning sections

When input sections have been ordered and before the base address is fixed, `arm\1ink` inserts padding, if required, to force each input section to start at an address that is a multiple of the input section alignment.

It is possible to use `ALIGN` to expand the alignment (changing something that is normally four-byte aligned to be eight-byte aligned), but you cannot reduce the natural alignment (forcing two-byte alignment on something that is normally four-byte aligned).

Shown in Example 3-1 on page 3-10, the input section alignment can be specified in assembly code using:

```
ALIGN Use this attribute to the `AREA` directive from assembly language. The input section address will be a multiple of 2<sup>(value in align attribute)</sup>.
```
Example 3-1 Using the ALIGN attribute in assembly code

AREA LDR_LABEL, CODE, READONLY, ALIGN=3 ; align on eight-byte boundary

See the description of ALIGN in the directives reference in RealView Compilation Tools v2.1 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 amount of veneers generated.

However, where an execution region contains Thumb code that exceeds 4MB, armlink cannot guarantee that the section sorting order conforms to the rules described in Section placement on page 3-8.
3.3 Optimizations and modifications

\texttt{arm\_link} performs optimizations and modifications in order to:
\begin{itemize}
  \item remove duplicate sections, unused virtual functions, and common sections or groups
  \item compress data
  \item enable interworking between ARM and Thumb code
  \item enable long branches.
\end{itemize}

This section describes:
\begin{itemize}
  \item \textit{Common debug section elimination}
  \item \textit{Common group or section elimination}
  \item \textit{Unused section elimination} on page 3-12
  \item \textit{Unused virtual function elimination} on page 3-12
  \item \textit{RW data compression} on page 3-14
  \item \textit{Veneer generation} on page 3-17
  \item \textit{Reuse of veneers with overlay execution regions} on page 3-18.
\end{itemize}

3.3.1 Common debug section elimination

The compiler and assembler generate one set of debug sections for each source file that contributes to a compilation unit. \texttt{arm\_link} 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.

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 a 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, \texttt{arm\_link} retains the best available variant of each common code section, or group, based on the attributes of the input objects. \texttt{arm\_link} discards the rest.
This optimization is controlled by linker options:

- Use the \texttt{--bestdebug} option to use the largest Comdat group (likely to give the best debug view).
- Use the \texttt{--nobestdebug} option to use the smallest Comdat group (likely to give the smallest code size). This is the default. Because \texttt{--nobestdebug} is the default, the final image is the same regardless of whether you generate debug tables during compilation with \texttt{-g}.

For details, see generating debug information in the chapter describing how to use the ARM compiler in \textit{RealView Compilation Tools v2.1 Compiler and Libraries Guide}.

\subsection*{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 \texttt{--(no)remove} and \texttt{--first}, \texttt{--last}, and \texttt{--keep} linker options. Use the \texttt{--info unused} linker option to instruct the linker to generate a list of the unused sections that have been eliminated.

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 was specified as the first or last input section by the \texttt{--first} or \texttt{--last} option (or a scatter-loading equivalent)
- if it has been marked as unremovable by the \texttt{--keep} option.

\textbf{Note} Unused section elimination is a property of all groups, not just common groups.

\subsection*{3.3.4 Unused virtual function elimination}

Unused virtual function elimination (VFE) is a refinement of unused section elimination to reduce ROM size in images generated from C++ code. This optimization can also be used to eliminate both unused virtual functions and RTTI objects from your code.
As described in Controlling image contents on page 2-12, VFE awareness is controlled by linker option --vfemode=mode.

--- Note ---
If you do not specify a VFE option on the command line, default mode is assumed, that is --vfemode=on.

---

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

### 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.

To stop the compiler producing VFE information, use the compiler option `--no_vfe`. However, the recommended way to switch off VFE is with the linker option `--vfemode=off`.

---

3.3.5 **RW data compression**

Most RW data areas contain over 75% zeroes making them highly compressible. RW data compression is enabled by default to optimize ROM size.

The ARM libraries contain some decompression algorithms and the linker chooses the optimal one to decompress the data areas in the final image. However, you can override the algorithm chosen by the linker.

This section describes data compression in more detail:
- **Choosing a compressor** on page 3-15
- **How is compression applied?** on page 3-15
- **Working with RW data compression** on page 3-16.
Choosing a compressor

armlink gathers information about the content of data sections before choosing the most appropriate compression algorithm to generate the smallest code size. Where 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:

\[
\text{compressed data size + size of decompressor} < \text{uncompressed data size}
\]

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

You can override the compression used by the linker by:

- using the --datacompressor off option to specify that no compression is applied
- specifying your own compressor.

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

<table>
<thead>
<tr>
<th>Num</th>
<th>Compression algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Run-length encoding</td>
</tr>
<tr>
<td>1</td>
<td>Run-length encoding, with LZ77 on small-repeats</td>
</tr>
<tr>
<td>2</td>
<td>Complex LZ77 compression</td>
</tr>
</tbody>
</table>

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. Lempel-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 non-zero bytes.
- Compressor 1 performs well on data where the non-zero 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. Where the image contains any Thumb code, Thumb decompressors are used. Where there is no clear preference, all compressors are tested to produce the best overall size (see Choosing a compressor on page 3-15).

--- 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 future releases.

---

**Working with RW data compression**

When working with RW data compression:

- In previous releases of RVCT, the only sections that had to be root were `__main` and the region tables. With the implementation of RW data compression there are more files that must be placed in a root region.
  
  The linker can detect these sections automatically.

- Use the linker option `--map` to see where compression has been applied to regions in your code.

- Where you are using an ARM processor with on-chip cache, enable the cache after decompression to avoid code coherency problems.
  
  See the chapter describing how to develop embedded software in *RealView Compilation Tools v2.1 Developer Guide* for details.

If you are using a scatter-loading description file:

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

  ```
  ER_ROOT
  {
    __main.o(*)
    # (Region$$Table)
    __scatter*.o(*)
    __dc*.o(*)
  }
  
  Or, alternatively, specify that a section must be root using `InRoot$$Sections`, for example:
ER_ROOT
{
    *(InRoot$$Sections)
}

See Assigning sections to a root region on page 5-28 for more details. Also, see the chapter on embedded software development in the RealView Compilation Tools v2.1 Developer Guide.

• Specify that a load or execution region should not be compressed by adding the NOCOMPRESS attribute (see Formal syntax of the scatter-loading description file on page 5-8 for details).

3.3.6 Veneer generation

armlink must generate veneers when:
• a branch involves change of state between ARM and Thumb state
• a branch involves a destination beyond the branching range of the current state.

A veneer can extend the range of branch and change processor state. armlink combines long branch capability into the state change capability. All interworking veneers are also long branch veneers.

The following veneer types handle different branching requirements. The linker generates position-independent versions of the veneers if required:

ARM to ARM Long branch capability.
ARM to Thumb Long branch and interworking capability.
Thumb to ARM Long branch and interworking capability.
Thumb to Thumb Long branch capability.

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 if the veneer can be reached by both sections.

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.
3.3.7 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 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.
3.4 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, the RW, and the 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.

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.

See *Type 1: one load region and contiguous output regions* on page 3-20 for more details.

**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 output regions* on page 3-21 for more details.

**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 `--ro-split` option to split the default load region into two RO output sections, one for code and one for data.

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

See *Type 3: two load regions and non-contiguous output regions* on page 3-23 for more details.

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 5-33 for more information on how to do this.

This section describes:

- *Type 1: one load region and contiguous output regions* on page 3-20
- *Type 2: one load region and non-contiguous output regions* on page 3-21
3.4.1 Type 1: one load region and contiguous output regions

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

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 on page 3-20.

3.4.2 Type 2: one load region and non-contiguous output 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](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, in ROM for example. 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 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 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.
3.4.3 Type 3: two load regions and non-contiguous output regions

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

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
This option 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 separately placed using --ro-base and --rw-base.

--ro-base address
This option 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 --ro-base option is not used to specify the address, the default value of 0x8000 is used by armlink.

--rw-base address
This option 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 (that is, it is a root region).
3.5 Using command-line options to handle exceptions

By default, the linker accepts exception tables in the linked image. Exception tables are discarded silently if no code throws an exception. If the option --exceptions is specified, the image can contain exception tables. However, if the option --noexceptions is specified, the linker generates an error if any exceptions sections are present after unused sections have been eliminated. If required, you can downgrade this error to a warning and then suppress the warning. Use the --noexceptions option to ensure that your code is exceptions free.

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.

Use the linker option --exceptions_tables=unwind 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. Where 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

The linker is unable to generate the cleanup code necessary for automatic variables in legacy (ADS v1.2 and RVCT v1.2) C++ code.
Chapter 4
Accessing Image Symbols

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

- ARM/Thumb synonyms on page 4-2
- Accessing linker-defined symbols on page 4-3
- Accessing symbols in another image on page 4-7
- Hiding and renaming global symbols on page 4-10
- Using $Super$$ and $Sub$$ to override symbol definitions on page 4-16.
4.1 ARM/Thumb synonyms

`armlink` 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 `armlink`, and `armlink` produces 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 Limited. The symbols are used to specify region base addresses, output section base addresses, and input section base addresses and their limits.

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 4-6 for details.

Note
Linker-defined symbols are defined by armlink only when your code references them.

This section describes:
- Region-related symbols
- Section-related symbols on page 4-5
- Importing linker-defined symbols on page 4-6.

4.2.1 Region-related symbols

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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load$$region_name$$Base</td>
<td>Load address of the region</td>
</tr>
<tr>
<td>Image$$region_name$$Base</td>
<td>Execution address of the region</td>
</tr>
<tr>
<td>Image$$region_name$$Length</td>
<td>Execution region length in bytes (multiple of 4)</td>
</tr>
<tr>
<td>Image$$region_name$$Limit</td>
<td>Address of the byte beyond the end of the execution region</td>
</tr>
<tr>
<td>Image$$region_name$$ZI$$Base</td>
<td>Execution address of the ZI output section in this region</td>
</tr>
<tr>
<td>Image$$region_name$$ZI$$Length</td>
<td>Length of the ZI output section in bytes (multiple of 4)</td>
</tr>
<tr>
<td>Image$$region_name$$ZI$$Limit</td>
<td>Address of the byte beyond the end of the ZI output section in the execution region</td>
</tr>
</tbody>
</table>
If you are using scatter-loading, the description file names all the execution regions in the image, and provides their load and execution addresses (see Chapter 5 Using Scatter-loading Description Files for details).

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

- **ER_R0**, for read-only regions
- **ER_RW**, for read-write regions
- **ER_ZI**, for zero-initialized regions.

For every execution region containing a ZI output section, armlink generates additional symbols containing **$ZI$$**.

---

**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.

### 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 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 the description of `__user_initial_stackheap()` in the chapter describing the C and C++ libraries in *RealView Compilation Tools v2.1 Compiler and Libraries Guide* for more information. See also the description of the example `retarget.c` in the chapter describing handling processor exceptions in *RealView Compilation Tools v2.1 Developer Guide* for an example of how to do this in C.

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

```assembly
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, *armlink* generates the input symbols shown in Table 4-2.

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.

Table 4-2 Section-related linker symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Section type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image$$RO$$Base</td>
<td>Output</td>
<td>Address of the start of the RO output section.</td>
</tr>
<tr>
<td>Image$$RO$$Limit</td>
<td>Output</td>
<td>Address of the first byte beyond the end of the RO output section.</td>
</tr>
<tr>
<td>Image$$RW$$Base</td>
<td>Output</td>
<td>Address of the start of the RW output section.</td>
</tr>
<tr>
<td>Image$$RW$$Limit</td>
<td>Output</td>
<td>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.)</td>
</tr>
<tr>
<td>Image$$ZI$$Base</td>
<td>Output</td>
<td>Address of the start of the ZI output section.</td>
</tr>
<tr>
<td>Image$$ZI$$Limit</td>
<td>Output</td>
<td>Address of the byte beyond the end of the ZI output section.</td>
</tr>
<tr>
<td>SectionName$$Base</td>
<td>Input</td>
<td>Address of the start of the consolidated section called SectionName.</td>
</tr>
<tr>
<td>SectionName$$Limit</td>
<td>Input</td>
<td>Address of the byte beyond the end of the consolidated section called SectionName.</td>
</tr>
</tbody>
</table>

**Note**

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, *armlink* diagnoses an error because the use of the base and limit symbols over non-contiguous memory usually produces unpredictable and undesirable effects.

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 \texttt{\_\_user\_initial\_stackheap()} uses the value in \texttt{Image$$ZI$$Limit}. Therefore, if you are using a scatter-loading description file you must reimplement \texttt{\_\_user\_initial\_stackheap()} to set the heap and stack boundaries. For more information, see \textit{Region-related symbols} on page 4-3 and the section on library memory models in \textit{RealView Compilation Tools v2.1 Compiler and Libraries Guide}.

### 4.2.3 Importing linker-defined symbols

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

- \texttt{extern unsigned int symbol\_name;}
- \texttt{extern void *symbol\_name;}

If you declare a symbol as an \texttt{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**

\begin{verbatim}
extern unsigned int Image$$ZI$$Limit
config.heap_base = (unsigned int) &Image$$ZI$$Limit
\end{verbatim}
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.

This section describes:
- Creating a symdefs file
- Reading a symdefs file on page 4-8
- Symdefs file format on page 4-8.

4.3.1 Creating a symdefs file

Use the arm\link option --symdefs filename to produce a symdefs file.

arm\link 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 arm\link 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, arm\link 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. arm\link 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 relink 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 would add any object file. armlink 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. armlink 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 out of a symdefs file are treated in exactly the same way as symbols extracted from an object symbol table. The same restrictions regarding multiple symbol definitions and ARM/Thumb synonyms apply.

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, RVCT2.1 [Build 216]: Last Updated: Tues Jan 06 11:10:30 2004
;value type name, this is an added comment
0x00008000 A __main
0x00008004 A __scatterload
0x000080e0 T main
0x0000814c T _main_arg
0x0000814e T __argv_alloc
0x00008198 T __rt_get_argv
...
# This is also a comment, blank lines are ignored
...
0x0000a4fc D __stdin
0x0000a540 D __stdout
0x0000a584 D __stderr
0xffffffff N __SIG_IGN
0xffffffff N __SIG_ERR
0xffffffff N __SIG_DFL
```
Identifying string

If the first 11 characters in the text file are #<SYMDEFS>#, armlink 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 armlink when the file is produced and must not be manually deleted.
- Any line where the first non-whitespace character is 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:

<table>
<thead>
<tr>
<th>Symbol value</th>
<th>armlink writes the absolute address of the symbol in fixed hexadecimal format, for example 0x00000000. If you edit the file, you can use either hexadecimal or decimal formats for the address value.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type flag</td>
<td>The single letter for type is:</td>
</tr>
<tr>
<td>A</td>
<td>ARM code</td>
</tr>
<tr>
<td>T</td>
<td>Thumb code</td>
</tr>
<tr>
<td>D</td>
<td>Data</td>
</tr>
<tr>
<td>N</td>
<td>Number.</td>
</tr>
<tr>
<td>Symbol name</td>
<td>The name of the symbol.</td>
</tr>
</tbody>
</table>
4.4 Hiding and renaming global symbols

This section describes how to use a steering file to hide or rename global 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 2-8). 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.

This section describes:

- Steering file format
- Steering file commands on page 4-11.

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 nonblank, non-comment line is either a command, or part of a command that is split over consecutive nonblank lines.
- Command lines that end with a comma (,) as the last non-whitespace character are continued on the next nonblank line.

Each command 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 or STT_FILE, are not affected.
4.4.2 Steering file commands

Steering file commands enable you to rename, resolve, hide, and show symbols in the symbol table. The following commands are supported:

- `RENAME`
- `RESOLVE` on page 4-12
- `HIDE` on page 4-14
- `SHOW` on page 4-15.

**RENAME**

The `RENAME` command renames defined and undefined global symbol names.

**Syntax**

```
RENAME pattern AS replacement_pattern [ ,pattern AS replacement_pattern] *
```

where:

- `pattern` 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` 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`.

`arm` 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 A string, optionally including wildcard characters, that must be matched to a defined global symbol.

defined_pattern 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 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 the files file1.c and file2.c, shown in Example 4-4 on page 4-13. 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

file1.c

extern int foo;
void MP3_Init(void);
void MP3_Play(void);

void main(void)
{
    int x = foo + 1;
    MP3_Init();
    MP3_Play();
}

file2.c

int foobar;

void MyMP3_Init()
{
}

void MyMP3_Play()
{
}
HIDE

The HIDE command makes defined global symbols in the symbol table anonymous.

Syntax

HIDE pattern [,pattern]*

where:

pattern 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_.

This example can be linked with other objects, provided they do not contain references to the hidden symbols. Once symbols are hidden in the output object, SHOW commands in subsequent link steps will have no effect on them.

The hidden references are removed from the output symbol table.

Example 4-5

---

steer.txt

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

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

---
SHOW

The SHOW command makes global symbols visible that were previously hidden with the HIDE command. This command is useful if you want to unhide a specific symbol that has been hidden using a HIDE command with a wildcard.

Syntax

SHOW pattern [,pattern]*

where:

pattern 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 4-14 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 will be 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 details, see ARM ELF specification in install_directory\Documentation\Specifications\...
Chapter 5
Using Scatter-loading Description Files

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

• About scatter-loading on page 5-2
• Formal syntax of the scatter-loading description file on page 5-8
• Examples of specifying region and section addresses on page 5-23
• Equivalent scatter-loading descriptions for simple images on page 5-33.
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).

To construct the memory map of an image, \texttt{armlink} must have:

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

The scatter-loading mechanism enables you to specify the memory map of an image to \texttt{armlink}. Scatter-loading gives you complete control over the grouping and placement of image components. Scatter-loading can be used for simple images (see Images with a simple memory map on page 5-4), but it is generally only used for images that have a complex memory map (see Images with a complex memory map on page 5-5).

Scatter-loading is capable of describing complex image maps that consist of multiple regions scattered in the memory map at load and execution time.

You specify this information using a scatter-loading description in a text file that is passed to \texttt{armlink}.

This section describes:

- Symbols defined for scatter-loading
- When to use scatter-loading on page 5-3
- Scatter-loading command-line option on page 5-4
- Images with a simple memory map on page 5-4
- Images with a complex memory map on page 5-5.

5.1.1 Symbols defined for scatter-loading

When \texttt{armlink} is creating an image using a scatter-loading description, it creates some region-related symbols. These are described in Region-related symbols on page 4-3. These special symbols are created by \texttt{armlink} only if your code references them.

\textbf{Note}

These symbols are not defined when a scatter-loading description file is used:

- \texttt{Image$$RW$$Base}
- \texttt{Image$$RW$$Limit}
- \texttt{Image$$RO$$Base}
- \texttt{Image$$RO$$Limit}
- \texttt{Image$$ZI$$Base}
- \texttt{Image$$ZI$$Limit}
Because the default implementation uses `Image$$ZI$$Limit`, you must reimplement `__user_initial_stackheap()` and define a value for the start of the heap region and the top of the stack region. For more information see:

- the chapter describing the C and C++ libraries in *RealView Compilation Tools v2.1 Compiler and Libraries Guide* (for details on library memory models)
- the chapter describing how to develop embedded software in *RealView Compilation Tools v2.1 Developer Guide*.

If you do not reimplement `__user_initial_stackheap()`, the following error message is displayed from the linker:

```
Undefined symbol Image$$ZI$$Limit (referred from sys_stackheap.o).
```

### 5.1.2 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 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, the interrupt code might be placed into fast SRAM to improve interrupt response time and 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.

5.1.3 Scatter-loading command-line option
The arm\link command-line option for using scatter-loading is:

```
--scatter description_file
```

This instructs arm\link 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 5-8.

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

- Examples of specifying region and section addresses on page 5-23.
- Equivalent scatter-loading descriptions for simple images on page 5-33.
- The chapter describing how to develop embedded software in RealView Compilation Tools v2.1 Developer Guide.

5.1.4 Images with a simple memory map
The scatter-loading description in Figure 5-1 on page 5-5 loads the segments from the object file into memory corresponding to the map shown in Figure 5-2 on page 5-5. 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.
5.1.5 Images with a complex memory map

The scatter-loading description in Figure 5-3 on page 5-6 loads the segments from the program1.o and program2.o files into memory corresponding to the map shown in Figure 5-4 on page 5-6.

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.
Using Scatter-loading Description Files

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

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

The scatter-loading description in Figure 5-3 on page 5-6 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 5-25).
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 arm\_link. 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

The assignment of input sections 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 5-20.

This section describes:

- BNF notation and syntax on page 5-9
- Overview of the syntax of scatter-loading description files on page 5-9
- Load region description on page 5-11
- Execution region description on page 5-14
- Input section description on page 5-17
- Resolving multiple matches on page 5-20.
5.2.1 BNF notation and syntax

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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;</td>
<td>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&quot;+&quot;C, for example, can only be replaced by the pattern B+C. The definition B+C can be replaced by, for example, patterns B, C, BB, or BBC.</td>
</tr>
<tr>
<td>A ::= B</td>
<td>Defines A as B. For example, A ::= B&quot;+&quot;</td>
</tr>
<tr>
<td>[A]</td>
<td>Optional element A. For example, A ::= B[C]D means that the definition A can be expanded into either BD or BCD.</td>
</tr>
<tr>
<td>A+</td>
<td>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.</td>
</tr>
<tr>
<td>A*</td>
<td>Element A can have zero or more occurrences.</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>(A B)</td>
<td>Element A and B are grouped together. This is particularly useful when the</td>
</tr>
</tbody>
</table>

5.2.2 Overview of the syntax of scatter-loading description files

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 ::= 

execution_region_name (base_address | "+" offset) [attributes] [max_size] 
"{" 
  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_attr | input_section_pattern) 
  ([",",] "+" input_section_attr | ",", input_section_pattern))
  
")"
]

Figure 5-5 on page 5-11 shows the contents and organization of a typical scatter-loading description file.
5.2.3 Load region description

A load region has:
- a name (this is 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 (these identify the type and location of modules in the execution view).
Figure 5-6 shows the components of a typical load region description.

\[
\text{LOAD\_ROM\_1} \ 0\times0000 \\
\{ \\
\ \text{EXEC\_ROM\_1} \ 0\times0000 \\
\ \{ \\
\ \ \text{program1.o} \ (+RO) \\
\ \} \\
\} \\
\text{DRAM} \ 0\times18000 \ 0\times8000 \\
\{ \\
\ \text{program1.o} \ (+RW,+ZI) \\
\ \} \\
\}
\]

**Figure 5-6 Components of a load region description**

The syntax, in BNF, is:

\[
\text{load\_region\_description} ::= \\
\text{load\_region\_name} \ (\text{base\_address} \ | \ ("+" \ \text{offset}) \ [\text{attribute\_list}] \ [\text{max\_size}] \ "]" \\
\ "\{" \\
\ \text{execution\_region\_description}+ \\
\ "}\"
\]

where:

- **load\_region\_name**: The linker generates a \text{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 5-14). 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
This 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 zero.

attribute_list
This specifies the properties of the load region contents:

- **ABSOLUTE**: Absolute address.
- **PI**: Position-independent.
- **RELOC**: Relocatable.
- **OVERLAY**: Overlaid.
- **NOCOMPRESS**: Should not be compressed.

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. armlink 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. 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 should not be compressed in the final image.

max_size
This 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
This specifies the execution region name, address, and contents. See Execution region description on page 5-14.
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.

The syntax of an execution-region description, in BNF, is:

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

where:

- `exec_region_name` This names the execution region.
- `base_address` This 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 `+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:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ABSOLUTE</strong></td>
<td>Absolute address. The execution address of the region is specified by the base designator.</td>
</tr>
<tr>
<td><strong>PI</strong></td>
<td>Position-independent.</td>
</tr>
<tr>
<td><strong>OVERLAY</strong></td>
<td>Overlaid.</td>
</tr>
<tr>
<td><strong>FIXED</strong></td>
<td>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 5-24). The base designator must be either an absolute base address, or an offset of +0.</td>
</tr>
<tr>
<td><strong>EMPTY</strong></td>
<td>This 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 5-29 for further information.</td>
</tr>
<tr>
<td><strong>PADVALUE</strong></td>
<td>This creates a section containing padding. If you specify PADVALUE, you must give a value, for example: EXEC 0x10000 FIXED PADVALUE 0xffffffff 0x2000 The value for PADVALUE in a scatter-loading description file must be a word. PADVALUE attributes on load regions are ignored.</td>
</tr>
<tr>
<td><strong>ZEROPAD</strong></td>
<td>Zero-initialized sections are written in the ELF file as a block of zeroes 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.</td>
</tr>
<tr>
<td><strong>NOCOMPRESS</strong></td>
<td>Should not be compressed.</td>
</tr>
<tr>
<td><strong>UNINIT</strong></td>
<td>Must not be zero-initialized.</td>
</tr>
</tbody>
</table>
You can specify only one of the attributes PI, OVERLAY, FIXED, and ABSOLUTE. Unless one of the attributes PI, FIXED, or OVERLAY 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 will generate a warning and the attribute will be 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 allowed to have overlapping address ranges. armlink 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.

\textbf{max\_size}

This is an optional number that instructs armlink to generate an error if the region has more than \texttt{max\_size} bytes allocated to it.

\textbf{-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 \textit{Reserving an empty region} on page 5-29 for more information.

\textit{input\_section\_description}

This specifies the content of the input sections. See \textit{Input section description} on page 5-17.
### 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`.

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

![Figure 5-8 Components of an input section description](image)

The syntax, in BNF, is:

```plaintext
input_section_description ::= 

  module_select_pattern

  [""
   ("*" input_section_attr | input_section_pattern)
   
   ("","" input_section_attr | "," input_section_pattern))
  
"
```

where:

- `module_select_pattern`
  - This is 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 was extracted from. If the names contain spaces, use wildcards to simplify searching. For example, use `*libname.lib` to match `C:\lib dir\libname.lib`. 

---

**Using Scatter-loading Description Files**

ARM DUI 0206D

Copyright © 2002-2004 ARM Limited. All rights reserved.
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.

--- 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**

This is 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**.

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). The first occurrence of FIRST or LAST as an *input_section_attr* terminates the list.

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.

You cannot have two * selectors in a scatter-loading description file. You can, however, use two modified selectors, *A and *B for example, 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.

The input section descriptions having the .ANY module selector pattern are resolved after all other (non-.ANY) input section descriptions have been resolved and input sections have been assigned to the closest matching execution region. If more than one .ANY pattern is present, the linker fills the first .ANY with as much as possible and then begins filling the next .ANY.

Each remaining unassigned input section is assigned to the execution region with the following characteristics:

- the biggest remaining space (determined by the value of max_size and the sizes of the input sections already assigned to it)
- a matching .ANY *input_section_description*
- memory access attributes (if they exist) matching the memory attributes of the input section.

*input_section_pattern*

This is 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.
5.2.6 Resolving multiple matches

If a section matches more than one execution region, the matches are resolved as described below. If a unique match cannot be found, arm\link faults the scatter-loading description. Each section is selected by a \texttt{module_selector_pattern} and an \texttt{input_section_selector}.

Examples of \texttt{module_selector_pattern} specifications are:

- \* matches any module or library
- \*.o matches any object module
- \texttt{math.o} matches the \texttt{math.o} module
- \*\texttt{math.lib} matches any library path ending with \texttt{math.lib} (for example \texttt{C:\apps\lib\math\satmath.lib}).

Examples of \texttt{input_section_selector} specifications are:

- \texttt{+RO} is an input section attribute that matches all RO code and all RO data
- \texttt{+RW,+ZI} is an input section attribute that matches all RW code, all RW data, and all ZI data
- \texttt{BLOCK_42} is an input section pattern that matches the assembly file area named \texttt{BLOCK_42}.

\textbf{Note}

The compiler produces areas that can be identified by input section patterns such as \texttt{.text}, \texttt{.data}, \texttt{.constdata}, and \texttt{.bss}. These names, however, might change in future versions and you should 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 \texttt{#pragma arm section} to specify the name of the section containing the code or data of interest. See the chapter describing the ARM compiler reference in \textit{RealView Compilation Tools v2.1 Compiler and Libraries Guide} for more information on pragmas.

The following variables are used to describe multiple matches:

- \texttt{m1} and \texttt{m2} represent module selector patterns
- \texttt{s1} and \texttt{s2} represent input section selectors.
In the case of multiple matches, armlink determines the region to assign the input section to on the basis of the module_selector_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, armlink:

- 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_selector_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 sl and s2 are both patterns matching section names, the same definition as for module selector patterns is used.
   - If one of sl, s2 matches the input section name and the other matches the input section attributes, sl and s2 are unordered and the description is diagnosed as faulty.
   - If both sl and s2 match input section attributes, the determination of whether sl 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 (sl more specific than s2) relationship between section attributes.

3. For the module_selector_pattern, input_section_selector pair, m1,s1 is more specific than m2,s2 only if any of the following are true:
   - sl 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
   - sl 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_selectors 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 5-1 shows multiple execution regions and pattern matching.

**Example 5-1 Multiple execution regions and pattern matching**

```
LR_1 0x040000
  {  
    ER_ROM 0x040000 ; The startup exec region address is the same
      {  
        application.o (+ENTRY) ; The section containing the entry point from
          ; as the load address.
      }  
    ER_RAM1 0x048000
      {  
        application.o (+RO-CODE) ; Other RO code from the object goes here
      }  
    ER_RAM2 0x050000
      {  
        application.o (+RO) ; 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
      }  
  }
```
5.3 Examples of specifying region and section addresses

This section describes how to use a scatter-loading description file to specify addresses for:

- veneers
- RO constants in ROM
- root execution regions.

For additional examples on accessing data and functions at fixed addresses, see the chapter describing how to develop embedded software in RealView Compilation Tools v2.1 Developer Guide.

This section describes:

- Selecting veneer input sections in scatter-loading descriptions
- Creating root execution regions on page 5-24
- Placing regions at fixed addresses on page 5-25
- Assigning sections to a root region on page 5-28
- Reserving an empty region on page 5-29
- Using preprocessing directives on page 5-31.

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 3-17). 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, armlink 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.
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 allowing 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 designator (address) for the execution region and the base designator (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 execution regions, then they will all be root regions.

  See Example 5-2.

- Use the FIXED execution region attribute to ensure that the load address and execution address of a specific region are the same. See Example 5-3 on page 5-25 and Figure 5-9 on page 5-25.

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 5-25 for more information.

Example 5-2 Specifying the same load and execution address

```plaintext
LR_1 0x040000 ; load region starts at 0x40000
{            ; start of execution region descriptions
  ER_RO 0x040000 ; load address = execution address
  {           ; all RO sections (must include section with
    * (+RO) ; initial entry point)
  }          ; rest of scatter description...
}
```
Example 5-3 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 sections other than those in init.o
    * (+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
  {                     ; all RO sections from init.o
    init.o(+RO)        ; all RO sections from init.o
  }
}                         ; rest of scatter description...
```

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). You can use this, for example, 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.

Figure 5-9 Memory Map for fixed execution regions
**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. To access an individual object, you can:

- 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 *RealView Compilation Tools v2.1 Compiler and Libraries Guide*). This option increases code size slightly (typically by a few percent) 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 `#pragma arm section` inside the C or C++ source code to create multiple named sections (see Example 5-5 on page 5-28).
- Use the `AREA` directive from assembly language. For assembly code, the smallest locatable unit is an `AREA` (see *RealView Compilation Tools v2.1 Assembler Guide*).

---

### Placing the contents of individual object files

The scatter-loading description file in Example 5-4 on page 5-27 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 5-4 Section placement

```c
LOADREG1 0x0 0x10000
{
  EXCREG1 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 data to be placed at 0x400000
  {                    ; maximum space available for table is 0xFF
    * (+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 0xFFFF
  }
}
```

--- 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 will then copy 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.) For more details, see the chapter describing how to develop embedded software in RealView Compilation Tools v2.1 Developer Guide.

Using the arm section pragma

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 a pragma and scatter-loading description file to place named sections. Create a module (adder.c for example) and name a section explicitly as shown in Example 5-5 on page 5-28.
Example 5-5 Naming a section

```c
// file adder.c
int x1 = 5;                      // in .data
int y1[100];                     // in .bss
int const z1[3] = {1,2,3};       // in .constdata
int sub1(int x) {return x-1;}    // in .text

#pragma arm section rwdata = "foo", code ="foo"
int x2 = 5;                      // in foo (data part of region)
char *s3 = "abc";                // s3 in foo, "abc" in .constdata
int add1(int x)  {return x+1;}   // in foo (.text part of region)
#pragma arm section code, rwdata // return to default placement
```

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

Example 5-6 Placing a section

```c
FLASH 0x24000000 0x4000000
{
    FLASH 0x24000000 0x4000000
    {
        init.o (Init, +First)          ; place area Init from init.o first
        * (+RO)                        ; sub1(), z1[]
    }
    32bitRAM 0x0000
    {
        vectors.o (Vect, +First)
        * (+RW,+ZI)                    ; x1, y1
    }
    ADDER 0x08000000
    {
        adder.o (foo)                  ; x2, string s3, and add1()
    }
}
```

5.3.4 Assigning sections to a root region

In previous releases of RVCT, the only sections that had to be root were __main and the region tables. However, with the implementation of RW data compression there are more files that must be placed in a root region. armlink can detect these sections automatically.
Use a scatter-loading description file to specify a root section in the same way as a named section. Example 5-7 defines a section must be root using the section selector InRoot$S$Sections.

Example 5-7 Specifying a root region

```
LR_APP 0x8000 ; load region starts at 0x8000
{
    ER_ROOT 0x8000
    {
        # (InRoot$S$Sections) ;place section
    }
    ER_WORLD 0x100000
    {
        # (RO, +RW, +ZI)
    }
    ; rest of scatter description...
}
```

5.3.5 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, armlink 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 should be an absolute address and not a relative one. For example, the execution region definition STACK 0x800000 EMPTY -0x10000 in Example 5-8 on page 5-30 defines a region called STACK that starts at address 0x7F0000 and ends at address 0x800000. Figure 5-10 on page 5-30 is a diagrammatic representation of this example.

Note

The dummy ZI region that is created for an EMPTY execution region is not initialized to zero at run time.
If the address is in relative (+n) form and the length is negative, the linker generates an error.

**Example 5-8 Reserving a region for the stack**

```assembly
LR_1 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 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$$Length    = 0x1000
- Image$$HEAP$$ZI$$Base       = 0x800000
- Image$$HEAP$$ZI$$Limit      = 0x810000
- Image$$HEAP$$ZI$$Length     = 0x1000
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.

5.3.6 Using preprocessing directives

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

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

Most typically the command is:

```plaintext
#! 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:

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

arm\link parses the preprocessed scatter-loading description file where these are treated as comments and ignored.

Consider the following simple example:

```plaintext
#define AN_ADDRESS (BASE_ADDRESS+(ALIAS_NUMBER*ALIAS_SIZE))
```

Use the directives:

```plaintext
#define BASE_ADDRESS 0x8000
#define ALIAS_NUMBER 0x2
#define ALIAS_SIZE 0x400
```

If the scatter-loading description file contains:

```plaintext
LOAD_FLASH AN_ADDRESS ; start address
```

Following preprocessing, this evaluates to:

```plaintext
LOAD_FLASH ( 0x8000 + ( 0x2 * 0x400 )) ; start address
```
Following evaluation, \texttt{armlink} parses the scatter-loading file to produce the load region:

\texttt{LOAD\_FLASH\ 0x8808\ ;\ start\ address}
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 3-19. 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.

This section describes:
- **Type 1**
- **Type 2** on page 5-35
- **Type 3** on page 5-37.

5.4.1 Type 1

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 5-9 shows the scatter-loading description equivalent to using --ro-base 0x040000.

**Example 5-9 Single load region and contiguous execution regions**

```plaintext
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 was 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 5-9 on page 5-33 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 5-10 shows the scatter-loading description equivalent to using --ro-base 0x010000 --ropi.

**Example 5-10 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.
   {                   ; The RW sections are placed next. They cannot be moved.
      *(+RW)         ; All the ZI sections are placed consecutively here.
   }
   ER_ZI +0            ; ER_ZI region placed after ER_RW region.
   {                   ; All the ZI sections are placed consecutively here.
      *(+ZI)         
   }
}
```
Shown in Example 5-10 on page 5-34, 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

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 5-11 shows the scatter-loading description equivalent to using --ro-base 0x010000 --rw-base 0x040000.

Example 5-11 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 was 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.
Using Scatter-loading Description Files

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 5-12 shows the scatter-loading description equivalent to using --ro-base 0x010000 --rw-base 0x018000 --rwpi.

Example 5-12 Position-independent data

```
LR_1 0x010000           ; The first load region is at 0x010000.
{                       ; Default ABSOLUTE attribute is inherited from parent. The execution address
  ER_RO +0            ; is 0x010000. The code and ro data cannot be moved.
  {                  ; All the RO sections go here.
    * (+RO)         ; The RW sections are placed at 0x018000 and they can be moved.
  }
  ER_RW 0x018000 PI   ; PI attribute overrides ABSOLUTE
  {                  ; ER_ZI region placed after ER_RW region.
    * (+RW)         ; 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

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 5-13 on page 5-38 shows the scatter-loading description equivalent to using

--ro-base 0x010000 --rw-base 0x040000 --split.
Example 5-13 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.
    }
  }
```

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.

**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 5-14 on page 5-39 shows the scatter-loading description equivalent to using --ro-base 0x010000 --rw-base 0x040000 --reloc --split.
Example 5-14 Relocatable load regions

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

LR2 0x040000 RELOC
{  
   ER_RW + 0  
   {    
      *(+RW)  
   }  
   ER_ZI +0  
   {    
      *(+ZI)  
   }  
}
Using Scatter-loading Description Files
Chapter 6
Creating and Using Libraries

This chapter describes the use of libraries with the ARM® linker, arm1ink, and the librarian, armar. It contains the following sections:

- About libraries on page 6-2
- Library searching, selection, and scanning on page 6-3
- The ARM librarian on page 6-6.
6.1 About libraries

An object file can refer to external symbols that are, for example, functions or variables. `armlink` attempts to resolve these references by matching them to definitions found in other object files and libraries. `armlink` recognizes a collection of ELF files stored in an `ar` format file as a library.
6.2 Library searching, selection, and scanning

The differences between the way armlink 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 armlink 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 `--noremove` is used.

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

**Note**

If the `--noscanlib` option is specified, armlink 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, armlink 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.

This section describes:

- Searching for ARM libraries on page 6-4
- Searching for user libraries on page 6-4
- Scanning the libraries on page 6-5.
6.2.1 Searching for ARM libraries

You can specify the search paths used to find the ARM standard libraries by:

- Using the environment variable RVCT21LIB. This is the default.
- Adding the --libpath argument 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 RVCT21LIB environment variable holds this path.

Note

The linker command-line option --libpath overrides the paths specified by the RVCT21LIB variable.

armlink combines each parent directory, given by either --libpath or the RVCT21LIB 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 armlink 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. armlink must select the best-suited variant from each of the directories identified during the library search.

armlink 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, armlink retains the first library selected and rejects all others.

The final list contains all the libraries that armlink scans in order to resolve references.

For more information on library variants, see the chapter describing the C and C++ libraries in RealView Compilation Tools v2.1 Compiler and Libraries Guide.

6.2.2 Searching for user libraries

You can specify user libraries by:

- Including them explicitly in the input file list.
• Adding the --userlibpath argument 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, armlink tries to locate the library in the directories specified by the --userlibpath argument. 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
```

Note

The search paths used for the ARM standard libraries specified by the RVCT21LIB environment variable or the linker command-line option --libpath are not searched for user libraries (see Searching for ARM libraries on page 6-4).

6.2.3 Scanning the libraries

When armlink 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, armlink 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 armlink 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.

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, armlink generates an error message.
6.3 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 armlink 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 armlink 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 armlink processes its input files, see Chapter 2 The armlink Command Syntax.

This section describes:
- Librarian command-line options
- Examples of armar usage on page 6-9.

6.3.1 Librarian command-line options

The syntax of the armar command when used to add or modify files in the library is:

```
        [--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] [ -C ] [--entries] [ -p ] [ -t ] [ -s ] [--sizes] [ -T ] [--vsn] [-v]
```

where:

- `a` This option places new files in `library` after the file `pos_name`.
- `b` This option places new files in `library` before the file `pos_name`.
- `C` This option 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` This option suppresses the diagnostic message normally written to standard error when a library is created.
--create  This option creates a new library even if library already exists.
-d           This option deletes one or more files from library.
--entries   This option lists all entry points defined in library. The format for the
            listing is:
            ENTRY at offset num in section name of member
file_list   This is 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      This option gives online details of the armar command.
-i           This option places new files in library before the member pos_name
            (equivalent to -b).
library     This is the path name of the library file.
-m           This option 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           This option suppresses the archive symbol table. This is used when the
            library is not an object library.
-p           This option prints the contents of files in library to stdout.
pos_name    This is the name of an existing library member to be used for relative
            positioning. This name must be supplied with options -a, -b, and -i.
-q           This option is an alias for -r.
-r           This option 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 will 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  This option prints a table of contents of library. The files specified by file_list will be included in the written list. If file_list is not specified, all files in the library will be included in the order of the archive.

-s  This option regenerates the archive symbol table.

--sizes  This option lists the Code, RO Data, RW Data, ZI Data, and Debug sizes of each member in library, for example:

```
<table>
<thead>
<tr>
<th>Code</th>
<th>RO Data</th>
<th>RW Data</th>
<th>ZI Data</th>
<th>Debug</th>
<th>Object Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>464</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>8612</td>
<td>app_1.o</td>
</tr>
<tr>
<td>3356</td>
<td>0</td>
<td>0</td>
<td>10244</td>
<td>11848</td>
<td>app_2.o</td>
</tr>
<tr>
<td>3820</td>
<td>0</td>
<td>0</td>
<td>10244</td>
<td>20460</td>
<td>TOTAL</td>
</tr>
</tbody>
</table>
```

-T  This option 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  This option updates older files. When used with the -r option, files within library will be 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

This option instructs the librarian to take options from option_file. See RealView Compilation Tools v2.1 Compiler and Libraries Guide for more information on writing via files.

-v  This option gives verbose output.

The output depends on what other options are used:

-d, -r or -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.
Creating and Using Libraries

-\texttt{x} \quad \text{Prints the filename preceding each extraction.}

\texttt{--vsn} \quad \text{This option prints the version number on standard error.}

\texttt{-x} \quad \text{This option 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.}

\texttt{--zs} \quad \text{This option shows the symbol table.}

\texttt{--zt} \quad \text{Lists member sizes and entry points in library. See \texttt{--sizes} and \texttt{--entries} for output format.}

\textbf{Note} \quad \text{The options \texttt{-a}, \texttt{-b}, \texttt{-C}, \texttt{-i}, \texttt{-m}, \texttt{-T}, \texttt{-u}, and \texttt{-v} are not required for normal operation.}

Options relating to library order (for example, \texttt{-a}, \texttt{-b}, \texttt{-i}, and \texttt{-m}) are not relevant, since the ARM toolchain cannot use a library that does not have a symbol table. (If there is a symbol table, the order is irrelevant.)

Options related to updating a library (\texttt{-C} and \texttt{-u}) are unlikely to be used because, in practice, the libraries are rebuilt rather than updated.

\subsection*{6.3.2 Examples of armar usage}

Syntax examples are shown in Example 6-1 to Example 6-8 on page 6-10.

\textbf{Example 6-1 \textit{Create a new library and add all object files}}

\begin{verbatim}
armar --create mylib *.o
\end{verbatim}

\textbf{Example 6-2 \textit{List the table of contents (verbose)}}

\begin{verbatim}
armar -tv mylib
\end{verbatim}

\textbf{Example 6-3 \textit{List the symbol table}}

\begin{verbatim}
armar --zs mylib
\end{verbatim}
Example 6-4 Add (or replace) files

armar -r mylib obj1.o obj2.o obj3.o ...
armar -ru mylib k*.o

Example 6-5 Add (or replace) files in specified position

armar -r -a obj2.o mylib obj3.o obj4.o ...

Example 6-6 Extract a group of files

armar -x mylib k*.o

Example 6-7 Delete a group of files

armar -d mylib sys_*

Example 6-8 Merge libraries and add (or replace) files

armar -r mylib my_lib.a other_lib.a obj1.o obj2.o obj3.o
This chapter describes the ARM® fromELF software utility provided with RVCT. It contains the following sections:

- About fromELF on page 7-2
- fromelf command syntax on page 7-3
- Examples of fromELF usage on page 7-10.
7.1 About fromELF

The fromELF utility translates *Executable Linkable Format* (ELF) image files produced by armlink into other formats suited to ROM tools and to loading directly into memory. You can also use the fromELF utility to display various information about an ELF object or to generate text files containing the information.

The fromELF utility outputs the following image formats:

- Plain binary format.
- Motorola 32-bit S-record format.
- Intel Hex-32 format.
- Byte Oriented (Verilog Memory Model) Hex format.
- ELF format. You can resave as ELF. For example, you can convert a --debug ELF image to a --nodebug ELF image.

The fromELF utility can also display information about the input file, for example disassembly output or symbol listings, to either stdout or a text file.

Note

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

- fromELF cannot translate the image into other file formats
- fromELF cannot produce a meaningful disassembly listing.

7.1.1 Image structure

The fromELF utility can translate a file from ELF to other formats. It cannot alter 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 armlink at link time.
7.2 fromelf command syntax

This section describes the syntax and options of the fromelf command:

```
fromelf [--base n] [--diag_style arm|ide] [--diag_suppress taglist]
        [--expandarrays] [--fieldoffsets [--select select_options ]] [--help]
        [--nolinkview] [memory_config] [--nodebug] [ [--text] text_output_format]
        code_output_format] [--vsn] [--output output_file] input_file
```

where:

--base n

This option 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
- 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.

--diag_style arm|ide

This option specifies that all warning and error messages are output in a style compatible with IDEs, such as Microsoft Visual Studio.

The default is ARM.

--diag_suppress taglist

This option disables all diagnostic messages that have the specified tag(s).

This option requires a taglist, identifying the number of the message to be suppressed, and 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 ...
```
--fieldoffsets  This option produces, to stdout, 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 RealView Compilation Tools v2.1 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 option 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.

--select select_options

Use --select select_options together with either the --fieldoffsets or --text -a options to select only those fields that match the patterns in the option list.

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 Sun Solaris or Red Hat Linux, you must enclose the options in quotes to prevent the shell expanding the selection.

--help

This option 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.

memory_config

This option outputs multiple files for multiple memory banks. The format of memory_config is --widthxbanks where:

width is the width of memory in the target memory system (8-bit, 16-bit, 32-bit, or 64-bit).

banks specifies the number of memory banks in the target memory system.

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
...
```

`--nodebug`

This option does not put debug information in the output files.
This is the default for binary images. If `--nodebug` 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 --nodebug image -o image_nodb.axf
```

To get ELF format output use the options:

```
fromelf --nodebug --elf image.axf -o image_ndb.axf
```

`--nolinkview`

Use `--nolinkview` to discard the section-level view (link-time 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 --nolinkview --elf image.axf -o image_nlk.axf

--expandarrays
Use the --expandarrays option to print data addresses, including arrays which are expanded both inside and outside structures.
This option can only be used in conjunction with --text=/a.

--text text_output_format
Use a text specification, for example --text -c, to display image information in text format. You can decode an ELF image or ELF object file using this option. This is the default. If no text or code output format is specified, --text is assumed.
If output_file is not specified with the -o option, the information is displayed on stdout.
If a specific text category is not specified, the default is to output header information.
If specified, the text category consists of one or more of the following:

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 table.

t  Prints the string table(s).

v  Prints detailed information on each segment and section header of the image.
Using fromELF

prints dynamic segments contents.

Prints the code and data sizes.

The category selectors can be specified as one of:

- individual options, --text -c -d
- a single concatenated string, --text -cd
- category selectors only, -c -d, -cd
- multiple characters following the =/ characters, --text=/cd.

If an output format is not specified, the default output format of
--text is used and the individual category selectors are
recognized. If another output format is specified, the selectors are
ignored.

code_output_format

This option 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.

--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.

--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.

--vhx Byte Oriented (Verilog Memory Model) Hex 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.
Using fromELF

--elf ELF format (resaves as ELF). This can be used to convert a debug ELF image into a no-debug ELF image.

If you use the fromELF utility 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.

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.

--vsn This option displays fromelf version information.

--output output_file
This option specifies the name of the output file, or the name of the output directory if multiple output files are created (see the description of text_output_format and code_output_format for more information). Specifying the output file is optional with the --text output option and mandatory with all other outputs.

input_file This option specifies the ELF file to be translated.

The fromELF utility 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.
7.3 Examples of fromELF usage

This section contains examples of using the fromELF utility to change image format or extract information from an ELF file.

--- Note ---

If you are using a wildcard character on Sun Solaris or Red Hat Linux, for example, *, ?, or ~, you must enclose the options in quotes to prevent the shell expanding the selection.

For example, enter '* , ~*.*' instead of * , ~*.*.

This section describes:

- Producing a plain binary file
- Disassembly
- Listing field offsets as assembly language EQUs
- Listing addresses of static data on page 7-11
- Converting debug to nodebug on page 7-12.

7.3.1 Producing a plain binary file

To convert an ELF file to a plain binary (.bin) file, enter:

```
fromelf --bin -o outfile.bin infile.axf
```

7.3.2 Disassembly

To produce a listing to stdout that contains the disassembled version of an ELF file, enter:

```
fromelf -c infile.axf
```

To produce a plain-text output file that contains the disassembled version of an ELF file and the symbol table, enter:

```
fromelf -c -s -o outfile.lst infile.axf
```

7.3.3 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, enter:

```
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`, enter:

```
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`, enter:

```
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`, enter:

```
fromelf --fieldoffsets --select tools.top.number* -o outputfile.a inputfile.o
```

### 7.3.4 Listing addresses of static data

To list to `stdout` all the global and static data variables and all the structure field addresses, enter:

```
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, enter:

```
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, enter:

```
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, enter:

```
fromelf --text -a --select *.,~*.* -o strucaddress.txt infile.axf
```
7.3.5 Converting debug to nodebug

To produce a new output file equivalent to using the --nodebug option from an ELF file produced with the --debug option, enter:

`fromelf --nodebug --elf -o outfile.axf infile.axf`
Glossary

The items in this glossary are listed in alphabetical order, with any symbols and numerics appearing at the end.

AAPCS
See Procedure Call Standard for the ARM Architecture.

ABI for the ARM Architecture (base standard) (BSABI)
The ABI for the ARM Architecture is a collection of specifications, some open and some specific to ARM architecture, that regulate the inter-operation of binary code in a range of ARM-based execution environments. The base standard specifies those aspects of code generation that must be standardized to support inter-operation and is aimed at authors and vendors of C and C++ compilers, linkers, and runtime libraries.

ADS
See ARM Developer Suite.

Angel
A debug monitor that enables you to develop and debug applications running on hardware that is based on an ARM core. Angel can debug applications running in either ARM state or Thumb state.

API
Application Program Interface.

Architecture
The term used to identify a group of processors that have similar characteristics.
**Glossary**

**ARM Developer Suite (ADS)**
A suite of software development applications, together with supporting documentation and examples, that enable you to write and debug applications for the ARM family of RISC processors. ADS is superseded by RealView Developer Suite (RVDS).

*See also* RealView Developer Suite.

**ARM state**
A processor that is executing ARM (32-bit) instructions is operating in ARM state.

*See also* Thumb state.

**armar**
The ARM librarian.

**armlink**
The ARM linker.

**Big-endian**
Memory organization where the least significant byte of a word is at a higher address than the most significant byte.

**BNF**
Backus Naur Format. Mathematical notation for defining logical structures.

**BSABI**
*See* ABI for the ARM Architecture (base standard).

**Byte**
A unit of memory storage consisting of eight bits.

**Deprecated**
A deprecated option or feature is one that you are strongly discouraged from using. Deprecated options and features will not be supported in future versions of the product.

**Double word**
A 64-bit unit of information. Contents are taken as being an unsigned integer unless otherwise stated.

**DWARF**
Debug With Arbitrary Record Format.

**ELF**
*See* Executable and Linking Format.

**Environment**
The actual hardware and operating system that an application will run on.

**Executable and Linking Format (ELF)**
The industry standard binary file format used by RealView Compilation Tools. ELF object format is produced by the ARM object producing tools such as armcc and armasm. The ARM linker accepts ELF object files and can output either an ELF executable file, or a partially linked ELF object.

**Execution view**
The address of regions and sections after the image has been loaded into memory and started execution.

**Flash memory**
Non-volatile memory that is often used to hold application code.

**fromELF**
The ARM image conversion utility. This accepts ELF format input files and converts them to a variety of output formats. fromELF can also generate text information about the input image, such as code and data size.
Heap
The portion of computer memory that can be used for creating new variables.

Host
A computer that provides data and other services to another computer.

IDE
Integrated Development Environment (for example, the ARM RealView Debugger IDE).

Image
An executable file that has been loaded onto a processor for execution.

Input section
Contains code or initialized data or describes a fragment of memory that must be set to zero before the application starts.

Interrupt
A change in the normal processing sequence of an application caused by, for example, an external signal.

Interworking
Producing an application that uses both ARM and Thumb code.

Library
A collection of assembler or compiler output objects grouped together into a single repository.

Linker
Software that produces a single image from one or more source assembler or compiler output objects.

Little-endian
Memory organization where the least significant byte of a word is at a lower address than the most significant byte.

Load view
The address of regions and sections when the image has been loaded into memory but has not yet started execution.

Multi-ICE
A JTAG-based tool for debugging embedded systems.

Output section
A contiguous sequence of input sections that have the same RO, RW, or ZI attributes. The sections are grouped together in larger fragments called regions. The regions will be grouped together into the final executable image.

See also Region.

PIC
Position Independent Code.

See also ROPI.

PID
Position Independent Data or the ARM Platform-Independent Development card.

See also RWPI.

Procedure Call Standard for the ARM Architecture (AAPCS)
Procedure Call Standard for the ARM Architecture defines how registers and the stack will be used for subroutine calls.

Profiling
Accumulation of statistics during execution of a program being debugged, to measure performance or to determine critical areas of code.
Call-graph profiling provides great detail but slows execution significantly. Flat profiling provides simpler statistics with less impact on execution speed.

For both types of profiling you can specify the time interval between statistics-collecting operations.

**Program image**

See Image.

**RDI**

See Remote Debug Interface.

**Read-Only Position Independent (ROPI)**

Code or read-only data that can be placed at any address.

**Read Write Position Independent (RWPI)**

Read/write data addresses that can be changed at runtime.

**RealView ARMulator ISS (RVISS)**

The most recent version of the ARM simulator, RealView ARMulator ISS is supplied with RealView Developer Suite. It communicates with a debug target using RV-msg, through the RealView Connection Broker interface, and RDI.

*See also* RDI and RealView Connection Broker.

**RealView Compilation Tools (RVCT)**

RealView Compilation Tools is a suite of tools, together with supporting documentation and examples, that enables you to write and build applications for the ARM family of RISC processors.

**RealView Connection Broker**

RealView Connection Broker is an execution vehicle that enables you to connect to simulator targets on your local system, or on a remote system. It also enables you to make multiple connections to the simulator.

*See also* RealView ARMulator ISS.

**RealView Developer Suite (RVDS)**

The latest suite of software development applications, together with supporting documentation and examples, that enable you to write and debug applications for the ARM family of RISC processors.

**RealView ICE (RVI)**

A JTAG-based debug solution to debug software running on ARM processors.

**Redirection**

The process of sending default output to a different destination or receiving default input from a different source. This is commonly used to output text, that would otherwise be displayed on the computer screen, to a file.
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Region</strong></td>
<td>In an Image, a region is a contiguous sequence of one to three output sections (RO, RW, and ZI). A region typically maps onto a physical memory device, such as ROM, RAM, or peripheral.</td>
</tr>
<tr>
<td></td>
<td><em>See also</em> Root region.</td>
</tr>
<tr>
<td><strong>Remapping</strong></td>
<td>Changing the address of physical memory or devices after the application has started executing. This is typically done to allow RAM to replace ROM once the initialization has been done.</td>
</tr>
<tr>
<td><strong>Remote Debug Interface (RDI)</strong></td>
<td>The Remote Debug Interface (RDI) is an ARM standard procedural interface between a debugger and the debug agent. RDI gives the debugger a uniform way to communicate with:</td>
</tr>
<tr>
<td></td>
<td>• a simulator running on the host (for example, RVISS)</td>
</tr>
<tr>
<td></td>
<td>• a debug monitor running on hardware that is based on an ARM core accessed through a communication link (for example, Angel)</td>
</tr>
<tr>
<td></td>
<td>• a debug agent controlling an ARM processor through hardware debug support (for example, RealView ICE or Multi-ICE).</td>
</tr>
<tr>
<td><strong>Retargeting</strong></td>
<td>The process of moving code designed for one execution environment to a new execution environment.</td>
</tr>
<tr>
<td><strong>Root region</strong></td>
<td>In an image, regions having the same load and execution address. A non-root region is a region that must be copied from its load address to its execution address.</td>
</tr>
<tr>
<td><strong>ROPI</strong></td>
<td><em>See</em> Read Only Position Independent.</td>
</tr>
<tr>
<td><strong>RTOS</strong></td>
<td>Real Time Operating System.</td>
</tr>
<tr>
<td><strong>RVCT</strong></td>
<td><em>See</em> RealView Compilation Tools.</td>
</tr>
<tr>
<td><strong>RVDS</strong></td>
<td><em>See</em> RealView Developer Suite.</td>
</tr>
<tr>
<td><strong>RWPI</strong></td>
<td><em>See</em> Read Write Position Independent.</td>
</tr>
<tr>
<td><strong>Scatter-loading</strong></td>
<td>Assigning the address and grouping of code and data sections individually rather than using single large blocks.</td>
</tr>
<tr>
<td><strong>Sections</strong></td>
<td>A block of software code or data for an Image.</td>
</tr>
<tr>
<td></td>
<td><em>See also</em> Input section and Output section.</td>
</tr>
<tr>
<td><strong>Semihosting</strong></td>
<td>A mechanism whereby the target communicates I/O requests made in the application code to the host system, rather attempting to support the I/O itself.</td>
</tr>
<tr>
<td><strong>Software Development Toolkit (SDT)</strong></td>
<td>Software Development Toolkit (SDT) is an ARM product, now superseded by ADS and RVDS.</td>
</tr>
</tbody>
</table>
Glossary

Software Interrupt (SWI)
An instruction that causes the processor to call a programmer-specifed subroutine. Used by ARM to handle semihosting.

Stack
The portion of memory that is used to record the return address of code that calls a subroutine. The stack can also be used for parameters and temporary variables.

SWI
See Software Interrupt.

Target
The actual target processor, real or simulated, on which the target application is running.

The fundamental object in any debugging session. The basis of the debugging system. The environment in which the target software will run. It is essentially a collection of real or simulated processors.

Thumb state
A processor that is executing Thumb (16-bit) instructions is operating in Thumb state.

See also ARM state.

Vector Floating Point (VFP)
A standard for floating-point coprocessors where several data values can be processed by a single instruction.

Veneer
A small block of code used with subroutine calls when there is a requirement to change processor state or branch to an address that cannot be reached in the current processor state.

VFP
See Vector Floating Point.

Volatile
Memory addresses where the contents can change independently of the executing application are described as volatile. These are typically memory-mapped peripherals.

Word
A 32-bit unit of information. Contents are taken as being an unsigned integer unless otherwise stated.

Zero Initialized (ZI)
R/W memory used to hold variables that do not have an initial value. The memory is normally set to zero on reset.

ZI
See Zero Initialized.
The items in this index are listed in alphabetical order, with symbols and numerics appearing at the end. The references given are to page numbers.

A

ABI 1-3, 1-4
Application Binary Interface 1-3
AREA
directive 3-9, 5-26
armar 1-5, 6-6
command syntax 6-6
options 6-6
options using wildcards 6-7
see also Librarian
armlink 1-2
command syntax 2-8
deprecated features in v2.1 1-6
options 2-8
options using wildcards 2-16
see also Linker

ARMv6
Byte Invariant Addressing mode 2-21
linker options 2-21
ARM/Thumb synonyms 4-2

B

Backus Naur Format (BNF) 5-9
Branching
veneers 3-17
BSABI 1-4

C

Callgraph
creating 2-21
cyclic functions 2-22
function list 2-21
indirect functions 2-22
no stack information 2-22
stack usage 2-21
stack usage rules 2-22
Comdat groups 1-3, 2-13
Command syntax
fromelf 7-3
librarian 6-6
linker 2-8
Comments
in steering files 4-10
in symdefs file 4-9
Complex images
scatter-loading 3-5, 5-5
see also Scatter-loading
Compression
algorithms 3-15
decompressor 3-15
in scatter-loading 3-16
InRoot$$Sections in scatter-loading 3-16, 5-29
LZ77 3-15
NOCOMPRESS in scatter-loading 3-17
RW data 3-14
sections in root regions 3-16
using processor cache 3-16
viewing compressed regions 3-16
Compressions
run-length 3-15
Consolidated section 4-5
D

Data
in scatter-loading 5-26
Data compression 3-14
Diagnostic messages
controlling deprecated options
  warnings 1-6
  controlling old syntax warnings 1-6
Directives
  AREA 3-9, 5-26
  ENDP 2-21
  ENTRY 2-19, 3-6, 3-7, 5-24
  EQU 7-4
  FRAME 2-21
  preprocessing in scatter-loading 5-31
  PRESERVE8 1-4
  PROC 2-21
  REQUIRE8 1-4

E

ELF 1-2, 2-3
  file format 2-10
  format 1-2, 2-3
  fromELF 1-3, 7-2
  fromELF command syntax 7-3
  fromELF debug information 2-13, 7-2
  fromELF images 7-2
  fromELF information 7-2
  fromELF outputs 7-2
  images 7-2
  objects 7-2
  segments 2-2
Eliminating
  common debug sections 3-11
  common groups 1-3, 3-11, 3-12
  common sections 3-11, 3-12
  unused sections 1-3, 3-12
  unused virtual functions 3-12, 3-13
ENDP
  directive 2-21
ENTRY
  directive 2-19, 3-6, 3-7, 5-24
Entry point
  ENTRY directive 3-6

in image 3-6
in image for C/C++ 3-7
initial 3-6
specifying initial 3-6
specifying multiple 3-7
specifying to the linker 2-15
Environment variables
  RVCT21LIB 2-18, 6-4, 6-5
  RVCT21_CLWARN 1-6
EQU
  directive 7-4
Exceptions
  in linker 3-25
Exceptions tables 3-25
  in linker 2-14, 3-25
Execution region
  description 5-14
  naming 4-4
  related symbols 4-3
  reuse of veneers 3-18
  simple images 3-19
  veneers generation 3-10
Execution view
  in memory map 2-2

F

File formats
  ELF 2-10
  fromELF output 7-7, 7-8
Files
  scatter-loading 5-1
  steering 4-10
  symdefs 4-7
  via 2-26, 6-8
FIRST
  sections 3-9
  sorting rules 3-9
FRAME
  directive 2-21
  fromELF 1-3
  examples 7-10
  see also ELF, fromelf
version number 7-9
fromelf
  diagnostics 7-3
  examples 7-10
  messages 7-3
  specifying input format 7-9
  specifying memory banks 7-5
  specifying output format 7-8
  binary 7-8
  ELF 7-9
  I32 7-8
  M32 7-8
  Verilog 7-8
specifying text format 7-7
  using wildcards 7-4
fromelf options
  -a 7-4, 7-7, 7-11
  -base 7-3, 7-8
  -bin 7-8, 7-10
  -c 7-7, 7-10
  -d 7-7
  -diag_style 7-3
  -diag_suppress 7-3
  -e 7-7
  -elf 7-6, 7-7, 7-9, 7-12
  -expandarrays 7-7
  -fieldoffsets 7-4, 7-10, 7-11
  -g 7-6, 7-7
  -help 7-5
  -i32 7-8
  -i32combined 7-8
  -m32 7-8
  -m32combined 7-8
  -nodebug 7-6, 7-12
  -nolinkview 7-6
  -o 7-10, 7-11, 7-12
  -output 7-9
  -r 7-7
  -s 7-7, 7-10
  -select 7-4, 7-11
  -t 7-7
  -text 7-4, 7-6, 7-7, 7-11
  -v 7-7
  -vhx 7-8
  -vsn 7-9
  -y 7-8
  -z 7-8
Functions
  in scatter-loading 5-26

H

HIDE
steering file 4-14
Hiding and renaming symbols 2-14, 4-10

Images
complex 3-5, 5-4
complex using scatter-loading 5-5
creating a symdefs file 4-7
describing complex 3-5
describing simple 3-5
ELF 7-2
ENTRY directive 3-6
entry points 3-6
entry points in C/C++ 3-7
format of a symdefs file 4-8
initial entry point 3-6
memory map information 3-4
reading a symdefs file 4-8
regions 3-3
sections 3-5, 5-8
simple 2-5, 2-10, 3-19
Simple type 1 3-19, 3-20, 5-33
--ropi example 5-34
Simple type 2 3-19, 3-21, 5-35
--reloc example 5-38
--rwpi example 5-36
Simple type 3 3-19, 3-23, 5-37
simple using scatter-loading 5-4, 5-33, 5-35, 5-37
specifying a memory map 3-5
specifying initial entry point 3-6
specifying memory map 3-5
specifying multiple entry point 3-7
structure of 3-2
using a symdefs file 4-7
using scatter-loading description files 3-5, 5-2
writing global symbols to a symdefs file 4-7

Initial entry point 3-6
specifying 3-6

Input section
description 5-17
using wildcards 5-17, 5-19
Installation directory
default location vii

Interworking 1-2
vectors 1-2, 3-17

L
LAST
sections 3-9
sorting rules 3-9
Librarian 6-6
adding object files to image 6-3
command syntax 6-6
creating libraries 6-3
examples 6-9
options 6-6
see also Librarian options
scanning libraries 6-5
searching for ARM libraries 6-4
searching for ARM variants 6-4
searching for user libraries 6-4
using via files 6-8
variants 6-3
Librarian options
-a 6-6, 6-7, 6-8, 6-9
-b 6-6, 6-7, 6-8, 6-9
-C 6-6, 6-9
-c 6-6
--create 6-7, 6-9
d 6-7, 6-8, 6-10
--entries 6-7
--help 6-7
-i 6-7, 6-8, 6-9
-m 6-7, 6-9
-p 6-7, 6-8
-q 6-7
-r 6-7, 6-8, 6-10
-s 6-8, 6-10
--sizes 6-8
-T 6-6, 6-8, 6-9
-t 6-8, 6-9
-u 6-8, 6-9, 6-10
-v 6-8, 6-9
--via 6-8
--vsn 6-9
-x 6-8, 6-9, 6-10
-z 6-10
--zs 6-9
--zt 6-9

Index
about 6-2
adding to image 6-3
and linker, see Linker, libraries
command syntax 6-6
creating 6-3
scanning 6-5
searching 6-4
variants 6-3, 6-4
Linker 1-2
about 2-2
ARM 1-2
ARMv6 options 2-21
best debug view
turning off 2-13
turning on 2-13
big-endian code 2-21
big-endian data 2-21
big-endian mode 2-21
Byte Addressing 2-6, 2-21
Byte Addressing mode 2-21
Byte Invariant Addressing mode 2-21
callgraph 2-21
code and data sizes 2-22
Comdat groups 1-3, 2-13
command syntax 2-8
commands in a via file 2-26
common debug section elimination 3-11
common group elimination 1-3, 3-11, 3-12
common section elimination 3-11, 3-12
compatibility 1-3
declaring 2-4
complex memory map 2-5, 2-10
constructing a partially-linked object 2-4
constructing an image 2-3, 2-5, 2-12
debug information
turning off 2-13
turning on 2-13
default addresses 2-10
deprecated features in v2.1 1-6
diagnostics 2-24, 2-25, 2-26
ELF format 2-10
errors 2-25
Exception support 2-14, 3-25
unwind table 2-14
exceptions table 2-23
exceptions tables
  generate  2-14, 3-25
  remove   2-14
execution information   3-4
feedback on unused functions   2-22
help on   2-4, 2-9
image
  entry point   2-15
  keeping sections   2-16
  load and execution views   3-4
  overview   3-2
image-related information   2-6, 2-21
information   2-19, 2-21, 2-22, 2-25, 2-26
input   2-3, 2-9
libraries
  creating   6-3
  defaults   2-19
  including during link step   6-2
  linker search path   2-18
  scanning   2-19, 6-5
  searching   6-4
  variants   6-3, 6-4
little-endian code   2-21
load information   3-4
local symbols   2-18
memory map   2-23
memory map information   2-2, 2-12, 3-4
messages   2-18, 2-24, 2-25, 2-26
optimization   2-23
options   1-3, 2-8
  see also Linker options
ordering input sections   3-8
output   1-3, 2-3
output file   2-4, 2-10
output formats   1-3, 2-4, 2-10
padding bytes   2-12
partial linking   2-10, 4-8
preprocessing directives in scatter-loading   5-31
regions   3-3
relocatable images   2-11, 3-19, 5-37
remove unused sections   2-19
remove unused virtual functions   2-20
RO section base address   2-10, 3-24
root region   2-2, 3-16
RW data compression   2-12, 2-13, 3-14
RW section base address   2-10
scatter-loading   2-5
  see also Scatter-loading
sections
  input   3-3, 4-5
  output   3-3, 4-5
  simple memory map   2-5, 2-10
  software version   2-9
  specifying input files   2-9
  standard output stream   2-25
steering files   2-14, 4-10
symbols   4-3, 4-6, 4-16
  hiding and renaming   2-14, 4-10
  undefined   2-26
  used in link step   2-23
  symbols in scatter-loading   5-2
  undefined symbols   2-26
  unused section elimination   1-3, 3-12
  unused sections   2-19, 2-22, 2-23
  unused virtual function elimination   3-12, 3-13
  unused virtual functions   2-20, 3-12, 3-13
user libraries
  linker search path   2-19
  searching   6-4
  veneers   2-23
  veneers generation   3-10, 3-17
  version number   2-4, 2-9
  via files   2-7, 2-26
$ symbols   4-16
$ symbols   4-3
Linker options
  --bestdebug   2-13, 3-12
  --BE32   2-21
  --BE8   2-21
  --callgraph   2-21
  --datacompressor id   2-12, 2-13, 3-15
  --datacompressor list   2-12, 3-15
  --datacompressor off   2-13, 3-15
  --debug   2-13
  --diag_style   2-24
  --diag_suppress   2-25
  --edit   2-14, 4-10
  --elf   2-10
  --entry   2-15
  --errors   2-25
  --exceptions   2-14, 3-25
  --exceptions_tables   2-14
  --feedback   2-22
  --first   2-16, 3-12
  --help   2-9
  --info   2-22, 3-12
  --info sizes, totals   2-23
  --keep   2-16, 3-12
  --last   2-17, 3-12
  --libpath   2-18, 6-4, 6-5
  --list   2-25
  --locals   2-18
  --mangled   2-25, 2-26
  --map   2-23, 3-16
  --match crossmangled   2-25
  --nobestdebug   2-13, 3-12
  --nodebug   2-13
  --noexceptions   2-14, 3-25
  --nolocs   2-19
  --noremove   2-19, 3-12
  --noscanlib   2-19
  --output   2-10
  --pad   2-12
  --partial   2-10
  --relloc   2-11, 3-19, 5-34, 5-37
  --remove   2-19, 3-12
  --ro-base   2-10, 3-19, 3-20, 3-21, 3-23, 5-33, 5-34, 5-35, 5-36, 5-37
  --ropi   2-10, 5-34
  --rospit   2-11, 3-19
  --rw-base   2-10, 3-19, 3-21, 3-23, 5-35, 5-36
  --rwpi   2-11, 5-36
  --scanlib   2-19
  --scatter   2-12, 5-4
  --split   2-11, 3-19, 3-23, 3-24, 5-37
  --strict   2-25
  --symdefs   2-23, 4-7
  --unmangled   2-26
  --unresolved   2-26
  --userlibpath   2-19, 6-4
  --verbose   2-26
  --vmode   2-20
  --via   2-26
  --vsn   2-9
  --xref   2-24
M
Memory map 3-4
describing to linker 5-8
execution view 2-2, 3-4
linker 2-23
linker options 2-5, 2-10, 3-19
load regions 2-2
load view 2-2, 3-4
overlaid 5-15
partial linking 2-5, 4-6
specifying 2-12
specifying in image 3-5
uninitialized 5-15

O
Object files
in scatter-loading 5-26

P
Position independence 2-10, 2-11
Pragma
arm section in scatter-loading 5-27
Preprocessor
directives in scatter-loading 5-31
PRESERVE8
directive 1-4
PROC
directive 2-21

R
Regions 3-3

S
Scatter-loading 5-2
complex images 5-5
description file 1-3, 2-12, 3-5
BNF symbols 5-9
element 5-10
execution region 5-14
FIRST 5-19
hierarchy 5-8, 5-10
input section 5-17
LAST 5-19
load region 5-11
multiple execution regions 5-22
multiple matches 5-20
placing data 5-26
placing functions 5-26
placing object file 5-26
placing regions 5-25
pseudo-attributes 5-19
region address 5-23
reserving empty regions 5-28, 5-29
root region 5-24
section address 5-23
sections 5-8
syonyms in 5-18
syntax 5-8
veneers 5-23
FIRST 3-9
LAST 3-9
linker command-line option 5-4
naming execution regions 4-4
region matching 5-20
section placement 2-16, 2-18
simple images 3-19, 5-4, 5-33, 5-35, 5-37
symbols defined by linker 5-2
when to use 5-3
+FIRST 2-16
+LAST 2-18
Sections
aligning 3-9
attributes 2-2
consolidated 4-5
forcing aligning 3-9
grouping 3-5
in region 2-2
in regions 3-3
input 3-3, 4-5
placement 3-8
keeping 2-16
multiple matches in scatter-loading 5-20
ordering execution 3-10
ordering rules 3-8
output 3-3, 4-5
placement 3-8
by attribute 3-8
FIRST and LAST 3-9
placement of 2-16, 2-17, 5-8, 5-19
related symbols 4-5
sorting rules 3-8
used 2-19
SHOW
  steering file  4-15
Simple images  2-5, 2-10, 3-19
  scatter-loading 3-19, 5-4, 5-33,
  5-35, 5-37
Simple type 1 image 3-19, 3-20
Simple type 2 image 3-19, 3-21
Simple type 3 image 3-19, 3-23
Stack
  alignment 1-4
  alignment errors 1-4
Steering file 2-14, 4-10
  commands 4-10, 4-11
    HIDE 4-14
    RENAME 4-11
    RESOLVE 4-12
    SHOW 4-15
comments 4-10
format 4-10
hiding and renaming symbols 4-10
specifying 4-10
using wildcards 4-10, 4-11, 4-12, 4-14, 4-15
Symbol definitions file
  see Symdefs file
Symbols
  across images 4-7
  ARM/Thumb synonyms 4-2
  consolidated sections 4-5
  definitions file 4-7
  hiding and renaming 4-10
  Image$$ undefined 5-2
  in symdefs file 4-9
  linker 2-14, 2-23, 4-3, 4-10, 4-16
  linker-defined 4-6
  mangled 2-25
  multiple definitions of 4-2
  overriding definitions 4-16
  region-related 4-3, 4-4
  scatter-loading 5-2
section-related 4-4, 4-5
  stack 4-4
  unmangled 2-25
  writing to symdefs file 4-7
ZI 4-4
  $ 4-16
  $Sub$ 4-16
  $Super$ 4-16
  $ 4-3
Symdefs file 4-7
  comments 4-9
  creating 4-7
  format 4-8
  identifying string 4-9
  reading 4-8
  symbol information 4-9
  writing global symbols 4-7
Synonyms
  ARM/Thumb 4-2
T
  Thumb 1-2
  branch range 3-10
  in execution sections 3-10
U
  Uninitialized memory 5-15
User libraries
  searching 6-4
V
  Veneers 1-2, 3-17
  ARM to ARM 3-17
  ARM to Thumb 1-2, 3-17
  branching 3-17
  generation 3-17
  in scatter-loading 5-23
  interworking 3-17
  placement 3-18
  reuse 3-18
  size 2-23
  Thumb to ARM 1-2, 3-17
VFE
  awareness 2-20, 3-13, 3-14
  default 2-20, 3-13
  force 2-20, 3-14
  force_no_rtti 2-20, 3-14
  in assembler source files 3-13
  modes 2-20, 3-13, 3-14
  off 2-20, 3-14
  on 2-20, 3-13
  RTTI 2-20, 3-12, 3-13, 3-14
  virtual function elimination 3-12, 3-13
Via files
  in librarian 6-8
via files 2-26, 6-8
  in linker 2-26
Virtual function elimination
  VFE 3-12, 3-13
Virtual functions
  elimination (VFE) 3-12, 3-13
  unused 2-20, 3-12, 3-13
W
Wildcards
  in scatter-loading 5-7, 5-17
  in steering files 4-10, 4-11, 4-12, 4-14, 4-15
  with armar 6-7
  with armlink 2-16
  with fromelf 7-4, 7-10, 7-11
  with input sections 5-17, 5-19
Z
  ZI symbols 4-4
Symbols
  $ symbols 4-16
  $Sub$ 4-16
  $Super$ 4-16
  $ 4-3