ARM® Developer Suite

Version 1.2

Linker and Utilities Guide

ARM

ARM Developer Suite
Linker and Utilities Guide

Copyright © 2001 ARM Limited. All rights reserved.

Release Information

Proprietary Notice

Contents
ARM Developer Suite Linker and Utilities Guide

ARM DUI 0151A

Preface

This preface introduces the *ARM Developer Suite* (ADS) linker and utilities documentation. It contains the following sections:

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

About this book

This book provides reference information for ADS. It describes the command-line options to the linker and other ARM tools in ADS.

Intended audience

This book is written for all developers who are producing applications using ADS. It assumes that you are an experienced software developer and that you are familiar with the ARM development tools as described in ADS *Getting Started*.

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 ADS version 1.2.

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 program and how you can use it to change image format.

Chapter 8 *Using the ARM Profiler*

> Read this chapter for a description of the ARM profiler utility program provided with ADS.

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.

<u>mono</u>space
: Denotes a permitted abbreviation for a command or option. The underlined text can be entered instead of the full command or option name.

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

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

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

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

Further reading

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

ARM periodically provides updates and corrections to its documentation. See http://www.arm.com for current errata sheets and addenda, and for the ARM Frequently Asked Questions list.

ARM publications

This book contains reference information that is specific to development tools supplied with ADS. Other publications included in the suite are:
* *ADS Installation and License Management Guide* (ARM DUI 0139)
* *Getting Started* (ARM DUI 0064)

- *ADS Developer Guide* (ARM DUI 0056)
- *ADS Assembler Guide* (ARM DUI 0068)
- *ADS Compilers and Libraries Guide* (ARM DUI 0067)
- *AXD and armsd Debuggers Guide* (ARM DUI 0066)
- *ADS Debug Target Guide* (ARM DUI 0058)
- *CodeWarrior IDE Guide* (ARM DUI 0065).

The following additional documentation is provided with the ARM Developer Suite:

- *ARM Architecture Reference Manual* (ARM DDI 0100). This is supplied in DynaText and PDF format.

- *ARM Applications Library Programmer's Guide*. This is supplied in DynaText and PDF format.

- *ARM ELF specification* (SWS ESPC 0003). This is supplied in PDF format in `install_directory`\PDF\specs\ARMELF.pdf.

- *TIS DWARF 2 specification*. This is supplied in PDF format in `install_directory`\PDF\specs\TIS-DWARF2.pdf.

- *ARM/Thumb® Procedure Call Standard specification*. This is supplied in PDF format in `install_directory`\PDF\specs\ATPCS.pdf.

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.

Additional code examples and information about writing code can be found in the *ARM Firmware Suite* (AFS) example code and documentation. Refer to the following ARM publications for more details on AFS:
- *ARM Firmware Suite User Guide* (ARM DUI 0136)
- *ARM Firmware Suite Reference Guide* (ARM DUI 0102).

Other publications

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.

The following book gives general information about the ARM architecture:

- *ARM System-on-chip Architecture* (second edition), Furber, S., (2000). Addison Wesley. ISBN 0-201-67519-6.

Feedback

ARM Limited welcomes feedback on both ADS and the documentation.

Feedback on the ARM Developer Suite

If you have any problems with ADS, please contact your supplier. To help them provide a rapid and useful response, please 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 have any problems with this book, please 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 and utility programs provided with ADS. It contains the following sections:

- *About the linker and utilities* on page 1-2
- *The linker* on page 1-2
- *The utilities* on page 1-3.

1.1 About the linker and utilities

ADS consists of a suite of applications, together with supporting documentation and examples, that enable you to write and debug applications for the ARM family of RISC processors. You can use ADS to develop, build, and debug C, C++, and ARM assembly language programs.

The ADS toolkit consists of the following major components:

- command-line development tools
- GUI development tools
- utilities
- supporting software.

This book describes the ARM linker, armlink, and the command-line utility tools provided with ADS. See *ARM publications* on page vii for a list of the other books in the ADS documentation suite that give information on the ARM assemblers, compilers, debuggers, and supporting software.

1.1.1 The linker

combine and produce

armlink combines the contents of one or more object files with selected parts of one or more object libraries to produce an ELF executable image, or a partially linked ELF object.

generate veneers

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.

specify memory location

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-load 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.

elimination and reduce

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 callgraph and list the stack usage over it
- control the contents of the symbol table in output images.

select

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 does not generate output formats other than ELF. To convert ELF images to other format, such as plain binary for loading into ROM, use the fromELF utility. See *The utilities*.

See Chapter 2 *The armlink Command Syntax* for detailed information on the ARM linker.

1.1.2   The utilities

This section gives an overview of the utility tools that are provided to support the main development tools, including:

*   *fromELF*
*   *armar*
*   *armprof*.

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 detailed information.

armar

The ARM librarian armar enables you to collect and maintain sets of ELF files in ar format libraries. You can pass libraries to the linker in place of several ELF object files. See *The ARM librarian* on page 6-6 for detailed information.

armprof

The ARM profiler armprof displays an execution profile of a simple program from a profile data file generated by an ARM debugger. See Chapter 8 *Using the ARM Profiler* for detailed information.

Chapter 2
The armlink Command Syntax

This chapter describes the full command syntax for armlink. This chapter contains the following sections:

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

2.1   About armlink

armlink, the ARM linker, 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 future 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 are equivalent to ELF segments.

Load regions typically exist in the system memory map at reset or after the image is loaded into the target by a debugger. Before you can execute the image, you might have to move some of its regions to their execution addresses. The memory map of an image therefore has two distinct views:

- the load view of the memory when the program and data are first loaded
- the execution view of the memory after code is moved to its normal execution location.

The term *root region* is used to describe regions that execute at their load address.

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

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 vii for more information.

- Optionally, one or more libraries created by armar as described in Chapter 6 *Creating and Using libraries*.

——— **Note** ———

For backward compatibility, armlink also accepts object files in AOF format and libraries in ALF format. These 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

- eliminates duplicate copies of 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 components objects by name in a scatter-loading file.

2.1.3   Summary of armlink options

This section gives a brief overview of each armlink command-line option. The options are arranged into functional groups.

Accessing help and information

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

```
-help
```

To get the tool version number use:

```
-vsn
```

Specifying the output type and the output file name

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

```
-partial
```

Name the output file using the following option:

```
-output
```

Specify the output file format using the following option:

```
-elf
```

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

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 *ADS Compilers and Libraries Guide* for more information.

Specifying memory map information for the image

Use the following options to specify simple memory maps:

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

For more complex images, use the option:

```
-scatter
```

Scatter-loading is described in Chapter 5 *Using Scatter-loading description files*.

The `-scatter` option is mutually exclusive with the use of any of the simple memory map options `-ro-base`, `-rw-base`, `-reloc`, `-ropi`, `-rwpi`, or `-split`. If you use the `-scatter` option, you must provide a scatter-loading description file and a reimplementation of the `__user_initial_stackheap()` function.

The simple memory map options cannot be used for partial linking because they specify the memory map of an executable image. See the *ADS Developer Guide* for more information.

Controlling image contents

These options control various miscellaneous factors affecting the image contents:

```
-debug | -nodebug
-entry
-first
-keep
-last
-libpath
-edit
-locals | -nolocals
-remove | -noremove
-scanlib | -noscanlib
-match
```

Generating image-related information

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

```
-callgraph
-info
-map
-symbols
-symdefs
-xref
-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:

```
-errors
-list
-verbose
-strict
-unresolved
-mangled
-unmangled
```

2.2   armlink command syntax

—— **Note** ——

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

The complete linker command syntax is:

```
armlink [-help] [-vsn] [-partial] [-output file] [-elf] [-reloc]
[[-ro-base address]] [-ropi] [-rw-base address] [-rwpi] [-split] [-scatter file]
[[-debug]|[-nodebug]] [[-remove (RO/RW/ZI/DBG)]|[-noremove]] [-entry location]
[-keep section-id] [-first section-id] [-last section-id] [-libpath pathlist]
[[-scanlib]|[-noscanlib]] [[-locals]|[-nolocals]] [-callgraph] [[-info topics]]
[-map] [-symbols] [[-symdefs file]] [[-edit file]] [-xref]
[[-xreffrom object(section)]] [[-xrefto object(section)]] [[-errors file]]
[[-list file]] [-verbose] [[-unmangled]|[-mangled]] [-match crossmangled]
[[-via file]] [-strict] [[-unresolved symbol]] [-MI|-LI|-BI] [input-file-list]
```

where:

-help	This option prints a summary of some commonly used command-line options.
-vsn	This option displays the armlink version information.
-partial	This option creates a partially-linked object instead of an executable image.
-output file	This option specifies the name of the output file. The file can be either a partially-linked object or executable image. If the output file name 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.

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

-reloc　　　　　This option makes relocatable images.

A relocatable image has a dynamic segment that contains relocations that can be used to relocate the image post link-time. Examples of post link-time relocation include advanced ROM construction and dynamic loading at 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, but the single load region has the RELOC attribute. See *Type 1: one load region and contiguous output regions* on page 3-14 for more information on type 1 images.

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 noncontiguous output regions* on page 3-17 for more information on type 3 images.

Using -reloc -rw-base without also using -split causes an error.

-<u>ro</u>-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, the default RO base address is 0x8000.

-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.

——— **Note** ———

The ARM tools cannot determine if the final output image will be read-only position-independent until armlink finishes processing input sections. This means that armlink might emit ROPI error messages, even though you have selected the ROPI option for the compiler and assembler.

-<u>rw</u>-base *address*	This option sets the execution addresses of the region containing the RW output section at *address*. *address* must be word-aligned.
	If this option is used with -split, it sets both the load and the execution address of the region containing the RW output sections at *address*.
-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 tables that are used when RO and RW regions are copied or initialized.

——— **Note** ———

The compiler does not force your writable data to be position-independent. This means that armlink might diagnose that data is not RWPI, even though you have selected the RWPI option for the compiler and assembler.

———————————

-split	This option splits the default load region, that contains the RO and RW output sections, into two 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 output section. 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.

 ARM DUI 0151A

-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*.

—— **Note** ——

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

-<u>d</u>ebug   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 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 as loaded into the debugger. 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 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.

-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 to be 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.
>
> ——— **Caution** ———
>
> You must take care not to remove exception handlers 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.

-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 int_handler`

If multiple definitions of *symbol* exist, armlink generates an error message.

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(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. If this form is used and there is more than one non-debug input section in `object`, armlink generates an error message.

-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:

`-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:
>>>
>>> `-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:
>>>
>>> `-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:

>>> `-keep dspdata.o`
>>>
>>> To keep the single input section from each of the objects that have a name starting with dsp, use:
>>>
>>> `-keep dsp*.o`
>>>
>>> There can be multiple occurrences of this option on the command line.

-first *section-id* This option places the selected input section first in its execution region. This can, for example, place the section containing the reset and interrupt vector addresses 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 more than one section cannot 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)`

       ARM DUI 0151A

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 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.

-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:
            `-last checksum`

object(*section*)

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

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

———— **Note** ————

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

————————————

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.

—— **Note** ——

This option does not affect searches for user libraries.

These paths override the path specified by the ARMLIB environment variable. *pathlist* is a comma-separated list of paths *path1, path2,... pathn* that are only used to search for required ARM libraries. The default path for the directory containing the ARM libraries is specified by the ARMLIB environment variable. See *Library searching, selection, and scanning* on page 6-3 for more information on including libraries.

-<u>scan</u>lib   This option enables scanning of default libraries (the standard ARM C and C++ libraries) to resolve references. This is the default.

-<u>noscan</u>lib   This option prevents the scanning of default libraries in a link step.

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

-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.

-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. For each function func it lists:

- the processor state for which the function is compiled (ARM or Thumb)
- the set of functions that call func
- the set of functions that are called by func
- the 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
- allowed to remain undefined (weak references).

The static callgraph also provides stack usage information. 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 in the callgraph.

-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:

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   Gives totals of the Code and Data (RO Data, RW Data, ZI Data, and Debug Data) sizes for input objects and libraries.

veneers   Gives details of armlink-generated veneers. For more information on veneers see *Veneer generation* on page 3-13.

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

—— **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 debugging and 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 definition 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.

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 definition file as input when linking another image. See *Accessing symbols in another image* on page 4-5 for more information.

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

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

See *Hiding and renaming global symbols* on page 4-9 for more information on steering file syntax.

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

 ARM DUI 0151A

-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.

-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 the output image is being written to.

-<u>v</u>erbose   This option prints detailed information about the link operation, including the objects that are included and the libraries from which they are taken.

-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.

-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.

-match crossmangled

This tells 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.

-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 the *ADS Compilers and Libraries Guide* for more information on writing via files.

-strict

This option instructs the linker to report conditions that might result in failure as errors, rather than warnings.

-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.

This option does not display warnings.

input-file-list   This is a space-separated list of objects or libraries.

A special type of object file, 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 4-5 for more information.

There are two ways you can use libraries in the input file list:

*   Specify particular members to be extracted from a library and added to the image as individual objects. For example, specify `mystring.lib(strcmp.o)` in the input file list.

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

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

    unconditionally adds all `a*.o` objects, and `stdio.o`.

3. Libraries are added to the list of libraries that are later used to resolve any remaining non-weak unresolved references.

Chapter 3
Using the Basic Linker Functionality

specify image structure
specify memory locations

This chapter describes the basic armlink functionality. 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 3-12
- *Using command-line options to create simple images* on page 3-14.

For information about advanced linker functionality, refer to the following sections:

Symbol access   Chapter 4 *Accessing Image Symbols*.

Scatter-loading   Chapter 5 *Using Scatter-loading description files*.

Libraries   Chapter 6 *Creating and Using libraries*.

Changing format   Chapter 7 *Using fromELF*.

3.1 Specifying the image structure

The structure of an image is defined by:

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

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.
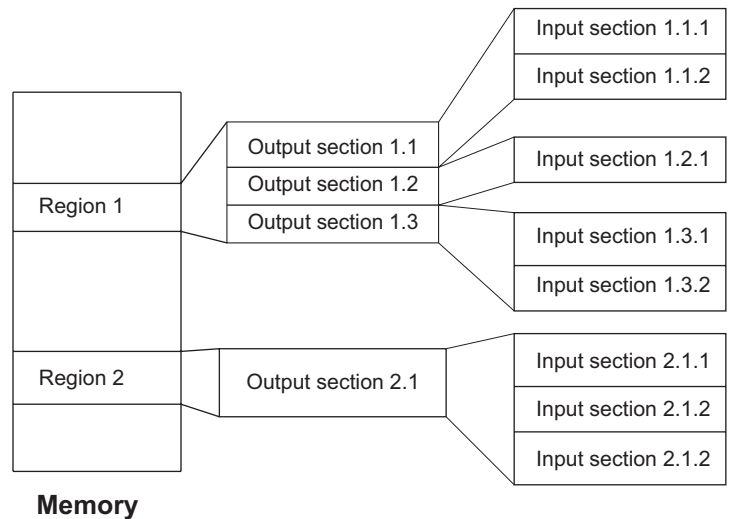
Memory

Figure 3-1 Building blocks for an image

Input sections

An input section contains code or initialized data, or describes a fragment of memory that is not initialized or that must be set to zero before the image can execute. 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 two 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.
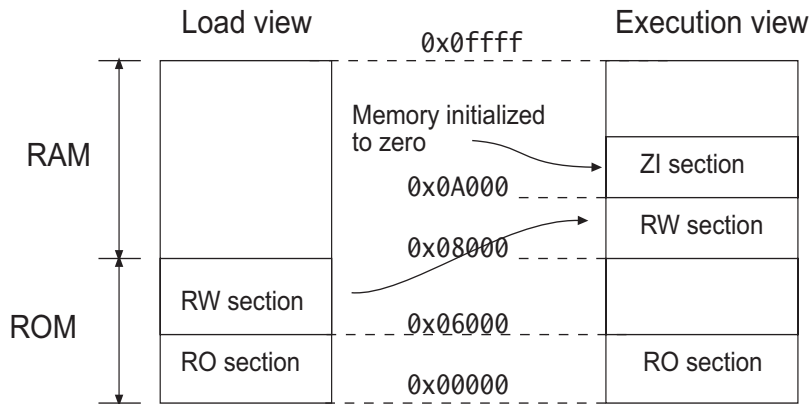
Figure 3-2 Load and execution memory maps

Table 3-1 compares the load and execution views.

Table 3-1

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

3.1.3 Specifying an image memory map

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

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

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

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

Using command-line options

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

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

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 3-14.

Using a scatter-loading description file

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

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.

> There can be only one initial entry point in an image. 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.

An image must contain an initial entry point if it is to be executed by a loader, such as a boot loader or operating system program loader. For example, an image that is an OS is loaded by the boot loader and entered at the initial entry point specified in the executable file header. After the image is loaded it overwrites the boot loader and becomes the OS. In the case of an operating system, the image contains additional entry points, such as the exception handler addresses.

For embedded applications with ROM at zero use -entry 0x0 (or 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 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 the linker attempts to set one if it can:

- If you do not use the -entry option and 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 output object that does not contain an initial entry point if you do not use the -entry option and 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.

  If the output object does not contain an initial entry point, 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 the *ADS 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:

- By attribute.
- By input section name.
- 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 3-10.

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 run-time on systems that have memory management hardware. RO sections can also be placed into ROM in the target.

3.2.1 Ordering input sections by attribute

Portions of the image associated with a particular language run-time system are collected together into a minimum number of contiguous regions. armlink orders input sections by attribute as follows:

- read-only code
- read-only data
- read-write code
- other initialized data
- zero-initialized (uninitialized) data.

Input sections that have the same attributes are ordered by their names. 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-load 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.

         ARM DUI 0151A

3.2.3 Aligning sections

When input sections have been ordered and before the base address is fixed, armlink inserts padding, if required, to force each input section to start at an address that is a multiple of the input section alignment. You can expand the alignment (forcing something that would normally be 4-byte aligned to be 8-byte aligned), but you cannot reduce the natural alignment (forcing 2-byte alignment on something that would normally be 4-byte aligned). Input sections are commonly aligned at word boundaries. The input section alignment can be specified by using either:

ALIGN   Use this attribute to the AREA directive from assembly language. The input section address will be a multiple of $2^{(value\ in\ align\ attribute)}$.

__align   Use this qualifier from C. The input section address will be a multiple of the value specified (the value must be a power of 2).

See Example 3-1 and the description of __align in the *ADS Compilers and Libraries Guide*. See Example 3-2 and the description of ALIGN in the *ADS Assembler Guide*.

Example 3-1 Using the __align qualifier

```
__align(8) long long big_var; // big_var eight-byte aligned
```

Example 3-2 Using the ALIGN attribute in assembly code

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

3.3 Optimizations and modifications

armlink performs some optimizations and modifications in order to remove duplicate sections, enable interworking between ARM and Thumb code, and enable long branches.

3.3.1 Common debug section elimination

The compilers and assemblers generate one set of debug sections for each source file that contributes to a compilation unit. armlink can detect multiple copies of a debug section for a particular source file and discard all but one copy in the final image. This can result in a considerable reduction in image debug size.

3.3.2 Common section elimination

If there are inline functions or templates used in the source, the ARM C++ compilers generate 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 compilers compile these functions into separate instances of common code sections and armlink retains just one copy of each common code section.

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

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

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

An input section is retained in the final image in 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 -first or -last option (or a scatter-loading equivalent)
- if it has been marked as unremovable by the -keep option.

3.3.4 Veneer generation

armlink must generate veneers when:

- a branch involves change of state between ARM state 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 capability and interworking capability.

Thumb to ARM   Long branch capability 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 to be within range of other input sections. If the sections are not within addressing range, long branching is not possible.

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.

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.

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.

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

- 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 files. See *Equivalent scatter-loading descriptions for simple images* on page 5-28 for more information on scatter-loading files

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 on page 3-15).

 ARM DUI 0151A

RAM

RW output section

RO output section

-ro-base value

Single load region

0x0008000

0x0

Load view

ZI output section

RW output section

RO output section

ZI execution region

RW execution region

RO execution region

Execution view

Figure 3-3 Type 1

Load view

The single load region consists of the RO and RW output sections placed consecutively. 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 address 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.

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.

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).

Figure 3-4 Type 2

Load view

In the load view, the single load region consists of the RO and RW output sections placed consecutively, in ROM for example. 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.

                   ARM DUI 0151A

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 noncontiguous 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).

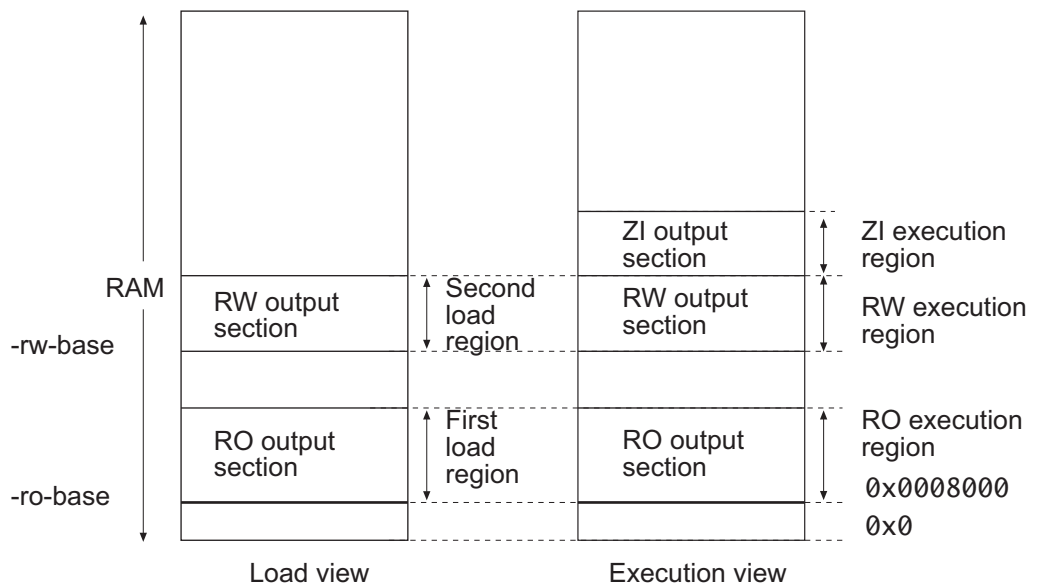**Figure 3-5 Type 3**

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 are not moved from their load address to their execution address.

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:

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

-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 4-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 4-byte aligned *address*. If this option is used with -split, both the load and execution addresses of the RW region are specified.

Using -reloc -rw-base without also using -split causes an error.

Chapter 4
Accessing Image Symbols

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

- *Accessing linker-defined symbols* on page 4-2
- *Accessing symbols in another image* on page 4-5
- *Hiding and renaming global symbols* on page 4-9
- *Using $Super$$ and $Sub$$ to overide symbol definitions* on page 4-12.

4.1 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 ARM-reserved names. 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.

——— **Note** ———

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

4.1.1 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.

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*).

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

- `ER_RO`, 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 `$$ZI$$`.

Table 4-1 Region-related linker symbols

Symbol	Description
Load$$*region_name*$$Base	Load address of the region.
Image$$*region_name*$$Base	Execution address of the region.
Image$$*region_name*$$Length	Execution region length in bytes (multiple of 4).
Image$$*region_name*$$Limit	Address of the byte beyond the end of the execution region.

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

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

——— **Note** ———

• The ZI output sections of an image are not created statically, but are automatically created dynamically at run-time. 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 *ADS Compilers and Libraries Guide* for more information. See also the description of retarget.c in the Writing Code for ROM chapter of the *ADS Developer Guide* for an example of how to do this in C.

Example 4-1

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

4.1.2   Section-related symbols

The output section symbols shown in Table 4-2 are generated if command-line options are used 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.

Table 4-2 Section-related linker symbols

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

———— **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 noncontiguously, armlink diagnoses an error because the use of the base and limit symbols over noncontiguous memory usually produces unpredictable and undesirable effects.

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

The standard implementation of __user_initial_stackheap() uses the value in Image$$ZI$$Limit. Therefore, if you are using a scatter-loading file you must reimplement __user_initial_stackheap() to set the heap and stack boundaries. See the section on library memory models in the *ADS Compilers and Libraries Guide* and *Region-related symbols* on page 4-2.

4.2 Accessing symbols in another image

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

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

4.2.1 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.

4.2.2 Creating a symdefs file

Use armlink option -symdefs *filename* to produce a symdefs file.

armlink 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 armlink rewrites the file.

———————————

Outputting a subset of the global symbols

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

When *filename* exists, armlink 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*. armlink creates a symdef 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 to, for example, update the symbol definitions after one or more objects use to create *image1* have changed.

armlink updates a symdefs file as follows:

1. armlink creates a temporary output file.

2. Lines from *filename* are processed:

   a. The comments and blank lines are copied unchanged to the temporary file.

   b. When a symbol name is found in *filename*, the definition of the named symbol from the current image is output to the temporary file.

      ——— **Note** ———

      If a symbol occurs in *filename* more than once, only one definition of the symbol is placed in the temporary file.

      If a symbol is found in *filename* but is not defined in the current image, no output will be produced for that symbol.

3. If the final link is successful, *filename* is deleted and the temporary file renamed as *filename*.

4.2.3 Symdefs file format

The symdefs file is a type of object file that 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 4-2.

Example 4-2

```
#<SYMDEFS># ARM Linker, ADS1.2 [Build 776]: Last Updated: Tue Jul 24 13:56:35 2001
;value type name, this is an added comment
0x00001000 A function1
0x00002000 T function2
    # This is also a comment, blank lines are ignored

0x00003300 A function3
0x00003340 D table1
```

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:

- Any line where the first non-whitespace character is ; or # is a comment.

- 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.

- A ; or # after the first non-whitespace character does not start a comment.

- Blank lines are ignored and can be inserted to improve readability.

Symbol information

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

Symbol value   armlink writes the absolute address of the symbol in fixed hexadecimal format, for example `0x00004000`. If you edit the file, you can use either hexadecimal or decimal formats for the address value.

Type flag   The single letter for type is:

     A   ARM code

     T   Thumb code

     D   Data.

Symbol name   The name of the symbol.

4.3 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` *filename* to specify the steering file (see the description of the `-edit` option in *armlink command syntax* on page 2-8).

The following commands are supported:
- *RENAME* on page 4-10
- *HIDE* on page 4-11
- *SHOW* on page 4-11.

4.3.1 Steering file format

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

- Lines with a semicolon (;) or hash (#) character as the first non-whitespace character are interpreted as comments. A comment is treated as a blank line.

- Blank lines are ignored.

- Each non blank, non comment line is either a command, or part of a command that is split over consecutive 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 wild card characters in operands.

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

4.3.2   Steering file commands

The following steering file commands enable you to rename, hide, and show symbols in the symbol table.

RENAME

The RENAME command renames defined and undefined global symbol names.

Syntax

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

where:

pattern   Is a string, optionally including wildcard characters, that matches zero or more global symbols. If *pattern* does not match any global symbol, the linker ignores the command. The operand can match both defined and undefined symbols.

replacement_pattern

Is a string, optionally including wild card characters, to which the symbol is to be renamed. Wild cards must have a corresponding wild card in *pattern*. The characters matched by the *pattern* wild card 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.

armlink 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.

HIDE

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

Syntax

HIDE *pattern* [,*pattern*] *

where:

pattern   Is a string, optionally including wildcard characters, that matches zero or more defined global symbols. If *pattern* does not match any defined global symbol, the linker ignores the command. You cannot hide undefined symbols.

SHOW

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

Syntax

SHOW *pattern* [,*pattern*] *

where:

pattern   Is a string, optionally including wildcard characters, that matches zero or more global symbols. If *pattern* does not match any global symbol, the linker ignores the command.

4.4 Using $Super$$ and $Sub$$ to overide 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 $Sub$$foo() and $Super$$foo():

$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.

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

Example 4-3 shows the legacy function foo() modified to result in a call to ExtraFunc() and a call to foo(). See the ARMELF.pdf documentation in the pdf\specs directory for more details.

Example 4-3

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

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

Chapter 5
Using Scatter-loading description files

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

- *About scatter-loading* on page 5-2
- *The formal syntax of the scatter-loading description file* on page 5-7
- *Equivalent scatter-loading descriptions for simple images* on page 5-28.

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).

The scatter-loading mechanism enables you to specify the memory map of an image to armlink. Scatter-loading gives you complete control over the grouping and placement of image components. It is capable of describing complex image maps consisting of multiple regions scattered in the memory map at load and execution time. Scatter-loading can also be used for simple images (see Figure 5-2 on page 5-5), but it is generally only used for images that have a complex memory map (see Figure 5-4 on page 5-6).

To construct the memory map of an image, 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.

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

5.1.1 Symbols defined for scatter-loading

When 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 4-2. These special symbols are created by armlink only if your code references them.

——— **Caution** ———

The symbols `Image$$RW$$Base`, `Image$$RW$$Limit`, `Image$$RO$$Base`, `Image$$RO$$Limit`, `Image$$ZI$$Base`, and `Image$$ZI$$Limit` are not defined when a scatter-loading description file is used.

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. See the section on library memory models in the *ADS Compilers and Libraries Guide* and the section on writing code for ROM in the *ADS Developer Guide* for more information. 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 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.

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 and the location of the enclosing module can be specified 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   Command-line option

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

`-scatter` *description_file_name*

This instructs armlink to construct the image memory map as described in *description_file_name*. The format of the description file is given in *The formal syntax of the scatter-loading description file* on page 5-7.

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

- *Examples of specifying region and section addresses* on page 5-22
- *Equivalent scatter-loading descriptions for simple images* on page 5-28
- the section on writing code for ROM in the *ADS Developer Guide*.

5.1.4   Images with a simple memory map

The scatter-loading description in Figure 5-1 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 specification for the regions are optional, but allow the linker to check that a region has not overflowed its boundary.
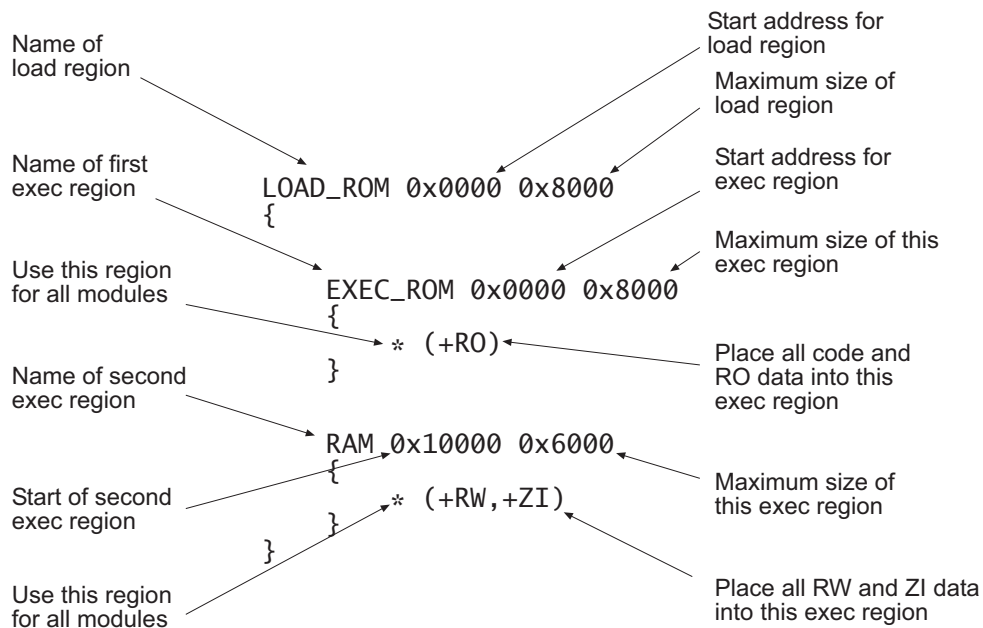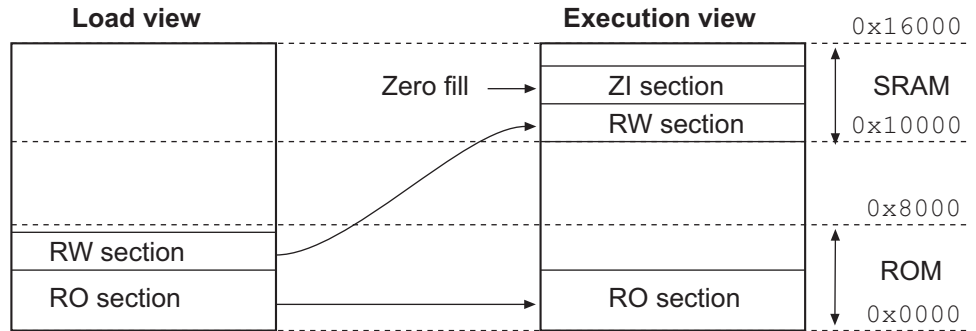

Figure 5-1 Simple file contents

Figure 5-2 Memory map

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

5.1.5   Images with a complex memory map

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

```
LOAD_ROM_1 0x0000                          Start address for first load region
{
    EXEC_ROM_1 0x0000                      Start address for first exec region
    {
        program1.o (+RO)                   Place all code and RO data from
    }                                      program1.o into this exec region
    DRAM 0x18000 0x8000                    Start address for this exec region
    {
        program1.o (+RW,+ZI)               Maximum size of this exec region
    }
}                                          Place all RW and ZI data from
                                           program1.o into this exec region
LOAD_ROM_2 0x4000
{
    EXEC_ROM_2 0x4000                      Start address for second load region
    {
        program2.o (+RO)                   Place all code and RO data from
    }                                      program2.o into this exec region

    SRAM 0x8000 0x8000
    {                                      Place all RW and ZI data from
        program2.o (+RW,+ZI)               program2.o  into this exec region
    }
}
```

Figure 5-3 File contents for a complex memory map

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 5-5 specifies the location for code and data for program1.o and program2.o only. If you link an additional module, program3.o for example, 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 5-24).

                                       ARM DUI 0151A

5.2 The 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 armlink. The file extension for the description file is not significant if you are using the linker from the command line. If you are using the CodeWarrior IDE however, the default extension for a description file is .scf. (However you can change the extension the CodeWarrior IDE recognizes as a scatter-loading file.) The description file enables you to specify:

- the load address and maximum size of each load region
- the attributes of each load region
- the execution regions derived from each load region
- the execution address and maximum size of each execution region
- the 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 5-18.

5.2.1 BNF notation and syntax

Table 5-1 summarizes the BNF symbols that are used to describe a formal language.

Table 5-1 BNF syntax

Symbol	Description
"	Quotation marks are used to indicate that a character that is normally part of the BNF syntax is used as a literal character in the definition.
	The definition B"+"C, for example, can only be replaced by the pattern B+C. The definition B+C can be replaced by, for example, patterns BC, BBC, or BBBC.
A ::= *B*	Defines *A* as *B*. For example, A::= B"+" | C means that *A* is equivalent to either B+ or C.
	The ::= notation is used to define a higher level construct in terms of its components. Each component might also have a ::= definition that defines it in terms of even simpler components.
	For example, A::= B and B::= C | D means that the definition *A* is equivalent to the patterns C or D.
[*A*]	Optional element *A*. For example, A::= B[C]D means that the definition *A* can be expanded into either BD or BCD.
A+	Element *A* can have one or more occurrences. A::= B+ means that the definition *A* can be expanded into B, BB, or BBB for example.
*A**	Element *A* can have zero or more occurrences.
A | *B*	Either element *A* or *B* can occur, but not both.
(*A B*)	Element *A* and *B* are grouped together. This is particularly useful when the | operator is used or when a complex pattern is repeated.
	For example, A::=(B C)+ (D | E) means that the definition *A* can be expanded into any of BCD, BCE, BCBCD, BCBCE, BCBCBCD, or BCBCBCE.

5.2.2   Overview of the syntax of scatter-loading description files

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))*
    ")" ]
```

The contents and organization of a typical scatter-loading description file are shown in Figure 5-5.

```
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)
    }
}
```

Scatter description
Load region description
Execution region description
Input section description
Execution region description
Input section description
Load region description
Execution region description
Input section description
Execution region description
Input section description

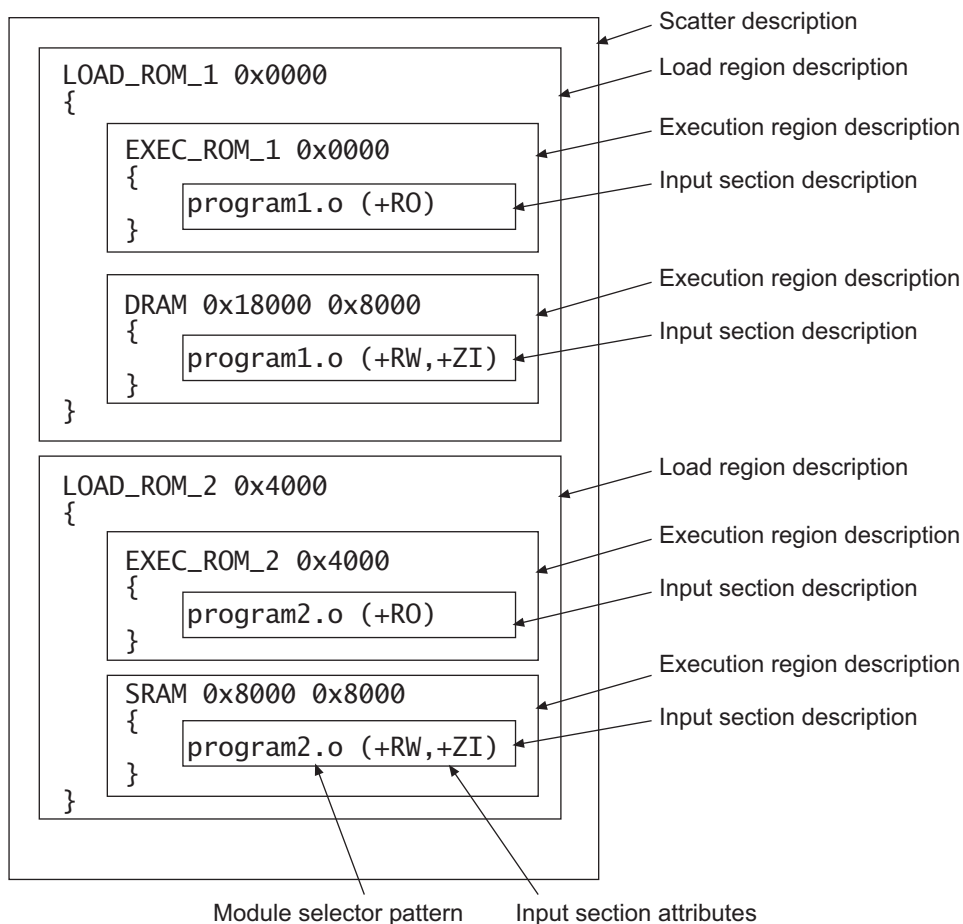Module selector pattern   Input section attributes

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

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).

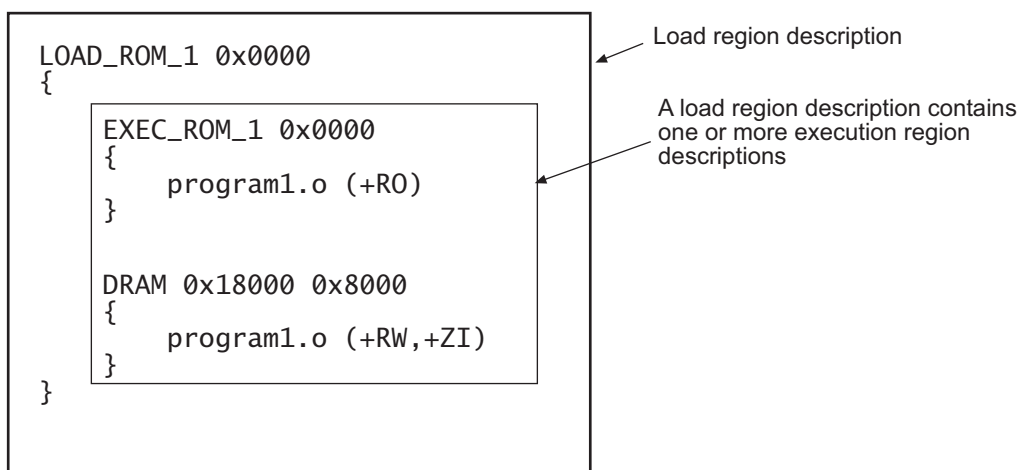The components of a typical load region description are shown in Figure 5-6.

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

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

Load region description

A load region description contains one or more execution region descriptions

Figure 5-6 Components of a load region description

The syntax, in BNF, is:

load_region_description ::=

  load_region_name (*base_address* | ("+" *offset*)) [*attribute_list*] [*max_size*]
    "{"
      execution_region_description+
    "}"

where:

load_region_name   This names the load region. Only the first 31 characters are significant. The name is only used to identify each region. Unlike the *exec_region_name* (page 5-13), the *load_region_name* 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   Is the address where objects in the region are to be linked. *base_address* must be a word-aligned number.

+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:

PI   Position-independent

RELOC   Relocatable

OVERLAY   Overlaid

ABSOLUTE   Absolute address

You can specify only one of these attributes. 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 ADS. To use multiple execution regions at the same address, you must provide your own overlay manager.

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 5-13.

       ARM DUI 0151A

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).

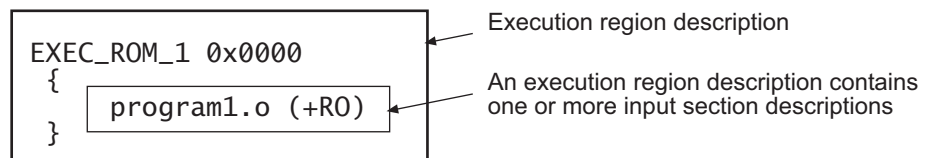The components of a typical execution region description are shown in Figure 5-7.

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

Execution region description

An execution region description contains one or more input section descriptions

Figure 5-7 Components of an 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]
      "{"
            input_section_description+
      "}"
```

where:

exec_region_name   This names the execution region. (Only the first 31 characters are significant.)

base_address   Is 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:

PI	Position-independent.
OVERLAY	Overlaid.
ABSOLUTE	Absolute address. The execution address of the region is specified by *base_designator*.
FIXED	Fixed address. Both the load address and execution address of the region is specified by *base_designator* (the region is a root region. See *Creating root execution regions* on page 5-22). *base_designator* must be either an absolute base address, or an offset of +0.
UNINIT	Uninitialized data.

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 *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.

The attribute RELOC cannot be explicitly specified for execution regions. The region can only be RELOC by inheriting the attribute from a previous execution region or parent.

It is not possible for an execution region that uses the *+offset* form of *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.

UNINIT specifies that the ZI output section, if any, in the execution region will not be initialized to zero. Use this to create execution regions containing uninitialized data or memory-mapped I/O.

max_size   This is an optional number that instructs armlink to generate an
                           error if the region has more than *max_size* bytes allocated to it.

input_section_description

                           This specifies the content of the input sections. See *Input section
                           description*.

5.2.5   Input section description

An input_section description is a pattern that identifies input sections by:

*   Module name (object file name, library member name, or library file name). The
    module name can use wildcard characters.

*   Input section name, or input section attributes such as READ-ONLY, or CODE.

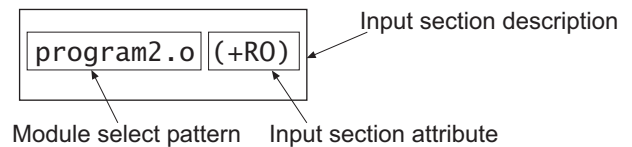The components of a typical input section description are shown in Figure 5-8.

Figure 5-8 Components of an input section description

The syntax, in BNF, is:

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 the *module_selector_pattern* matches one of the following:

- The name of the object file containing the section.

- The name of the library member (without leading pathname).

- The full name of the library (including pathname) 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`.

The special module-selector pattern `.ANY` allows 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 the `module_selector_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 compilers, 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 compilers 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 allows 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, armlink faults the scatter-loading description. Each section is selected by a `module_selector_pattern` and an `input_section_selector`.

Examples of `module_selector_pattern` specifications are:

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

Examples of `input_section_selector` specifications are:

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

                                       ARM DUI 0151A

———— **Note** ————

The compilers produce areas that can be identified by input section patterns such as `.text`, `.data`, `.constdata`, and `.bss`. These names, however, might change in future versions and you should avoid using them.

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

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

— use `#pragma arm section` to specify the name of the section containing the code or data of interest. See the section on pragmas in the *ADS Compilers and Libraries Guide*.

———————————————

The following variables are used to describe multiple matches:

- *m1* and *m2* represent module-selector-patterns

- *s1* and *s2* represent input-section-selectors.

In the case of multiple matches, 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 *s1* and *s2* are both patterns matching section names, the same definition as for module selector patterns is used.

    - If one of *s1*, *s2* matches the input section name and the other matches the input section attributes, *s1* and *s2* are unordered and the description is diagnosed as faulty.

    - If both *s1* and *s2* match input section attributes, the determination of whether *s1* is more specific than *s2* is defined by the relationships below:

ENTRY is more specific than RO-CODE, RO-DATA, RW-CODE or RW-DATA

RO-CODE is more specific than RO

RO-DATA is more specific than RO

RW-CODE is more specific than RW

RW-DATA is more specific than RW.

There are no other members of the (*s1* more specific than *s2*) relationship between section attributes.

3. For the `module_selector_pattern, input_section_selector` pair, *m1,s1* is more specific than *m2,s2* only if any of the following are true:

- *s1* is a literal input section name (that is, it contains no pattern characters) and *s2* matches input section attributes other than +ENTRY

- *m1* is more specific than *m2*

- *s1* is more specific than *s2*.

This matching strategy has the following consequences:

- Descriptions do not depend on the order they are written in the file.

- Generally, the more specific the description of an object, the more specific the description of the input sections it contains.

- The `input_section_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 Pattern matching

```
LR_1 0x040000
{
    ER_ROM 0x040000               ; The startup exec region address is the same
    {                             ; as the load address.
        application.o (+ENTRY)    ; The section containing the entry point from
    }                             ; the object is placed here.
    ER_RAM1 0x048000
    {
        application.o (+RO_CODE)  ; Other RO code from the object goes here
    }
    ER_RAM2 0x050000
    {
        application.o (+RO)       ; 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 section on writing code for ROM in the *ADS Developer Guide*.

5.3.1 Selecting veneer input sections in scatter-loading descriptions

Veneers are used to switch between ARM and Thumb code or to perform a longer program jump than can be specified in a single instruction (see *Veneer generation* on page 3-13). Use a scatter-loading 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.

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 that executes at its load 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 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 5-1.

- Use the FIXED execution region attribute to ensure that the load address and execution address of a specific region are the same. See Example 5-9 on page 5-24 and Figure 5-10 on page 5-24.

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 5-24 for more information.

Example 5-1 Specifying the same load and execution address

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

Example 5-9 Using the FIXED attribute

```
LR_1 0x040000          ; load region starts at 0x40000
{                      ; start of execution region descriptions
    ER_RO 0x040000     ; load address = execution address
    {
        *(+RO)         ; RO sections other than those in init.o
    }
    ER_INIT 0x080000 FIXED ; load address and execution address of this
                           ; execution region are fixed at 0x80000
    {
        init.o(+RO)    ; all RO sections from init.o
    }
    ; rest of scatter description...
```
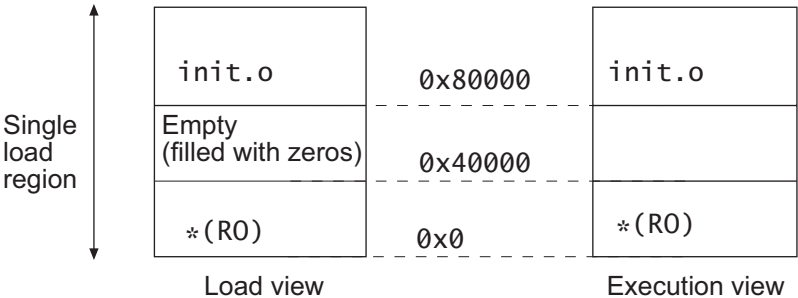

Figure 5-10 Memory Map for fixed execution regions

5.3.3   Placing regions at fixed addresses

You can use the FIXED attribute to an execution region scatter-loading description 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 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 a ROM and a checksum at the end of ROM, some of the memory contents might be unused. Use the * or .ANY module selector to flood fill the region between the end of the initialization block and the start of the data block.

——— **Note** ———

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

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

Placing functions and data at a 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 fuction 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, either:

* Place the function or data item in its own source file

  ——— **Note** ———

  Specifying one object for each function or data block limits the ability of the compiler to perform optimizations.

* Use the `-zo` compiler option to produce an object file for each function (see the *ADS Compilers and Libraries Guide*)

* Use `#pragma arm section` inside the C or C++ source code to create multiple named sections (see Example 5-2 on page 5-27).

* Use `AREA` directive from assembly language. For assembly code, the smallest locatable unit is an `AREA` (see the *ADS Assembler Guide*).

Placing the contents of individual object files

The scatter-loading description file in Example 5-1 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 5-1 Section placement

```
LOADREG1 0x0 0x10000
{
    EXECREG1 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
    {
        * (+RW +ZI)
    }
    DATABLOCK 0x1ff00 0xff FIXED  ; execution region fixed at 0x1ff00
    {                       ; the maximum space available for table is 0xff
        data.o(+RO_DATA)    ; place read-only data between 0x1ff00 and 0x1ffff
    }
}
```

——— **Note** ———

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

- If your loader can handle multiple load regions, place the RO code or data in its own load region.

- If you do not require the function or data to be at a fixed location in ROM, use ABSOLUTE instead of FIXED. The loader 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 does not zero-initialize the memory locations.) For more details, see the section on writing code for ROM chapter of the *ADS 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 5-2.

Example 5-2 Naming a section

```
// 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 5-3). If both code and data sections have the same name, the code section is placed first.

Example 5-3 Placing a section

```
FLASH 0x24000000 0x4000000
{
    FLASH 0x24000000 0x4000000
    {
        init.o (Init, +First)        ; place code 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()
    }
}
```

5.4 Equivalent scatter-loading descriptions for simple images

The command-line options (`-ro-base`, `-rw-base`, `-reloc`, `-split`, `-ropi`, and `-rwpi`) create the simple image types described in *Using command-line options to create simple images* on page 3-14. You can create the same image types by using the `-scatter` command-line option and a file containing one of the corresponding scatter-loading descriptions.

5.4.1 Type 1

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.

Not ropi

`-ro-base` *address* specifies the load and execution address of the region containing the RO output section. Example 5-4 shows the scatter-loading description equivalent to using `-ro-base 0x040000`.

Example 5-4 Single load region and contiguous execution regions

```
LR_1 0x040000      ; Define the load region name as LR_1, the region starts at 0x040000.
{                  ; Start of execution region descriptions.
    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 region are placed consecutively here.
    }
}
```

This description 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. 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 allows an execution region to be placed immediately following the end of the preceding execution region.

ropi

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.

-ro-base *address* is used to specify the load and execution address of the region containing the RO output section.

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

Example 5-5 Position independent code

```
LR_1 0x010000 PI        ; The first load region is at 0x010000.
{
    ER_RO +0            ; The PI attribute is inherited from parent.
                        ; The default execution address is 0x010000, but the code can be moved.
    {
        *(+RO)          ; All the RO sections go here.
    }
    ER_RW +0 ABSOLUTE   ; PI attribute is overridden by ABSOLUTE.
    {
        *(+RW)          ; The RW sections are placed next. They cannot be moved.
    }
    ER_ZI +0            ; ER_ZI region placed after ER_RW region.
    {
        *(+ZI)          ; All the ZI sections are placed consecutively here.
    }
}
```

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.

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.

Not rwpi

-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 5-6 shows the scatter-loading description equivalent to using -ro-base 0x010000 -rw-base 0x040000.

Example 5-6 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 execution address of ER_RO is 0x010000.

The ER_RW execution region is not contiguous with ER_RO. Its execution address is 0x040000.

The `ER_ZI` execution region is placed immediately following the end of the preceding execution region, `ER_RW`.

rwpi

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.

As with other type 2 images, -ro-base *address* is used to specify the load and execution address of the region containing the RO output section and -rw-base *address2* is used to specify the load and execution address for the region containing the RW output section.

Example 5-7 shows the scatter-loading description equivalent to using
-ro-base 0x010000 -rw-base 0x018000 -rwpi.

Example 5-7 Position independent data

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

`ER_RO`, the RO execution region, inherits the `ABSOLUTE` attribute from the load region `LR_1`. The next execution region, `ER_RW`, is marked as `PI`.

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. It is similar to images of type 2 except that the single load region in type two is now split into two load regions

Non-relocatable load regions

`-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` is used to split 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 5-8 shows the scatter-loading description equivalent to using `-ro-base 0x010000 -rw-base 0x040000 -split`.

Example 5-8 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

This type 3 images 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.

-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 is used to split 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 5-11 on page 5-34 shows the scatter-loading description equivalent to using -ro-base 0x010000 -rw-base 0x040000 -reloc -split.

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 default execution address of ER_RO is 0x010000.

The ER_RW execution region is not contiguous with ER_RO. Its default execution address is 0x040000.

The ER_ZI execution region is placed immediately following the end of the preceding execution region, ER_RW.

Example 5-11 Relocatable load regions

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

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

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

 ARM DUI 0151A

Chapter 6
Creating and Using libraries

This chapter describes the use of libraries with armlink. 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.

                             ARM DUI 0151A

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, however, unused sections are subsequently eliminated unless -noremove is used.

- 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** ——

    Members selected from libraries using patterns are added to the image unconditionally, as if they were objects.

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.

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 were an ordinary object.

armlink creates a list of libraries as follows:

1.   armlink adds any libraries specified in the input file list to the list.
2.   The user-specified search path is examined by armlink to identify library directories for library requests embedded in the input objects. See *Searching for ARM libraries* on page 6-4 for details on the search process.

The best-suited library variants are chosen from the searched directories and their subdirectories. ARM-supplied libraries have multiple variants that are named according to the attributes of their members. For details on the library variants see the *ADS Compilers and Libraries Guide* and *Selecting ARM library variants* on page 6-4.

When armlink has constructed the list of libraries, it repeatedly scans each library in the list to resolve references. See *Scanning the libraries* on page 6-5 for details.

6.2.1 Searching for user libraries

For user libraries explicitly included on the command line, a path is required if they are not in the current working directory.

——— **Note** ———

The search paths used for the ARM standard libraries are *not* searched for user libraries.

6.2.2 Searching for ARM libraries

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

* 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 ARMLIB variable holds the path to the ARM library parent directory.

  ——— **Caution** ———

  -libpath overrides the paths specified by the ARMLIB variable.

* Using the CodeWarrior IDE linker configuration panel (see the *CodeWarrior IDE Guide*).

* Using the environment variable ARMLIB.

armlink combines each parent directory given by either -libpath, the configuration panel, or the ARMLIB 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*.

6.2.3 Selecting ARM library variants

From each of the directories selected by armlink when searching for ARM libraries, armlink must select the best-suited library. 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 name of the library. See the section on library naming conventions in the *ADS Compilers and Libraries Guide*.

armlink accumulates the attributes of each input object and uses them to select the library variant best suited to the accumulated attributes. If more than one of the selected libraries are equally suited, the library selected first is retained and the others are rejected.

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

6.2.4 Scanning the libraries

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 in 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 are incapable of being 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 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 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, refer to Chapter 2 *The armlink Command Syntax*.

6.3.1   Librarian command-line options

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]
[-via option_file] [-x] [-zs] [-zt] library [file_list]
```

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

```
armar [-help] [-create] [-c] [-d] [-m] [-q] [-r] [-u] [-vsn] [-v]
[-via option_file] [{[-a]|[-b]|[-i]} {pos_name}] library [file_list]
```

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*.

-create   This option creates a new library even if *library* already exists.

-c   This option suppresses the diagnostic message normally written to standard error when a library is created.

-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 file names from replacing files with the same prefix.

-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 a path name of the library file.
-m	This option moves files. If -a, -b, or -i with *pos_name* is specified, move files 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 standard output.
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*. An example of the output is shown below:

```
Code    RO Data    RW data    ZI Data    Debug    Object Name
 464          0          0          0     8612    app_1.o
3356          0          0      10244    11848    app_2.o
3820          0          0      10244    20460    TOTAL
```

-T   This option allows file name 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 the *ADS Compilers 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 standard output.

            -t   Includes a long listing of information about the files within the library.

            -x   Prints the filename preceding each extraction.

-vsn   This option prints the version number on standard error.

-x   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.

-zs   This option shows the symbol table.

-zt   Lists member sizes and entry points in *library*. See -sizes and -entries
            for output format.

———— **Note** ————

The options -a, -b, -C, -i, -m, -T, -u, and -v are not required for normal operation.

Options relating to library order (for example, -a, -b, -i, and -m) are not relevant, since
the ARM tool chain 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 (-C and -u) are unlikely to be used because, in
practice, the libraries are rebuilt rather than updated.

6.3.2   Examples

Syntax examples are shown in Example 6-1 to Example 6-7 on page 6-10.

Example 6-1 Create a new library and add all object files

```
armar -create mylib *.o
```

Example 6-2 List the table of contents

```
armar -t mylib
```

Example 6-3 List the symbol table

```
armar -zs mylib
```

Example 6-4 Add (or replace) files

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

Example 6-5 Extract a group of files

```
armar -x my_lib k*.o
```

Example 6-6 Delete a group of files

```
armar -d my_lib sys_*
```

Example 6-7 Merge libraries and add (or replace) files

```
armar -r my_lib.a obj1.o my_lib.a other_lib.a obj2.o obj3.o
```

Chapter 7
Using fromELF

This chapter describes the fromELF software utility provided with ADS. It contains the following sections:

- *About fromELF* on page 7-2
- *fromELF command-line options* 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 it to display various information about an ELF object or to generate text files containing the information.

fromELF 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).

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

——— **Note** ———

Do not link your images with the -nodebug linker option if you require a -fieldoffsets fromELF step. 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

fromELF 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.

 ARM DUI 0151A

7.2 fromELF command-line options

The fromELF command syntax is as follows:

```
fromelf [-help] [-fieldoffsets|[[-select select_options]]] [-nolinkview]
[-nodebug] [-vsn] [[text_output_format]| code_output_format] [-base n]
[memory_config] [-output output_file] {input_file}
```

where:

-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.

-fieldoffsets
: This option produces, to standard output, a list of assembly language EQU directives that equate C++ class or C structure field names to their offsets from the base of the class or structure. The input ELF file can be a relocatable object or an image.

    Use -o to redirect the output to a file. Use the INCLUDE command from armasm to load the produced file and provide access to C++ classes and C structure members by name from assembly language. See the *ADS Assembler Guide* for more information on armasm.

    ——— **Note** ———

    If the source file does not have debug information, this option is not available. You cannot use this option together with a code_output_format.

    ————————————————

    -fieldoffsets 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_option 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 , 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_option` string. This is the default.

- Specify the fields to exclude by prefixing a ~ to the `select_option` string.

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

-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.

——— **Caution** ———

This option can have unexpected effects if `-elf` is not specified on the command line. See Example 7-2 on page 7-5 for an example of correct usage.

-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.

                  ———— **Caution** ————

                  This option can have unexpected effects if -elf is not specified on
                  the command line. The options in Example 7-1 produce a text file
                  because no output format has been specified.

Example 7-1 Text output

```
fromelf -nodebug image -o image_nodb.asf
```

                  To get ELF format output use the options shown in Example 7-2

Example 7-2 ELF output

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

-vsn   This option displays fromELF version information.

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.

c   Disassembles code

d   Prints contents of the data sections

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

z   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
- multiple characters following a slash character, -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.

 -ihf   Extended Intellec Hex format. You can specify the base
             address of the output with the -base option. (This
             option is deprecated and will be removed from future
             versions of the product.)

 -m32   Motorola 32-bit format (32-bit S-records). You can
             specify the base address of the output with the -base
             option.

 -i32   Intel Hex-32 format. 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 HDL simulators. You can split output from
             this option into multiple files with the *memory_config*
             option.

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

If you use fromELF to convert an ELF image containing multiple
load regions to a binary format using any of the -bin, -ihf, -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.

ELF images contain multiple load regions if, for example, they are
built with a scatter-load description file that defines more than one
load region.

-base *n*   This option specifies the base address of the output for Motorola
                     S-record, Intel Hex, and Extended Intellec Hex file formats. This
                     option is available only if -m32, -i32, or -ihf is specified as the
                     output format.

                     You can specify the base address as either:
                     • a decimal value, for example -base 0
                     • as a hexadecimal value, for example -base 0x8000.

All addresses encoded in the output file start at the base address *n*. If you do not specify a -base option, the base address is taken from the load region address.

——— **Note** ———

If multiple load regions are present, the -base value is used for each output file. That is, it overrides all load region addresses.

memory_config   This option outputs multiple files for multiple memory banks. This option is available only if -vhx or -bin is specified as the output format.

The format of *memory_config* is -*width*x*banks* 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 this option is selected for any format other than -bin or -vhx it is ignored.

If the image has one load region, fromELF generates *bank* files with the following naming conventions:

- If there is one memory bank (*banks* is 1) the output file is named by the -o *output_file* argument.

- If there are multiple memory banks (bank>1), fromELF generates *banks* number of files starting with *output_file0* and finishing with *output_file bank*-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
...
```

-**o**utput *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.

fromELF accepts only ARM-executable ELF files and ARM object ELF files (.o). If *input_file* is a scatter-loaded image that contains more than one load region and the output format is either -bin, -ihf, -m32, -i32, or -vhx, fromELF creates a separate file for each load region.

Copyright © 2001 ARM Limited. All rights reserved.

7.3   Examples of fromELF usage

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

───── **Note** ─────

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

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

─────────────

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 structure 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
```

Chapter 8
Using the ARM Profiler

This chapter describes the ARM Profiler utility provided with ADS. It contains the following sections:

- *About the ARM profiler* on page 8-2
- *Profiler command-line options* on page 8-3
- *Sample output* on page 8-4.

8.1   About the ARM profiler

The ARM profiler, armprof, displays an execution profile of a program from a profile data file generated by an ARM debugger.

——— **Note** ———

The ARM profiler is useful only with simple images compiled to the base ATPCS standard. It does not provide useful information if, for example, you are using scatter-loading.

The profiler displays one of two types of execution profile depending on the amount of information present in the profile data:

•   If only pc-sampling information is present, the profiler can display only a flat profile giving the percentage time spent in each function itself, excluding the time spent in any of its children.

•   If function call count information is present, the profiler can display a *callgraph* profile that shows approximations of the time spent in each function, the time accounted for by calls to all children of each function, and the time allocated to calls from different parents.

                   ARM DUI 0151A

8.2 Profiler command-line options

A number of options are available to control the format and amount of detail present in the profiler output. The command syntax is as follows:

```
armprof [[-parent]|[-noparent]] [[-child]|[-nochild]] [-sort options] prf_file
```

where:

-parent   This option displays information about the parents of each function in the profile listing. This gives approximate information about how much time is spent in each function servicing calls from each of its parents.

-noparent   This option turns off the parent listing.

-child   This option displays information about the children of each function. The profiler displays the approximate amount of time spent by each child performing services on behalf of the parent.

-nochild   This option turns off the child listing.

-sort *options*

This option sorts the profile information using one of the following options:

cumulative   Sorts the output by the total time spent in each function and all its children.

self   Sorts the output by the time spent in each function (excluding the time spent in its children).

descendants   Sorts the output by the time spent in all the children of a function but excluding time spent in the function itself.

calls   Sorts the output by the number of calls to each function in the listing.

prf_file   This option specifies the file containing the profile information.

By default, child functions are listed, but not parent functions, and the output is sorted by cumulative time.

8.2.1 Example

```
armprof -parent sort.prf
```

8.3   Sample output

The profiler output is split into a number of sections, each section separated by a line. Each section gives information on a single function. In a flat profile, one with no parent or child function information, each section is a single line.

The following shows sample sections for functions called `insert_sort` and `strcmp`.

```
Name                  cum%     self%   desc%     calls
-------------------------------------------------------
 main                          17.69%  60.06%        1
insert_sort           77.76%   17.69%  60.06%        1
 strcmp                        60.06%   0.00%   243432
-------------------------------------------------------
 qs_string_compare             3.21%   0.00%    13021
 shell_sort                    3.46%   0.00%    14059
 insert_sort                  60.06%   0.00%   243432
strcmp                66.75%   66.75%   0.00%   270512
-------------------------------------------------------
```

Glossary

ADS	See *ARM Developer Suite*.
Angel	Angel is a debug monitor program that enables you to develop and debug applications running on ARM-based hardware. Angel can debug applications running in either ARM state or Thumb state.
ANSI	American National Standards Institute. An organization that specifies standards for, among other things, computer software.
API	Application Program Interface.
Architecture	The term used to identify a group of processors that have similar characteristics.
ARM Developer Suite	A suite of applications, together with supporting documentation and examples, that enable you to write and debug applications for the ARM family of RISC processors.
ARMulator	ARMulator is an instruction set simulator. It is a collection of modules that simulate the instruction sets and architecture of various ARM processors.
ATPCS	ARM and Thumb Procedure Call Standard defines how registers and the stack will be used for subroutine calls.
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.

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	Executable and linking format.
Environment	The actual hardware and operating system that an application will run on.
Executable and linking format	
	The industry standard binary file format used by the ARM Developer Suite. 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 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.
Heap	The portion of computer memory that can be used for creating new variables.
Host	A computer which provides data and other services to another computer.
IDE	Integrated Development Environment (for example, the CodeWarrior IDE).
Image	An executable file which 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.
	See also Output sections.
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 which 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.
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.
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.
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.
Region	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. *See also* Root region.
Remapping	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.
Retargeting	The process of moving code designed for one execution environment to a new execution environment.
Root region	In an image, a region that executes at its load address. A non-root region is a region that must be copied from its load address to its execution address.
ROPI	Read Only Position Independent. Code and read-only data addresses can be changed at run-time.

RTOS	Real Time Operating System.
RWPI	Read Write Position Independent. Read/write data addresses can be changed at run-time.
Scatter-loading	Assigning the address and grouping of code and data sections individually rather than using single large blocks.
Sections	A block of software code or data for an Image. *See also* Input sections.
Semihosting	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.
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	Software Interrupt. An instruction that causes the processor to call a programer-specified subroutine. Used by ARM to handle semihosting.
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.
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.
Volatile	Memory addresses where the contents can change independently of the executing application are described as volatile. These are typically memory-mapped peripherals. *See also* Memory mapped
VFP	Vector Floating Point. A standard for floating-point coprocessors where several data values can be processed by a single instruction.
Word	A 32-bit unit of information. Contents are taken as being an unsigned integer unless otherwise stated.
ZI	Zero Initialized. R/W memory used to hold variables that do not have an initial value. The memory is normally set to zero on reset.

 ARM DUI 0151A

Index

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.

Symbols