Application Note 48

Scatter Loading

Document number: ARM DAI 0048A

Issued: January 1998

ENGLAND

Advanced RISC Machines Limited
Fulbourn Road
Cherry Hinton
Cambridge CB1 4JN
UK
Telephone: +44 1223 400400
Facsimile: +44 1223 400410
Email: info@arm.com

GERMANY

Advanced RISC Machines Limited
Otto-Hahn Str. 13b
85521 Ottobrunn-Riemerling
Munich
Germany
Telephone: +49 89 608 75545
Facsimile: +49 89 608 75599
Email: info@arm.com

JAPAN

Advanced RISC Machines K.K.
KSP West Bldg, 3F 300D, 3-2-1 Sakado
Takatsu-ku, Kawasaki-shi
Kanagawa
213 Japan
Telephone: +81 44 850 1301
Facsimile: +81 44 850 1308
Email: info@arm.com

USA

ARM USA Incorporated
Suite 5
985 University Avenue
Los Gatos
CA 95030 USA
Telephone: +1 408 399 5199
Facsimile: +1 408 399 8854
Email: info@arm.com

World Wide Web address: http://www.arm.com

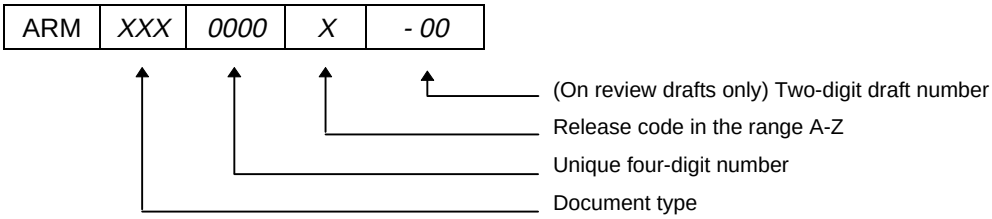
ARM
Advanced RISC Machines

Proprietary Notice

Key

Document Number

This document has a number which identifies it uniquely. The number is displayed on the front page and at the foot of each subsequent page.



Document Status

The document's status is displayed in a banner at the bottom of each page. This describes the document's confidentiality and its information status.

Confidentiality status is one of:

ARM Confidential	Distributable to ARM staff and NDA signatories only
Named Partner Confidential	Distributable to the above and to the staff of named partner companies only
Partner Confidential	Distributable within ARM and to staff of all partner companies
Open Access	No restriction on distribution

Information status is one of:

Advance	Information on a potential product
Preliminary	Current information on a product under development
Final	Complete information on a developed product

Change Log

Issue	Date	By	Change
A	January 1998	SKW	Released

Table of Contents

1 Introduction **2**

2 Improvements Made to Scatter Loading in SDT 2.11 **4**

3 Load Regions and Execution Regions **5**
3.1 Definitions 5

4 Placing Execution Regions with -RO and -RW Options **6**
4.2 Example 7

5 Placing Regions with Scatter Loading **10**
5.1 Command line options 10
5.2 Image formats 10
5.3 Linker pre-defined symbols 11
5.4 Area ordering 12
5.5 Scatter loading and long distance branching 13
5.6 The description file 14

6 Scatter Loading Examples **16**
6.1 Example 1 16
6.2 Example 2 17
6.3 Example 3 19

7 Initialization Code for Scatter Loading **21**

8 The Description File Format **22**

1 Introduction

Scatter loading is a mechanism provided by the ARM Linker, which enables you to partition an executable image into regions that can be positioned independently in memory.

In a simple embedded computer system, memory is divided into ROM and RAM. The image produced by the linker is divided into the “Read-Only” segment, which contains the code and read-only data, and the “Read-Write” segment, which contains the initialized and non-initialized or zero-initialized (ZI) data. Usually, the “Read-Only” segment is placed in ROM and the “Read-Write” segment is copied from ROM to RAM before execution begins.

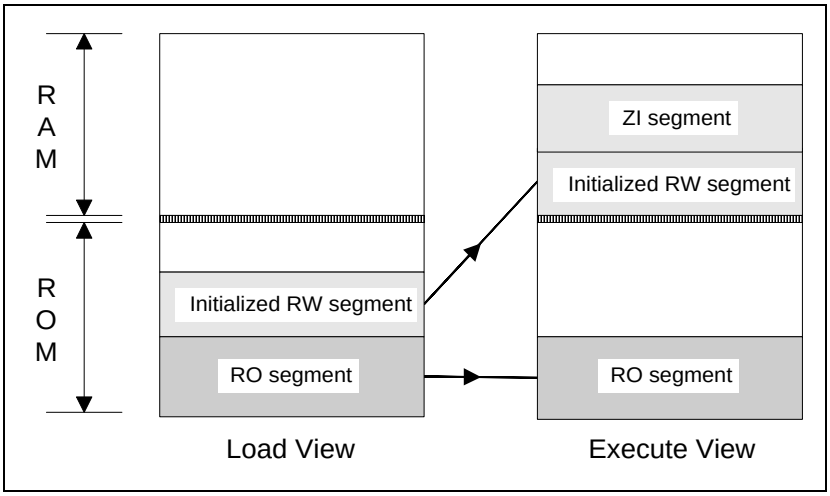


Figure 1: A simple scatter loaded memory map

Embedded systems often use a more complex memory map, which can consist of ROM, SRAM, DRAM, FLASH and so on. The scatter loading mechanism lets you place various parts of the image in these distinct memory areas.

Scatter loading enables you to partition your program image into several regions of code and data which can be placed separately in the memory map. Each region is placed in a contiguous chunk of memory space. The location of a region can differ between load time and execution time, with the application copying code and data from its load address to its execution address.

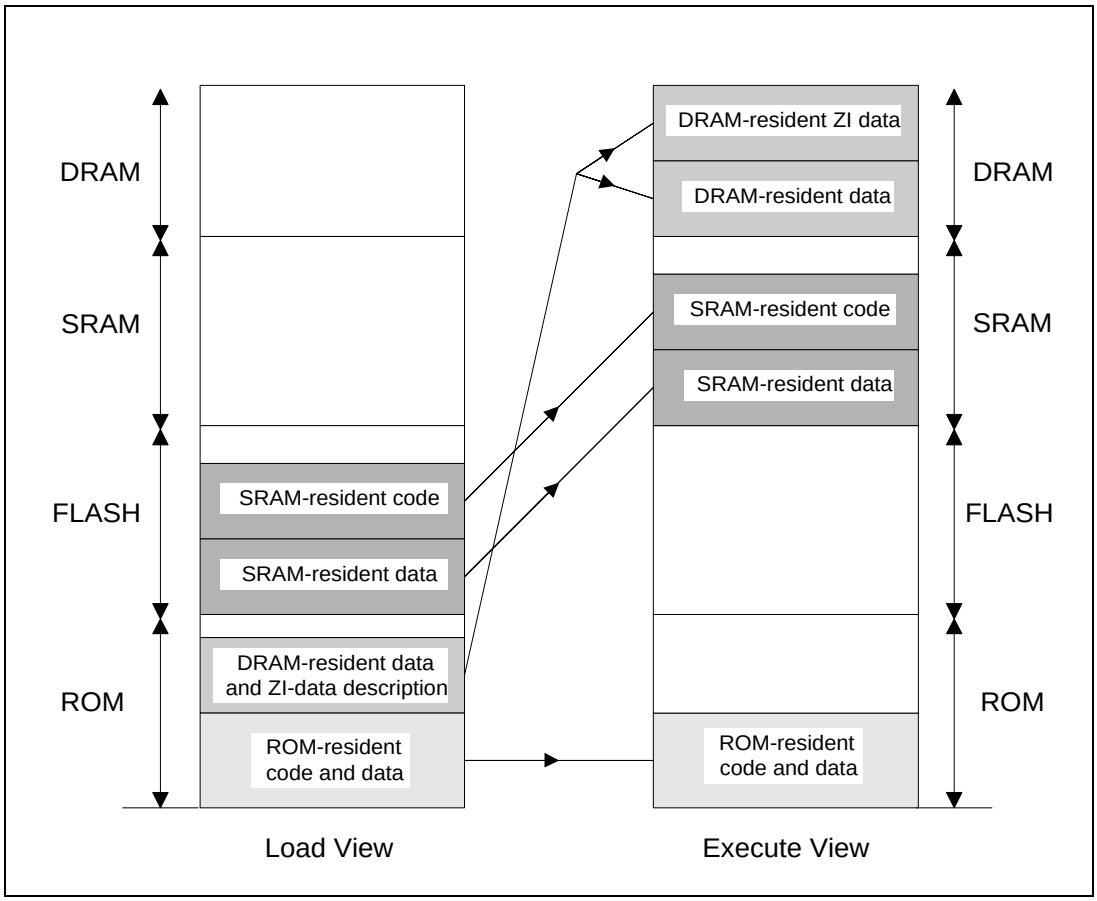


Figure 2: A more complicated scatter loaded memory map

The placement information is contained in a description file, the name of which is passed as a command line parameter to the linker.

2 Improvements Made to Scatter Loading in SDT 2.11

The original ARM scatter loading mechanism was introduced in SDT 2.0. The mechanism remained relatively unchanged in SDT 2.10. However, SDT 2.11 introduced major improvements to the scheme. This section details these improvements for users who are already familiar with the previous scatter loading implementation.

The main improvements from the user's point of view are:

- You no longer need special `ROOT` and `ROOT-DATA` regions. These were previously used to contain code and data not specified in any other area. However, they complicated the writing of description files for many users.
- You guide the assignment of areas to execution regions by writing patterns. Previously, a multiple match could not be diagnosed and the pattern presented most recently in the text would prevail. Now the most specific match is chosen if there is one, otherwise the description is diagnosed faulty.
- You have precise diagnostics: the linker identifies the line and column number of faults in a description file.
- You can now mark an area "first" or "last" in each execution region, though it must still meet the requirement that, within each execution region, `RO AREAs` must precede the `RW AREAs` which must precede the `ZI AREAs`.
- New *+n* notation for the base address of an execution region allows you to place an execution region "n" bytes after the previous one. If it is used for the first execution region in the load region, the region is placed "n" bytes after the load address. See ***6.3 Example 3*** on page 19.
- More linker-generated areas have been made assignable to an execution region of your choice. This is particularly useful for placing the ARM/Thumb interworking veneer area called `IWV$$Code`. See ***6.3 Example 3*** on page 19.
- The ARM ELF Image format is now fully supported.

3 Load Regions and Execution Regions

A program image consists of regions which may occupy different locations at load time and execution time.

This means that just before an image is executed, there are some regions which need to be moved from the locations at which they were initially loaded in memory. For example, initialized read-write data may reside in ROM, but it must be copied into RAM when the program starts executing.

There are two mechanisms available to describe where image regions should be placed in memory at execution time:

- Using `-RO` and `-RW` command line options, to specify the execution addresses of read-only and read-write regions. This is the simple method, used in systems which have a simple memory map of ROM and RAM, and where the image itself has one load region and two execution regions. See ***4 Placing Execution Regions with -RO and -RW Options*** on page 6 for more information.
- Scatter loading, which is the preferred method for more complex memory maps and for images which have more than two execution regions. See ***5 Placing Regions with Scatter Loading*** on page 10 for more information.

3.1 Definitions

Load region

The memory which is occupied by a program before it starts executing, but after it has been loaded into memory, can be split into a set of disjoint load regions, each of which is a contiguous chunk of bytes.

Execution region

The memory used by a program while it is executing can also be split into a set of disjoint execution regions.

Note *A load region contains one or more execution regions.*

Note *Each execution region belongs to only one load region.*

4 Placing Execution Regions with -RO and -RW Options

In a simple image, you can specify the execution addresses at which the "Read-Only" segment and the "Read-Write" segment will be placed in the memory map by using the `-RO` *`exec-address`* and `-RW` *`exec-address`* options of the linker, where:

`-RO` *`exec-address`* instructs the linker to place the "Read-Only" segment at `exec address` (often the address of the first location in ROM)

`-RW` *`exec-address`* instructs the linker to place the "Read-Write" segment at `exec address`

Note *`-RO-base` and `-base` options are equivalent to `-RO`. `-RW-base` and `-RW-data` options are equivalent to `-RW`.*

At application load time, the `RO` region is loaded at its execution address and the `RW` region is loaded immediately after the `RO` region.

The read-write (data) segment may contain code, as programs sometimes modify themselves (or better, generate code and execute it). Similarly, the read-only (code) area may contain read-only data (for example string literals, floating-point constants, ANSI C `const` data).

Using the addresses passed to it, the linker generates the symbols required to allow the region to be copied from its load address to its execution addresses. These symbols describe the execution address and the limit of each region. They are defined independently of any input files and, along with all other external names containing `$$`, are reserved by ARM.

4.1.1 Linker pre-defined symbols

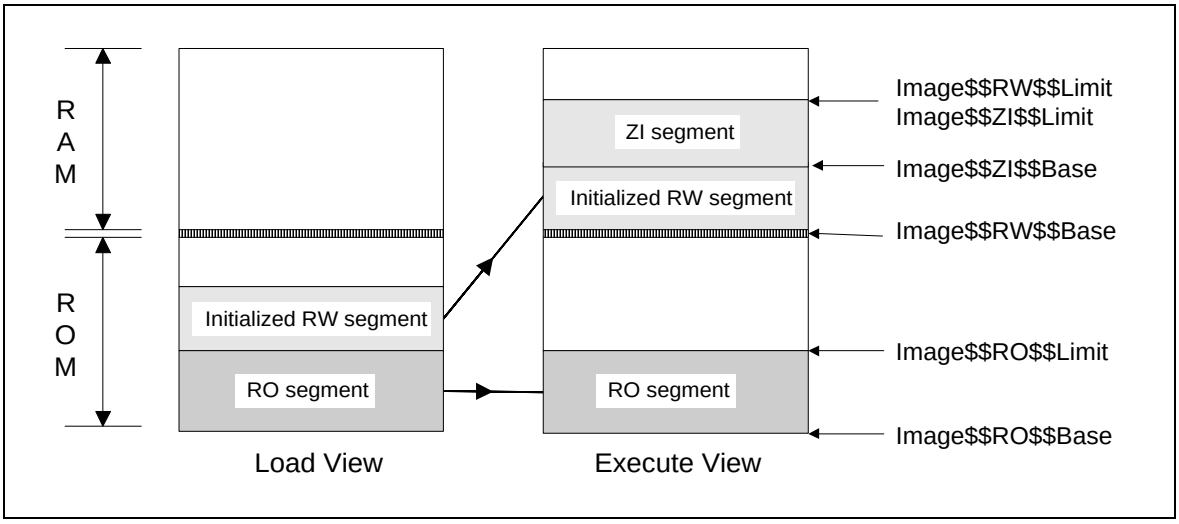


Figure 3: Linker pre-defined symbols

`Image$$RO$$Base`	is the address of the read-only execution region (usually contains code and read-only data).
`Image$$RO$$Limit`	is the address of the word immediately after the end of read-only execution region.
`Image$$RW$$Base`	is the address of the read-write execution region (usually contains data).
`Image$$RW$$Limit`	is the address of the word immediately after the end of read-write execution region.
`Image$$ZI$$Base`	is the address of the ZI execution segment (zeroed at image load or startup time).
`Image$$ZI$$Limit`	is the address of the word immediately after the end of the ZI execution segment.

These symbols can be imported and used as relocatable addresses by assembly language programs, or referred to as `extern` addresses from C (using the `-fc` compiler option which allows $ in identifiers). Image region bases and limits are often of use to programming language runtime systems.

Note *`Image$$RW$$Base` is not generally the same as `Image$$RO$$Limit`. In simple cases of `-aif` and `-bin` outputs, they might be the same, but they differ if the `-RW` option is used (or if the image uses overlays, or if it is a shared library).*

Note *`Image$$RW$$Limit` is the same as `Image$$ZI$$Limit`, and not the same as `Image$$ZI$$Base`.*

Note *For both ARM and Thumb, a word is four bytes long.*

4.2 Example

Consider the system shown in ***Figure 1: A simple scatter loaded memory map*** and ***Figure 3: Linker pre-defined symbols***. The image produced by the linker is blown into the ROM and contains both the code and a copy of the initialized data.

Assume that ROM begins at address 0x01000000 and the RAM begins at 0x8000.

The image consists of 10KB of code and 10KB of initialized data. Before the image is executed, you also need to create a 10KB zero-initialized data area. Such an area is not included in the image as it can be easily created when the image is loaded or just before it is executed.

The linker is called with the following command line options:

```
-RO 0x01000000 -RW 0x8000
```

When the image is loaded in memory, both the code and initialized data are placed together in ROM, beginning at address 0x01000000. Thus it has only one load region, starting at 0x01000000.

Before execution, however, the initialized data has to be moved to RAM at location 0x8000, and the zero-initialized data created immediately after that. The code is to remain at the load address. Thus there are two execution regions for the image, one region each in ROM and RAM.

To describe this memory mapping to the startup routine, the linker generates the following symbols and values:

Symbol	Value
`Image$$RO$$Base`	0x01000000
`Image$$RO$$Limit`	0x01002800
`Image$$RW$$Base`	0x00008000
`Image$$RW$$Limit`	0x0000D000
`Image$$ZI$$Base`	0x0000A800
`Image$$ZI$$Limit`	0x0000D000

Table 1: Linker pre-defined symbol values

Using these symbols, the following assembly code included in the startup routine of the application moves the initialized data to RAM and creates the zero-initialized data.

The same example is later described with scatter loading, though in this simple case, scatter loading does not offer any advantage. However, if you have more regions than in this example, as in ***Figure 2: A more complicated scatter loaded memory map***, you must use scatter loading.

4.2.1 Initialization code

Initialization code is included in the startup routine of an application. Generically, such initialization code copies the required regions from their load addresses to their execution addresses, and creates all the zero-initialized areas.

It uses the symbols generated by the linker to obtain information about each region. For a list of the symbols generated by the linker, refer to ***4.1.1 Linker pre-defined symbols***.

The initialization code can be written in ARM assembly language. The following is a sample initialization code that corresponds to ***4.2 Example***. Other examples have similar initialization code, only the regions to be moved or created vary.

```
; r0 contains the load address of the region
LDR    r0,   = |Image$$RO$$Limit|
; r1 contains the execution address of the region
LDR    r1,   = |Image$$RW$$Base|
; r2 contains the address of the word beyond the end of this
; execution region
LDR    r2,   = |Image$$ZI$$Base|

CMP    r0,   r1              ; check source & destination are different
BEQ    do_zi_init            ; if not, do not move this region

; copy this region from load address to execution address
BL     copy
```

```
do_zi_init
        ; r1 contains the execution address of the region
        LDR   r1, = |Image$$ZI$$Base|
        ; r2 contains the address of the word beyond the end of this
        ; execution region
        LDR   r2, = |Image$$ZI$$Limit|
        ; r3 contains the value to be used to initialize area
        MOV   r3,   #0
        BL    zi_init          ; call subroutine zi_init
```

The code initially loads the values of the `Image$$RO$$Limit`, which points to the first word of the initialized data (the initialized data immediately follows the `RO` region) and compares it to the `Image$$RW$$Base` (which was set to the required value using the `-RW` option). If the two are same, the initialized data is where it is supposed to be, and you can create the zero-initialized data by calling the subroutine `zi_init`. If not, the initialized data is copied to its execution location by calling the `copy` subroutine.

The code for the `copy` and `zi_init` subroutines is given in the next section.

4.2.2 Subroutine code

```
; copy is a subroutine which copies a region, from an address given by
; r0 to an address given  by  r1. The address of the word beyond the
; end of this region is held in r2
copy
    CMP           r1,    r2
    LDRCC         r3,    [r0],  #4
    STRCC         r3,    [r1],  #4
    BCC           copy
    MOV           PC,    LR            ; return to caller

; zi_init is a subroutine which initializes a region, starting at the
; address in r1, to a value held in r3. The address of the word beyond
; the end of this region is held in r2
zi_init
    CMP           r1,    r2
    STRCC         r3,    [r1],  #4
    BCC           zi_init
    MOV           PC,    LR            ; return to caller
```

Note *`Image$$RO$$base` and so on have no valid values when scatter loading is used. Instead, a different set of symbols are produced by the linker, described in* ***4.1.1 Linker pre-defined symbols*** on page 6*. Thus, if you move from using `-RO` and `-RW` to using scatter loading, then your initialization code must be rewritten using the appropriate symbols. For initialization code for scatter loading, refer to* ***7 Initialization Code for Scatter Loading*** on page 21*.*

5 Placing Regions with Scatter Loading

Scatter loading is a mechanism to position load regions and execution regions in their respective memory maps. The region and position descriptions are given to the linker in a description file.

5.1 Command line options

The ARM Linker generates a scatter loaded image when the option:

```
-scov description-file -scf
```

is given on its command line.

The `-scov` option accepts scatter load descriptions; `-scf` selects the respective output format.

Another way of specifying this is to use

```
-scatter description-file
```

Using scatter loading causes the linker to ignore the following options, which are irrelevant to scatter loading:

- `-RO-base`
- `-RW-base`
- `-split`
- `-NoZeroPad` (tells the linker not to pad the end of output binary with zeroes)

If a scatter loaded application requires overlays, the scatter load description file must be used to specify the overlays. The description of scatter loading with overlays can be found in the *Software Development Toolkit User Guide (ARM DUI 0040),* ***14.4 Overlays using Scatter Loading***.

5.2 Image formats

Scatter load images can be output in three formats:

BIN

Generates one file for each load region, in the directory given as the output filename. These can then be blown into ROM, Flash and so on as appropriate. The output name is treated as a directory name. Each load region is placed in a separate file in that directory, with the same name as the load region. Load region names must therefore not contain characters, or be of a length, unacceptable to the host file system.

AIF BIN

Generates a single extended AIF file suitable for loading into the debugger. A single output file containing one section per load region is produced (so-called *fat* AIF). The name of the file is given by the `-output` option.

ELF

Generates a single executable ELF file suitable for loading into the debugger. A single output file, containing one section per load region, is produced. The name of the file is given by the `-output` option.

5.3 Linker pre-defined symbols

Using the region names given in the scatter loading description file, the linker generates the symbols required to allow each region to be copied from its load address to its execution address.

Neither the linker nor the C library provide the code required to copy an execution region from its load address or create a zero-initialized region; you must do this, as the application code writer. Sample code is provided in ***7 Initialization Code for Scatter Loading*** on page 21.

The linker generates symbols which allow your routine to initialize all the execution regions that have different load and execution addresses. These symbols give the length, load address and execution address of each region.

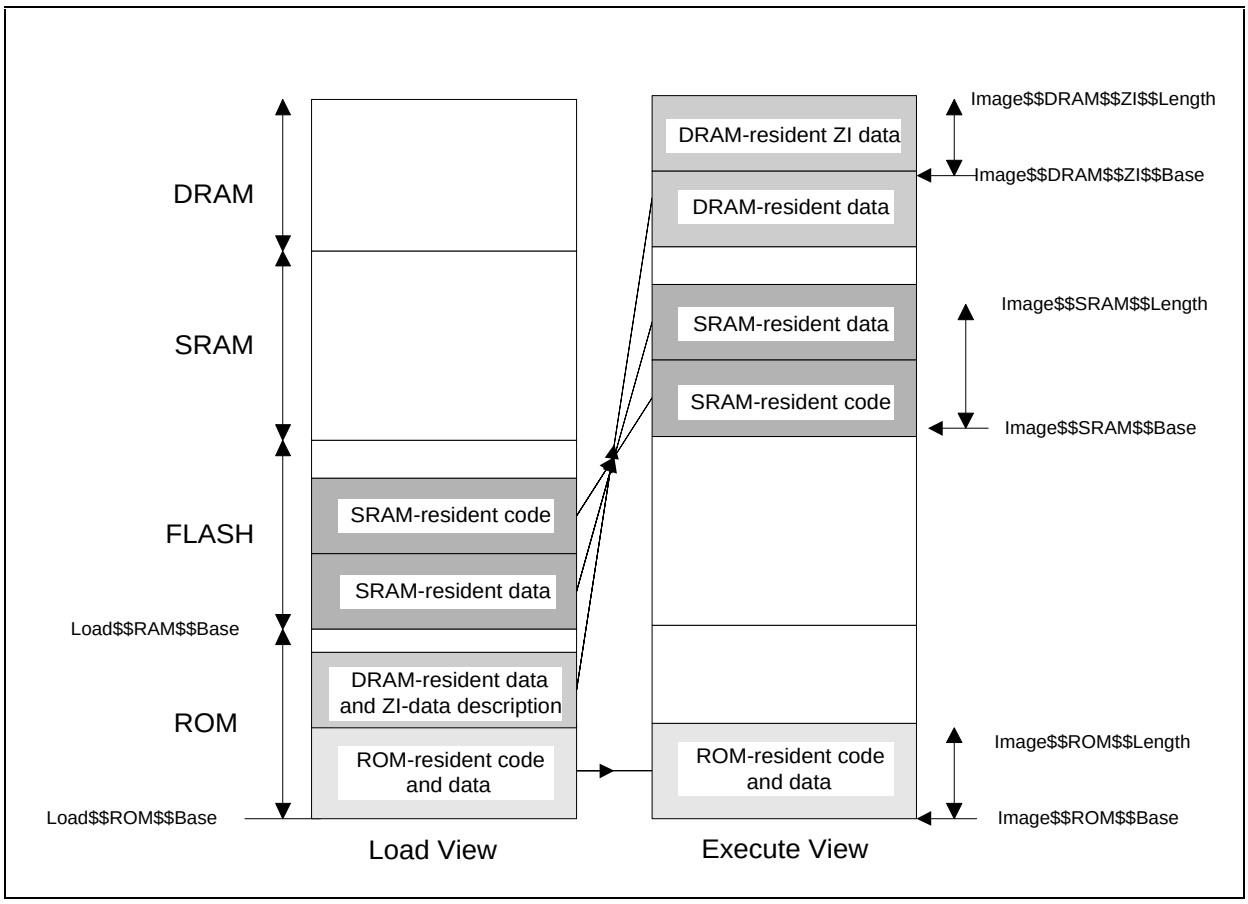


Figure 4: Linker pre-defined symbols

For RO and RW segments:

`Load$$`*`region_name`*`$$Base`	is the load address of the region
`Image$$`*`region_name`*`$$Base`	is the execution address of the region
`Image$$`*`region_name`*`$$Length`	is the execution region length in bytes (multiple of 4)

For zero-initialized segments:

`Image$$`*`region_name`*`$$ZI$$Base`	is the execution address of the region
`Image$$`*`region_name`*`$$ZI$$Length`	is the execution region length in bytes (multiple of 4)

These symbols can be imported and used by assembly language programs, or referred to as *extern* addresses from C (using the `-fc` compiler option which allows `$` in identifiers).

Note *These symbols are generated for every region named in the scatter load description.*

A scatter load image is not padded with zeros, and requires the ZI data areas to be created dynamically. This is similar to the case with a normal `-bin` file when the `-nozeropad` option is used. There is therefore no need for a load address symbol for ZI data.

The linker sorts AREAs within execution regions according to their attributes. For example, all initialized data areas are grouped together. Therefore, you can assume that all initialized data that needs copying is contiguous.

5.4 Area ordering

The linker orders AREAs within each execution region by attributes. The ordering is:

- read-only code
- read-only based data
- read-only data
- read-write code
- based data
- other initialized data
- zero-initialized data

The pseudo-attributes `FIRST` and `LAST` can be used in the description file to mark the first and last AREAs in an execution region if the placement order is important (for example, if the `ENTRY` must be first and a checksum last).

However, `FIRST` and `LAST` must not violate the basic attribute sorting order. That implies that in a region containing any read-only AREAs, the `FIRST` AREA must be a read-only area. Similarly, if the region contains any ZI data, the `LAST` AREA must be a ZI area.

5.5 Scatter loading and long distance branching

The ARM instruction set has branch instructions that allow a branch forwards or backwards up to 32Mb. A subroutine call is a variant of the standard branch; as well as allowing a branch forwards or backwards up to 32Mb, the BL (Branch with Link) instruction also preserves the return address in register 14 (Link Register, LR).

The Thumb instruction set has much shorter branch ranges, from 256 bytes in the case of conditional branches to 2048 bytes for unconditional branches. The BL instruction has a range of 4Mb.

The linker has to ensure that no branch or subroutine call violates these range restrictions. If you place your execution regions in such a way as to require inter-region branches beyond the range, the linker generates an error stating `Relocated value too big for instruction sequence`.

There are two ways to work around this restriction:

- Using function pointers in code, removing the dependence on branch ranges.
- Calling the out-of-range routines via assembler veneers.

  For example, if the application currently has a function:

  ```
  int func(int a, int b);
  ```

  which is invoked as:

  ```
  func( a, b);
  ```

  you can change this using function pointers into:

  ```
  typedef int FuncType( int, int);
  FuncType *fn = func;
  ```

  and invoke the function as:

  ```
  fn( a, b);
  ```

  If you use assembly veneers, you can write it as

  ```
  asm_func(a,b);
  ```

  where `asm_func` is an assembler routine.

  As ARM and Thumb assembly languages differ, the code for the assembler veneers is slightly different.

  The following is the assembly veneer for ARM:

  ```
      AREA arm_longbranch_veneers, CODE, READONLY
      EXPORT asm_func
      IMPORT func
  asm_func
      LDR pc, addr_func
  addr_func
      DCD func
      END
  ```

The following is the assembly veneer for Thumb:

```
    AREA thumb_longbranch_veneers, CODE, READONLY
    EXPORT asm_func
    IMPORT func
asm_func
    SUB sp,#4
    PUSH {r0}
    LDR r0, addr_func
    STR r0, [sp,#4]
    POP {r0,pc}
    ALIGN
addr_func
    DCD func
    END
```

The file containing these veneers must then be put within range of the module calling `asm_func(a,b)`.

5.6 The description file

A *Scatter Load Description* is a text file describing how the AREAs in a linked image are assigned to separate regions of memory.

In a Scatter Load Description:

- You list the separate regions of memory in which your image will execute, and specify an *execution base address* for each region.
- You describe how *execution regions* are packed into regions of physical memory (called *load regions*). The linker generates a separately loadable chunk of image for each load region. In some image formats (for example BIN), each separate load region is written to its own output file; in others (for example ELF, AIF), each load region has its own section within a single output file. You can think of each load region corresponding to a separate *persistent* memory such as ROM, EPROM, FLASH, and so on.
- Using simple patterns and attributes you describe how *armlink* should assign the constituent AREAs of your image to execution regions.

5.6.1 Notes

- *If you want to execute a code region directly then you must ensure its execution address is the same as its load address. Consequently, you must assign the AREA containing your image's ENTRY point to such a region.*
- *Your scatter load description does **not** describe the objects which make up your image. You describe that in the same way for all image types by listing the objects and libraries which the linker should use to make your image on its command line. The patterns you write in a scatter load description describe how to assign the AREAs which armlink has already selected to the execution regions you defined.*
- *You can describe execution regions which overlap, provided that you give each region the OVERLAY attribute. If you do this, the linker generates support code and data to allow overlapping regions to be swapped dynamically at execution time and adds a reference from the support code to an overlay manager. You must have included ARM's standard overlay manager, or one compatible with it, in your list of object files.*

- *ARM-Thumb interworking veneers are built in an AREA called `IWV$$Code`. You can assign this AREA to an execution region just like any other area using the AREA selector:*

  ```
  *(IWV$$Code)
  ```

  *Although there is no associated module, * still matches. Because there is only one `IWV$$Code` AREA, this selection is unambiguous.*

  A detailed description of the file format is given at the end of this Application Note.

6 Scatter Loading Examples

The following examples describe some scatter loading scenarios of increasing complexity. Each example includes a memory map, a description file and a table of linker-generated symbols. A generic initialization routine is also explained which can be used with all these scatter loading examples.

6.1 Example 1

This is the same example that is explained in ***4.2 Example*** on page 7 using the `-RO` and `-RW` options.

Memory map

The memory map has just one block of ROM (beginning at address 0x01000000), and one block of RAM (beginning at address 0x00008000). The image produced by the linker is blown into the ROM and contains both the code and a copy of the initialized data.

Image properties

The image is contained in a single object file called `object1.o`.

The image consists of 10KB of code and 10KB of initialized data. Before the image is executed, you must also create a 10KB zero-initialized data area.

When the image is loaded in memory, both the code and initialized data are placed together in ROM, beginning at address 0x01000000. Thus the image has only one load region, starting at 0x01000000.

Before execution, however, the initialized data has to be moved to RAM at location 0x8000, and the zero-initialized data created immediately afterwards. The code is to remain at the load address. There are therefore two execution regions for the image, one region each in ROM and RAM.

Description file

```
ROM 0x01000000 0x80000         ; load-region base-address max-size
{
    ROM 0x01000000             ; execution-region1 base-address
    {
          object1.o            ; area attribute is (+RO) by default
    }
    RAM 0x8000                 ; execution-region2 base-address
    {
          object1.o ( +RW )    ; module-name (area attributes)
          object1.o ( +ZI )
    }
}
```

There is one load description for the load region, and within it are two execution region descriptions for the two execution regions.

The first execution region description specifies that it can be executed at its load address in ROM.

The other execution region consists of two areas from `object1.o`; the initialized data, which needs relocation to 0x8000, and the zero-initialized data, which needs to be created immediately after the initialized data.

Symbols

To describe this memory mapping to the startup routine, the linker generates the following symbols and values:

Symbol	Value
`Load$$ROM$$Base`	0x01000000
`Image$$ROM$$Base`	0x01000000
`Image$$ROM$$Length`	0x2800
`Load$$RAM$$Base`	0x01002800
`Image$$RAM$$Base`	0x8000
`Image$$RAM$$Length`	0x2800
`Image$$RAM$$ZI$Base`	0xA800
`Image$$RAM$$ZI$$Length`	0x2800

Table 2: Example 1—symbol values

6.2 Example 2

Memory map

The memory map is shown in the following table:

Name	Base	Size
ROM	0x0000	0x8000
SRAM	0x8000	0x8000
EEPROM	0x10000	0x8000
DRAM	0x18000	0x8000

Table 3: Example 2—memory map

Image properties

The image is contained in two object files: `object1.o` and `object2.o`.

There are three areas in each of the object files (RO, RW, and ZI).

When the image is loaded in memory, assume that the RO and RW areas from `object1.o` are placed in ROM starting at 0x0000, and the RO and RW areas from `object2.o` are placed in ROM starting at 0x4000. The image therefore has two load regions.

Before execution, the DRAM should contain the RW area from `object1.o` and the ZI area from `object2.o`. The SRAM should contain the RW area from `object2.o` and the ZI area from `object1.o`.

The required positions of the various areas are:

From `object1.o`	The RO area remains where it is loaded.
	The RW area needs to be placed in DRAM at 0x18000.
	The ZI area needs to be created in SRAM.
From `object2.o`	The RO area remains where it is loaded.
	The RW area needs to be placed in SRAM at 0x8000.
	The ZI area needs to be created in DRAM.

For simplicity, assume that each area is of the same size: 1024 bytes (0x400)

Description file

```
ROM_1 0x0000                    ; load-region-name base-address
{
    ROM 0x0000                  ; execution-region1-name base-address
    {
           object1.o (+RO)
    }
    DRAM 0x18000                ; execution-region2-name base-address
    {
           object1.o (+RW)
           object2.o (+ZI)
    }
}
ROM_2 0x4000
{
    ROM_2 0x4000                ; execution-region1-name base-address
    {
           object2.o (+RO)
    }
    SRAM 0x8000                 ; execution-region2-name base-address
    {
           object1.o (+ZI)
           object2.o (+RW)
    }
}
```

There are therefore two load regions and four execution regions.

Symbols

To describe this memory mapping to the startup routine, the linker generates the following symbols and values:

Symbol	Value
`Load$$ROM_1$$Base`	0x0000
`Image$$ROM_1$$Base`	0x0000
`Image$$ROM_1$$Length`	0x400
`Load$$DRAM$$Base`	0x0400
`Image$$DRAM$$Base`	0x18000
`Image$$DRAM$$Length`	0x400
`Image$$DRAM$$ZI$$Base`	0x18400
`Image$$DRAM$$ZI$$Length`	0x400
`Load$$ROM_2$$Base`	0x4000
`Image$$ROM_2$$Base`	0x4000
`Image$$ROM_2$$Length`	0x400
`Load$$SRAM$$Base`	0x4400
`Image$$SRAM$$Base`	0x8000
`Image$$SRAM$$Length`	0x400
`Image$$SRAM$$ZI$$Base`	0x8400
`Image$$SRAM$$ZI$$Length`	0x400

Table 4: Example 2—symbol values

6.3 Example 3

Memory map

The memory map is shown in the following table:

Name	Base	Size
ROM	0x0000	0x8000
RAM	0x8000	0x10000

Table 5: Example 3—memory map

Image properties

The image is contained in one object file called `object1.o`.

There are three areas in the object files (RO, RW, and ZI).

There is also an Interworking veneer area called `IWV$$Code` within `object1.o`.

When the image is loaded in memory, it is loaded in ROM at 0x0000, and has only one load region.

Before execution, the RW area should be placed in RAM, starting at 0x8000. The ZI area should be placed in RAM, 1024 (0x400) bytes after the RW area in RAM.

The Interworking veneer area should be placed in ROM, alongside the RO area.

For simplicity, assume that each area is of the same size : 1024 bytes (0x400)

Description file

```
ROM 0x0000              ; load-region-name base-address
{
    ROM 0x0000          ; execution-region1-name base-address
    {
           object1.o (+RO)
           object1.o (IWV$$Code)
    }
    RAM_1 0x8000        ; execution-region2-name base-address
    {
           object1.o (+RW)
    }
    RAM_2 +1024         ; execution-region3-name offset
    {
           object1.o (+ZI)
    }
}
```

There are therefore one load region and three execution regions.

Symbols

To describe this memory mapping to the startup routine, the linker generates the following symbols and values:

Symbol	Value
Load$$ROM$$Base	0x0000
Image$$ROM$$Base	0x0000
Image$$ROM$$Length	0x800
Load$$RAM_1$$Base	0x0800
Image$$RAM_1$$Base	0x8000
Image$$RAM_1$$Length	0x400
Image$$RAM_2$$ZI$$Base	0x8400
Image$$RAM_2$$ZI$$Length	0x400

Table 6: Example 3—symbol values

7 Initialization Code for Scatter Loading

You should place your scatter loading initialization code in the startup routine of an application. Your initialization code should copy the required regions from their load addresses to their execution addresses, and create all the zero-initialized areas.

The code uses the symbols generated by the linker to obtain information about each region. For a list of the symbols generated by the linker, refer to ***5.3 Linker pre-defined symbols*** on page 11.

The initialization code can be written in ARM assembly language. The following is a sample initialization code that corresponds to ***6.1 Example 1*** on page 16. Other examples have similar initialization code, only the regions to be moved or created vary.

```
        ; r0 contains the load address of the region RAM
        LDR  r0,  = |Load$$RAM$$Base|
        ; r1 contains the execution address of the region RAM
        LDR  r1,  = |Image$$RAM$$Base|

        CMP  r0,  r1         ; check source & destination are different
        BEQ  do_zi_init      ; if not, do not copy this region

        ; copy this region from load address to execution address
        ; r2 contains the address of the word beyond the end of this
        ; execution region
        MOV  r2,  r1
        LDR  r4,  = |Image$$RAM$$length|
        ADD  r2,  r2,  r4
        BL   copy

do_zi_init
        ; r1 contains the execution address of the region
        LDR  r1, = |Image$$RAM$$ZI$$Base|
        ; r2 contains the address of the word beyond the end of this
        ; execution region
        MOV  r2,   r1
        LDR  r4,  = |Image$$RAM$$ZI$$length|
        ADD  r2,  r2,  r4

        ; r3 contains the value to be used to initialise area
        MOV  r3,  #0
        BL   zi_init   ; call subroutine zi_init
```

The code initially loads the values of the `Load$$RAM$$Base`, which is the load address of the region called RAM, and compares it to the `Image$$RAM$$Base`, which is the execution address. If the two are same, then the region is where it is supposed to be, and you can create the zero-initialized data, by calling the subroutine `zi_init`. If not, the region RAM is copied to its execution location by calling the `copy` subroutine.

The code for the `copy` and `zi_init` subroutines is given in ***4.2.2 Subroutine code*** on page 9.

8 The Description File Format

The file format reflects the hierarchy of load regions, execution regions and object areas. An object area can be in precisely one execution region. An execution region can be in precisely one load region.

Lexically, a description is a sequence of tokens, whitespaces and comments.

special characters Single-characters with special significance are: ( ) { } " , + and ; (`LPAREN`, `RPAREN`, `LBRACE`, `RBRACE`, `QUOTE`, `COMMA`, `PLUS` and `SEMIC`)

tokens Tokens are `LPAREN`, `RPAREN`, `LBRACE`, `RBRACE`, `COMMA`, `PLUS`, `WORD` and `NUMBER`.

comments A `SEMIC` following the end of a token begins a comment which extends to the end of the current line. This means that a `WORD` cannot begin with a `SEMIC` (unless it is enclosed in `QUOTE`s).

numbers A `NUMBER` has one of the forms:
"0" octal-digit+
"&" hex-digit+
"0x" hex-digit+
"0X" hex-digit+
decimal-digit+

A `NUMBER` encodes a 32-bit unsigned value.

words A `WORD` is an alternation of quoted and unquoted `WORD`-segments.

unquoted word segment An unquoted `WORD`-segment terminates on the first character in the set {`Whitespace, LPAREN, RPAREN, LBRACE, RBRACE, COMMA, PLUS, QUOTE`}.

quoted word segment A quoted `WORD`-segment is enclosed by `QUOTE` characters and may contain any characters except `newline`. All other characters of which the ANSI C function `isspace()` is true are translated to space. Two consecutive `QUOTE`s stand for the literal `QUOTE` character and neither begin nor end a quoted `WORD`-segment.

Structurally, a scatter load description is a sequence of load region descriptions. Formally:

```
Scatter-description ::= load-region-description+
```

Load region description

A load region has a name, a base address, an optional maximum size and a non-empty list of execution regions. Formally:

```
load-region-description ::=

   load-region-name base-address [ max-size ]

          LBRACE execution-region-description+ RBRACE
```

The linker allows an empty description.

Only the first 31 characters of the *load-region-name* are significant. In multi-file output formats (for example -BIN -SCATTER) *load-region-name* is used to name the file containing initializing data for this load region. On hosts which have shorter limits on directory entries, fewer characters are used.

The first 31 characters of *load-region-name* are also used to manufacture base and limit symbols for the region.

base-address is the address at which the contents of the region are loaded. It must be a word-aligned NUMBER, so &1234ABDC, 0x1e4,4000 and 0 are acceptable, but 1234CD is not.

max-size is an optional NUMBER: if specified, the description is faulted if the region has more than *max-size* bytes allocated to it.

Execution region description

An execution region is described by a name, a base address and an optional OVERLAY attribute. Formally:

```
execution-region-description ::=
   exec-region-name base-designator [ "OVERLAY" ]
         LBRACE object-AREA-description* RBRACE

base-designator ::= base-address  |  "+" offset
```

The "+" offset form of base-designator describes a base-address *offset* bytes beyond the end of the preceding execution region. The length of a region is always a multiple of four bytes, so *offset* must be a multiple of four bytes too. If there is no preceding execution region (that is, if this is the first in the load region) then "+" *offset* means *offset* bytes after the base of the containing load region.

base-address is the address at which objects in the region should be linked. It must be a word-aligned NUMBER.

Armlink faults overlapping execution regions unless they have the "OVERLAY" attribute. For overlay regions that overlap, armlink builds clash maps and generates a reference to the overlay manager (which must already have been included in the image). Overlay segments are given names derived from the *exec-region-name*.

A root region is a non-OVERLAY execution region with its load address equal to its execution address. Only a root region may contain an entry point.

Object area descriptions

An object-AREA-description is a pattern that identifies AREAs by:

- module name (object file name, library member name or library file name) and;
- AREA name or AREA attributes such as READ-ONLY, CODE, and so on. Formally:

```
object-AREA-description ::=
   module-selector-pattern [ LPAREN area-selectors RPAREN ]
```

An omitted LPAREN area-selectors RPAREN defaults to (+RO) (see below).

`area-selectors` is a comma-separated list of expressions. Each expression is a pattern against which the AREA name, or the name of an attribute you want the selected AREA to have, is matched. In the latter case the name must be preceded by a plus (+). You may omit any comma immediately followed by a PLUS. Formally:

```
area-selectors ::=

   (PLUS area-attrs | area-pattern )([ COMMA ] PLUS area-attrs |
COMMA area-pattern)*
```

Additionally, the first occurrence of FIRST or LAST as an *`area-attrs`* terminates the list.

Only AREAs that match both the module-selector and at least one area-selector are included in the execution region.

If an AREA matches more than one execution region, the matches are disambiguated as described below. If a unique match cannot be found, armlink faults the scatter description.

Note that the assignment of AREAs to regions is completely independent of the order in which patterns are written in the scatter load description.

Module-selector patterns and area patterns

A `module-selector-pattern` and an `area-pattern` are patterns constructed from literal text, and the wildcard characters * (matches 0 or more characters) and ? (matches any single character). For example:

```
*armlib.*       ; matches AREAs from any armlib.*
```

An AREA matches a `module-selector-pattern` if:

- the name of the object file containing the AREA or name of the library member (with no leading pathname) matches the `module-selector-pattern`
- the full name of the library from which the AREA was extracted matches the `module-selector-pattern`

Matching is case-insensitive, even on hosts with case-sensitive file naming.

Area selector

An `area-selector` is:

- a pattern matched case-insensitively against the AREA's name
- an attribute selector matched against the area's attributes

Note *ARM-Thumb interworking veneers are built in an AREA called `IWV$$Code`. You can assign this AREA to an execution region just like any other area using the AREA selector:*

```
*(IWV$$Code)
```

*Although there is no associated module, * still matches; because there is only one `IWV$$Code` AREA, this selection is unambiguous.*

An attribute selector follows a plus (+) character. The following selectors are recognized (case-insensitively):

RO-CODE
RO-BASED-DATA
RO-DATA (includes RO-BASED-DATA)
RO (includes RO-CODE and RO-DATA)
RW-CODE
RW-BASED-DATA
RW-STUB-DATA (shared library stub data)
RW-DATA (includes RW-BASED-DATA and RW-STUB-DATA)
RW (includes RW-CODE and RW-DATA)
ZI
ENTRY (the AREA containing the ENTRY point)

The following synonyms are recognized:

```
CODE (= RO-CODE)
CONST (= RO-DATA)
TEXT (= RO)
DATA (= RW)
BSS (= ZI)
```

The following pseudo attributes are recognized:

```
FIRST
LAST
```

The pseudo-attributes FIRST and LAST can be used to mark the first and last AREAs in an execution region if the placement order is important (for example, if the ENTRY must be first and a checksum last).

Note that RO-NOTBASED-DATA cannot be specified directly. Rather, RO-BASED-DATA must be selected in one region and (less specifically) RO-DATA in another.

Disambiguating multiple matches

Every AREA is selected by a module-selector and an area-selector. Suppose AREA A matches *m1,s1* for execution region R1 and *m2,s2* for execution region R2:

1 Assign A to R1 if and only if (iff) *m1,s1* < (more specific than) *m2,s2*.

2 Assign A to R2 iff *m2,s2* < *m1,s1*.

3 Diagnose the scatter description as faulty if neither *m1,s1* < *m2,s2* nor *m2,s2* < m*1,s1*.

4 Define *m1,s1* < *m2,s2* iff:

   a) s1 is a literal AREA name (containing no pattern characters) and s2 matches AREA attributes other than +ENTRY; **or**

   b) `(m1 < m2) || !(m2 < m1) && (s1 < s2)`

5 Define m1 < m2 iff:

```
(text(m1) matches pattern(m2)) && !(text(m2) matches
pattern(m1))
```

6 If s1 and s2 are both patterns matching AREA names, the same definition of s1 < s2 holds as for m1 < m2.

Otherwise, if one of s1, s2 matches the AREA name and the other matches the AREAs attributes then neither s1 < s2 nor s2 < s1.

7 If both s1 and s2 match AREA attributes then define s1 < s2 by:

```
ENTRY < RO-CODE < RO
ENTRY < RO-BASED-DATA < RO-DATA < RO
ENTRY < RW-CODE < RW
ENTRY < RW-BASED-DATA < RW-DATA < RW
ENTRY < RW-STUB-DATA < RW-DATA < RW
```

No other members of the s1 < s2 relation between AREA attributes exist.

Consequences of this matching strategy include:

- All warning-free SDT2.1 descriptions remain valid.
- Some descriptions warned of in SDT2.1 succeed without warning; some are faulted.
- Descriptions become independent of the order in which they are written (also true of SDT2.1 warning-free descriptions).
- Usually, the more specific description of an object is the more specific description of the AREAs it contains: AREA selectors are not examined unless object selection is inconclusive or one selector fully names an AREA and the other selects by attribute. In this case, the explicit AREA name is more specific than any attribute other than ENTRY (which selects exactly one AREA from one object), even if the object selector associated with the AREA name is less specific than that associated with the attribute.

Default root region specification (obsolete)

The use of ROOT and ROOT-DATA declarations is now discouraged.

You should not use them in new scatter loading descriptions. Armlink uses the default ROOT to contain debug data and other impedimenta, but this is best kept separate from your image. They exist in SDT 2.11 only for backward compatibility, and future versions of Armlink may stop recognizing them in the description file.

You can specify a default ROOT region to contain AREAs you do not assign to any other execution region. You can also specify a default ROOT-DATA region. If you do, the unassigned read-only AREAs are placed in the default ROOT and unassigned read-write AREAs are placed in the default ROOT-DATA. You cannot specify a default ROOT-DATA region unless you specify a default ROOT.

You describe the default ROOT regions as follows:

```
ROOT root_load-address [ root_max_size ]
ROOT-DATA root_data_load_address
```

This is similar to:

```
ROOT root_load_address
{
; the ROOT load region
   ROOT root_load_address { *(*) }
}
```

or

```
ROOT root_load_address
{
; the ROOT load region
   ROOT root_load_address { *(+RO) }
   ROOT-DATA root_data_load_address { *(+RW,+ZI) }
}
```

as described in ***3 Load Regions and Execution Regions*** on page 5.

The region names `ROOT` and `ROOT-DATA` are reserved for armlink. If you do not specify a default `ROOT`, armlink faults any areas not placed specifically by your scatter description.