

SEGGER Linker

A linker for Cortex-M Microcontrollers

User Guide & Reference Manual

Document: UM20005
Software Version: 2.22
Revision: 0
Date: August 8, 2018



A product of SEGGER Microcontroller GmbH

www.segger.com

Disclaimer

Specifications written in this document are believed to be accurate, but are not guaranteed to be entirely free of error. The information in this manual is subject to change for functional or performance improvements without notice. Please make sure your manual is the latest edition. While the information herein is assumed to be accurate, SEGGER Microcontroller GmbH (SEGGER) assumes no responsibility for any errors or omissions. SEGGER makes and you receive no warranties or conditions, express, implied, statutory or in any communication with you. SEGGER specifically disclaims any implied warranty of merchantability or fitness for a particular purpose.

Copyright notice

You may not extract portions of this manual or modify the PDF file in any way without the prior written permission of SEGGER. The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such a license.

© 2017-2018 SEGGER Microcontroller GmbH, Hilden / Germany

Trademarks

Names mentioned in this manual may be trademarks of their respective companies.

Brand and product names are trademarks or registered trademarks of their respective holders.

Contact address

SEGGER Microcontroller GmbH

In den Weiden 11
D-40721 Hilden

Germany

Tel.	+49 2103-2878-0
Fax.	+49 2103-2878-28
E-mail:	support@segger.com
Internet:	www.segger.com

Manual versions

This manual describes the current software version. If you find an error in the manual or a problem in the software, please report it to us and we will try to assist you as soon as possible.

Contact us for further information on topics or functions that are not yet documented.

Print date: August 8, 2018

Software	Revision	Date	By	Description
2.22	0	180720	PC	Chapter "Linker script reference" <ul style="list-style-type: none"> • Added maximum packing sorting. • Added minimum size order sorting. Chapter "Command-line options" <ul style="list-style-type: none"> • Added --unwind-tables and --no-unwind-tables. • Added --enable-lzss and --disable-lzss. • Added --enable-packbits and --disable-packbits. • Added --enable-zpak and --disable-zpak. • Added --min-code-align. • Added --min-data-align. • Added --min-rx-align. • Added --min-ro-align. • Added --min-rw-align. • Added --min-zi-align. • Added --min-zi-align. • Deprecated --full-program-headers. • Deprecated --load-program-headers.
2.20	0	180717	PC	Chapter "Linker script reference" <ul style="list-style-type: none"> • Added size then alignment order sorting. • Added alignment then size order sorting.
2.18	0	180705	PC	Chapter "Command-line options" <ul style="list-style-type: none"> • Added --auto-es-block-symbols. • Added --auto-es-region-symbols.
2.16	0	180502	PC	Upgraded to latest software version.
2.14	0	180411	PC	Chapter "Linker script reference" <ul style="list-style-type: none"> • place at now offers extended selection. Chapter "Command-line options" <ul style="list-style-type: none"> • --auto-es-symbols defines additional symbols.
2.12	0	180216	PC	Upgraded to latest software version.
2.10	0	180107	PC	Initial release.
1.00	0	170909	PC	Internal release.

About this document

Assumptions

This document assumes that you already have a solid knowledge of the following:

- The software tools used for building your application (assembler, linker, C compiler).
- The C programming language.
- The target processor.
- DOS command line.

If you feel that your knowledge of C is not sufficient, we recommend *The C Programming Language* by Kernighan and Richie (ISBN 0--13--1103628), which describes the standard in C programming and, in newer editions, also covers the ANSI C standard.

How to use this manual

This manual describes all the functions and macros that the product offers. It assumes you have a working knowledge of the C language. Knowledge of assembly programming is not required.

Typographic conventions for syntax

This manual uses the following typographic conventions:

Style	Used for
Body	Body text.
Parameter	Parameters in API functions.
Sample	Sample code in program examples.
Sample comment	Comments in program examples.
User Input	Text entered at the keyboard by a user in a session transcript.
Secret Input	Text entered at the keyboard by a user, but not echoed (e.g. password entry), in a session transcript.
Reference	Reference to chapters, sections, tables and figures or other documents.
Emphasis	Very important sections.
SEGGER home page	A hyperlink to an external document or web site.

Table of contents

1	About the linker	10
1.1	Introduction	11
1.1.1	What is the SEGGER Linker?	11
1.1.2	Linker features	11
1.1.3	Linker inputs	11
1.1.4	Linker outputs	11
2	Using the linker	12
2.1	Memories	13
2.2	Regions	14
2.2.1	Memory ranges	14
2.2.2	Naming a region	15
2.2.3	Combining regions	15
2.3	A simple linker script	17
2.4	Grouping code and data	19
2.4.1	Creating groups	19
2.4.2	Inline and nested blocks	19
2.5	Fixed placement of objects	20
2.5.1	Placement by linker script	20
2.5.2	Placement by section name	20
2.6	Placing code in RAM	21
2.7	Placement with preferences	22
2.7.1	A simple example	22
2.7.2	A more complex example	22
2.8	Thread-local data	24
3	Linker script reference	25
3.1	Memory ranges and regions	26
3.1.1	define memory statement	26
3.1.2	define region statement	27
3.2	Sections and symbols	28
3.2.1	define block statement	28
3.2.2	define symbol statement	30
3.2.3	initialize statement	31
3.2.4	do not initialize statement	32
3.2.5	place in statement	33
3.2.6	place at statement	35
3.2.7	keep statement	37

4	Command-line options	38
4.1	Input file options	39
4.1.1	--script	39
4.1.2	--via	40
4.2	Output file options	41
4.2.1	--bare	41
4.2.2	--block-section-headers	42
4.2.3	--debug	43
4.2.4	--entry	44
4.2.5	--force-output	45
4.2.6	--full-program-headers	46
4.2.7	--full-section-headers	47
4.2.8	--load-program-headers	48
4.2.9	--no-debug	49
4.2.10	--no-entry	50
4.2.11	--no-force-output	51
4.2.12	--minimal-section-headers	52
4.2.13	--no-symbols	53
4.2.14	--output	54
4.2.15	--strip	55
4.2.16	--symbols	56
4.3	Map file options	57
4.3.1	--map-all	57
4.3.2	--map-defaults	58
4.3.3	--map-detail	59
4.3.4	--map-file	60
4.3.5	--map-init-table	61
4.3.6	--map-listing	62
4.3.7	--map-modules	63
4.3.8	--map-narrow	64
4.3.9	--map-none	65
4.3.10	--map-placement	66
4.3.11	--map-symbols	67
4.3.12	--map-wide	68
4.3.13	--no-map-detail	69
4.3.14	--no-map-init-table	70
4.3.15	--no-map-listing	71
4.3.16	--no-map-modules	72
4.3.17	--no-map-placement	73
4.3.18	--no-map-symbols	74
4.4	Log file options	75
4.4.1	--log-file	75
4.5	Symbol and section options	76
4.5.1	--add-region	76
4.5.2	--autoat	77
4.5.3	--autokeep	78
4.5.4	--auto-arm-symbols	79
4.5.5	--auto-es-symbols	80
4.5.6	--auto-es-block-symbols	81
4.5.7	--auto-es-region-symbols	82
4.5.8	--define-symbol	83
4.5.9	--disable-lzss	84
4.5.10	--disable-packbits	85
4.5.11	--disable-zpak	86
4.5.12	--enable-lzss	87
4.5.13	--enable-packbits	88
4.5.14	--enable-zpak	89
4.5.15	--keep	90
4.5.16	--min-code-align	91

4.5.17	--min-data-align	92
4.5.18	--min-ro-align	93
4.5.19	--min-rw-align	94
4.5.20	--min-rx-align	95
4.5.21	--min-zi-align	96
4.5.22	--no-autoat	97
4.5.23	--no-autokeep	98
4.5.24	--no-auto-arm-symbols	99
4.5.25	--no-auto-es-symbols	100
4.5.26	--no-auto-es-block-symbols	101
4.5.27	--no-auto-es-region-symbols	102
4.5.28	--no-unwind-tables	103
4.5.29	--unwind-tables	104
4.6	Control options	105
4.6.1	--cpu=name	105
4.6.2	--list-all-undefs	106
4.6.3	--no-list-all-undefs	107
4.6.4	--no-warnings	108
4.6.5	--no-remarks	109
4.6.6	--remarks	110
4.6.7	--remarks-are-errors	111
4.6.8	--remarks-are-warnings	112
4.6.9	--silent	113
4.6.10	--verbose	114
4.6.11	--warnings	115
4.6.12	--warnings-are-errors	116
4.7	Compatibility options	117
4.7.1	--begin-group	117
4.7.2	--end-group	118
4.7.3	--emit-relocs	119
4.7.4	--allow-multiple-definition	120
4.7.5	--gc-sections	121
4.7.6	-EL	122
4.7.7	--omagic	123
4.7.8	--discard-locals	124
5	Indexes	125
5.1	Subject index	126

Chapter 1

About the linker

1.1 Introduction

This section presents an overview of the SEGGER Linker and its capabilities.

1.1.1 What is the SEGGER Linker?

The SEGGER Linker is a fast linker that links applications for execution on Cortex-M microcontrollers. It is designed to be very flexible, yet simple to use.

The linker combines the one or more ELF object files and supporting ELF object libraries to produce an executable image. This image is suitable for programming a Cortex-M microcontroller.

1.1.2 Linker features

The SEGGER Linker has the following features:

- Highly efficient and very fast to link.
- Flow code and data over multiple memory areas.
- Place a function or data at a specific address with ease.
- Avoid placing code and data in "keep out areas."
- Sort code and data sections for improved packing or by user preference.
- Automatically generate runtime initialization code prior to entering `main()`.
- Easily places code in RAM with optional flash-image compression.
- Will copy initialized data with optional flash-image compression.
- Eliminates all unused code and data for a minimum-size image.
- Accepts standard Arm ELF object files and libraries from any toolset.
- Generates range extension veneers as required for branch and branch-and-link instructions.
- Writes a map file that is clean and easily understood.
- Optionally generates a log file that provides additional information about the linking process.

1.1.3 Linker inputs

The SEGGER linker accepts one or more Arm ELF object files generated by standard-conforming toolchains such as SEGGER Embedded Studio.

In addition, the following files can be used as input to the linker:

- Arm object libraries created by a librarian.
- A linker control file to control placement of sections and how linking proceeds.
- An indirect file that extends command line arguments.

1.1.4 Linker outputs

The linker generates the following outputs:

- A fully-linked Arm ELF executable.
- An optional map file containing symbol addresses and related information.
- An optional log file providing additional information about the linking process.

Chapter 2

Using the linker

This section describes how to use the linker to link your Cortex-M application.

2.1 Memories

The SEGGER linker is capable of linking an application for any Cortex-M device. However, it has no internal knowledge of the way that memory is laid out for any device and must be told, using a linker script or command line options, the specific memory layout of the target device.

The **define memory** statement defines the single memory space used by Cortex-M devices and its size.

2.2 Regions

The **define region** statement defines a region in the available memory into which sections of code and data can be placed. The **define region** statement is much the same as a C preprocessor `#define` directive, it merely names a region of memory.

A memory region consists of one or more memory ranges, where a memory range is a contiguous sequence of bytes. Memory regions are central to the way that the linker allocates sections, and groups of sections, into the available memory.

2.2.1 Memory ranges

A memory range is specified in one of two forms:

- a *base address* and a *size*, or
- a *start address* and an *end address*.

The way that you specify a memory region does not matter, and the form that you choose is the one that naturally fits your view of the memory region.

2.2.1.1 Base address and size

The memory range specified by a base and size has the following syntax:

```
[ from addr size expr ]
```

This declares a memory range that starts at the base address *addr* and extends for *expr* bytes. A typical memory range for a Cortex-M device declared in this fashion might be:

```
[from 0x20000000 size 128k]
```

Note that it is possible for the size to be zero, in which case the memory range is considered “null” and will not have anything allocated into it.

2.2.1.2 Start address and end address

The memory range specified by a start address and an end address has the following syntax:

```
[ from start-addr to end-addr ]
```

This declares a memory range that starts at the address *start-addr* and extends up to and including the address *end-addr*. A typical memory range for a Cortex-M device declared in this fashion might be:

```
[from 0x20000000 to 0x2001ffff]
```

2.2.1.3 Repeated ranges

It is possible to define a repeated range using **repeat** and **displacement** in the memory range:

```
repeat count [ displacement offset ]
```

If the displacement is absent it defaults to the size of the base range.

For instance, the range:

```
[from 0x20000000 size 32k repeat 4 displacement 64k]
```

results in a repeated range equivalent to the following:

```
[from 0x20000000 size 0x8000]  
[from 0x20010000 size 0x8000]  
[from 0x20020000 size 0x8000]  
[from 0x20030000 size 0x8000]
```

2.2.2 Naming a region

Regions are used to place data and code into memory, which is described later. However, it's convenient to name regions that correspond to the use they have. The **define region** statement defines a name for a region so that the region can be referred to using that name, for example:

```
define region FLASH = [0x00000000 size 2m];  
define region RAM   = [0x20000000 size 192k];
```

2.2.3 Combining regions

The linker can combine ranges and regions in different ways. Every memory range is a valid memory region. The syntax for region expression is:

```
region-expr + region-expr  
region-expr - region-expr  
region-expr & region-expr  
region-expr | region-expr
```

2.2.3.1 Region union

Two regions can be combined by using the + or | operators. In this case the result is the union of the two regions, for instance:

```
[from 0x1fff0000 size 32k] + [from 0x20000000 size 192k]
```

The resulting region is the combination of the two ranges, and as such there is a hole of 32k in the region, at 0x1fff8000 of size 32k.

It is possible to combine two contiguous ranges into a single region:

```
[from 0x1fff0000 size 64k] + [from 0x20000000 size 192k]
```

In this case the resulting region defines a contiguous range of addresses from 0x1fff0000 of size 256k, but the region remains divided into two ranges.

Note

The fact that these two remain divided is very important for some devices where two memories are contiguous and, from a programmer perspective, would appear to be a single linear address space to allocate data into, *but from a hardware perspective are discrete*. It is common for a microcontroller to fault when reading misaligned data that spans the boundary between the two memories or when reading double-word data that spans the same boundary.

The Kinetis devices, which have such a boundary, are affected by this. Because the linker knows that there are two distinct ranges, it ensures that objects are allocated completely within each range and not over the seemingly contiguous memory area thus avoiding any latent bugs with memory accesses.

When two ranges overlap, they are combined into a single range. Therefore:

```
[from 0x1fff0000 size 64k+1] + [from 0x20000000 size 192k]
```

produces a region with a single range:

```
[from 0x1fff0000 size 256k]
```

2.2.3.2 Region subtraction

Two regions can be subtracted by using the - operator. In this case the result is generally two ranges, for instance you can punch a hole in a range by subtracting from it:

```
[from 0x1fff0000 size 256k] - [from 0x1fff8000 size 32k]
```

The resulting region is broken into two ranges:

```
[from 0x1fff0000 size 32k] + [from 0x20000000 size 192k]
```

This operator is particularly effective when you need to reserve places in flash or RAM for configuration or personalisation data or for bootloaders and other items.

Example

```
define region FLASH      = [from 0x00000000 size 2m];  
define region BOOTLOADER = [from 2m-128k    size 64k];  
define region CONFIG     = [from 2m-32k     size 32k];  
define region APP        = FLASH - BOOTLOADER - CONFIG;
```

In this case the region APP is:

```
[0x00000000 to 0x001dffff] + [0x001f0000 to 0x001f7fff]
```

2.2.3.3 Region intersection

The intersection of two regions is calculated by the & operator. For instance:

```
[from 0x20000000 size 256k] & [from 0x1fff8000 size 96k]
```

results in the region:

```
[from 0x20000000 size 64k]
```


2.3 A simple linker script

The linker places code and data into regions according to the user's linker script. Each required section required from the input files is mapped to a specific region using **place in** and, optionally, **place at** statements.

The specific layout of code and data will depend upon the target device. A simple linker script would define the memory required for code and data and map the appropriate sections to RAM and flash, for instance:

```
define memory with size = 4g; ❶

define region FLASH = [from 0x00000000 size 2m]; ❷
define region RAM    = [from 0x1fff0000 size 64k] + ❸
                      [from 0x20000000 size 192k];

place at start of FLASH { section .vectors }; ❹
place in FLASH          { readonly, readexec }; ❺
place in RAM            { readwrite, zeroinit }; ❻

keep { section .vectors }; ❼
```

❶ Define memory

The **define memory** statement covers the entire Cortex-M address space.

❷ Define flash region

This **define region** statement declares that the first two megabytes of memory are for flash. In fact, the linker does not distinguish between read-only flash and read-write RAM, all it is concerned with is mapping the user's sections from relocatable object files into the regions defined to hold them: the fact we have called this region `FLASH` is for our convenience only.

❸ Define RAM region

This **define region** statement declares that there are two ranges that are allocatable as RAM. The linker will ensure that individual sections placed into this region will be entirely contained in one of the ranges and will not span between the range.

❹ Place vectors

This **place at** statement instructs the linker to place the `.vectors` section at the start of flash memory. The vectors section typically contains the reset program counter and initial stack pointer along with interrupt and exception vectors for the target device and must, by convention, be placed at the start of memory.

❺ Place read-only code and data

This **place in** statement instructs the linker to place all read-only and read-execute sections into the FLASH region. Vectors have already been placed at the start of region, so the linker will place additional input sections into the remaining flash.

❻ Place read-write data

This **place in** statement instructs the linker to place all read-write and zero-initialized data into RAM.

❼ Keep vectors

This **keep** statement instructs the linker to keep the vectors section. The linker eliminates all code and data that is not specifically referenced by the linked application, so dead code and data are removed from the linked application and do not form part of the image. As there is usually no reference to the vectors section by the application, it would be eliminated by

the linker in normal operation. The **keep** statement instructs the linker to keep the vectors section, and anything it references, in the application.

You can use the **keep** statement to keep otherwise-unreferenced data in your application, such as configuration data or personalization data.

2.4 Grouping code and data

Although the previous script works, it may not produce the most compact output as initialize data (selected by **readwrite**) and initialized data (selected by **zeroinit**) can be interleaved in order to reduce the amount of padding required to align each section and, in doing so, make the linker initialization tables larger.

It is better to group all zero-initialized data together and all read-write initialized data together so that each produce a single entry in the initialization table rather than many entries.

2.4.1 Creating groups

You can collect all such data by using a block:

```
define block rdata { readwrite };  
define block zidata { zeroinit };
```

And then place these blocks into RAM:

```
place in RAM { block rdata, block zidata };
```

Blocks can also be used to reserve memory or fix the order of sections when allocating them to memory.

2.4.2 Inline and nested blocks

Rather than define two blocks separately, it's possible to declare blocks inline, so the above can be written as:

```
place in RAM {  
    block rdata { readwrite },  
    block zidata { zeroinit }  
};
```

As the block names are now redundant, they can be omitted:

```
place in RAM {  
    block { readwrite },  
    block { zeroinit }  
};
```

2.5 Fixed placement of objects

With embedded systems it is essential to be able to place code and data in memory at known locations or to avoid known locations. Examples of this include placing jump tables or vectors at a specific location, avoiding flash security areas, or placing configuration data at a specific address.

2.5.1 Placement by linker script

Placing an object at a fixed address requires a single statement:

```
place at address 0x400 { section .config };
```

This places the `.config` section at address `0x400`. In GNU C syntax you can define the object allocated to the section like this:

```
static unsigned char __attribute__((section(".config"))) _aConfig[32];
```

If the object has `extern` linkage, like this:

```
unsigned char aConfig[32];
```

then you can use the symbol name directly bypassing the requirement to use a named section:

```
place at address 0x400 { symbol aConfig };
```

2.5.2 Placement by section name

A more convenient way to place objects at a specific address is to use a capability of the linker which interprets section names. A section name that is `".ARM.__at_0xaddr"` is interpreted as an instruction to place the corresponding section at the address `addr`. For instance:

```
void __attribute__((section(".ARM.__at_0x8000"))) MyFunc(void) {  
    ...  
}
```

This locates the function `MyFunc` to start at address `0x8000`. The same scheme extends to variables:

```
unsigned char __attribute__((section(".ARM.__at_0x400"))) aConfig[32] = {  
    ...  
}
```

Interpretation of section names is turned on by default. You control this capability using the `--autoat` and `--no-autoat` command line switches: see `--autoat` on page 77 and `--no-autoat` on page 97.

2.6 Placing code in RAM

The linker is capable of placing code in RAM and ensuring that the code is copied from flash to RAM before entering `main()`. In fact, the linker treats RAM-based code no different from RAM-based data that is also initialized before entering `main()`.

It's possible to use the linker script with section names or symbol names to select the code that is to be placed in RAM. Continuing from the previous examples, a single function can be placed in RAM by setting its section name:

```
void __attribute__((section(".ramfunc"))) MyFunc(void) {  
    ...  
}
```

And then placing all such sections in RAM in the linker script:

```
place in RAM      { section .ramfunc };  
initialize by copy { section .ramfunc };
```

The `initialize by copy` is required as the linker assumes that all sections containing read-execute code are implicitly present at system startup, *even if they have been placed into RAM*. The `initialize by copy` overrides the default handling of this particular section and instructs the linker to make an image of the section and copy it to its final destination on system startup.

Note

At high optimization levels the compiler may make an inline copy of a function even though it is declared to be in a nondefault section. To ensure that the function is indeed run from RAM the compiler must be told not to inline the function:

```
void __attribute__((section(".ramfunc"), noinline)) MyFunc(void) ...
```

Tightly-coupled memory

Some controllers have tightly-coupled instruction memory which benefits fast execution, and the above mechanism is easily extended to place code into ITCM SRAM.

2.7 Placement with preferences

The linker is capable of automatically placing code and data across multiple noncontiguous memory ranges.

2.7.1 A simple example

Kinetis devices have two separate memory ranges where execution from the lower RAM is fast, but execution from the upper RAM is slower.

A simple configuration which allocates code to run in RAM along with data would be:

```
define region RAM = [from 0x1fff0000 size 64k] +  
                   [from 0x20000000 size 192k];  
  
place in RAM { readwrite, zeroinit, section .fastrun };
```

The linker will allocate code and data, however it fits, into the RAM with no priority given to placing code in the lower fast-execute region.

A simple remedy for this is to use two separate ranges and manually distribute the code and data:

```
define region LORAM = [from 0x1fff0000 size 64k];  
define region HIRAM = [from 0x20000000 size 192k];  
  
place in LORAM      { section .fastrun      };  
place in LORAM + HIRAM { readwrite, zeroinit };
```

This will allocate the code that should run quickly into the low memory, with read-write and zero-initialized data allocated to the remainder. The linker will typically allocate from low memory to high memory in a memory region, so it's highly likely that some data will be allocated to the low memory which may impact performance.

What we need to do is instruct the linker to apply a *preference order* for the data, to try to allocate objects first into high memory and, if any do not fit, to place the overflow into the low memory with the code. We specify a preference using `then`:

```
define region LORAM = [from 0x1fff0000 size 64k];  
define region HIRAM = [from 0x20000000 size 192k];  
  
place in LORAM      { section .fastrun      };  
place in HIRAM then LORAM { readwrite, zeroinit };
```

With this script, we accomplish the best layout possible without sacrificing flexibility: the code is placed into fast-executing low memory, the data is separated into high memory that doesn't affect performance, and if data is so big it will not fit into high memory, it overflows into low memory which might affect performance but will not cause a link error.

2.7.2 A more complex example

The STM32H7 has a number of RAM regions; taking the STM32H743x as an example, it has:

- 64K of tightly-coupled instruction memory (ITCM-SRAM).
- 128K of tightly-coupled data memory (DTCM-SRAM) organized as two banks of 64K.
- 512K of RAM on the AXI bus, D1 domain (AXI-SRAM).
- Two 128K banks of RAM on the AHB bus, D2 domain (AHB-SRAM1 and AHB-SRAM2).
- An addition 32K of RAM on the AHB bus, D2 domain (AHB-SRAM3)
- 64K of RAM on the AHB bus, D3 domain (AHB-SRAM4).
- 4K of backup RAM, D3 domain.

Clearly, organizing how memories are assigned is highly important when designing a system using this processor as each domain can be accessed concurrently by processor and peripheral. In addition, the two DTCM-SRAM banks can be used in parallel.

The following is an example of how to construct a linker script for this device, placing data used for Ethernet and USB in separate memories so they can be used in parallel, and assigning data in a way that maximizes performance whilst, at the same time, relieving the developer from arduous section placement.

```
define region ITCM_SRAM    = [from 0x00000000 size 64k]; ❶
define region DTCM_SRAM   = [from 0x20000000 size 128k];
define region AXI_SRAM    = [from 0x24000000 size 128k];
define region AHB_SRAM1   = [from 0x30000000 size 128k];
define region AHB_SRAM2   = [from 0x30020000 size 128k];
define region AHB_SRAM3   = [from 0x30040000 size 32k];
define region AHB_SRAM4   = [from 0x38000000 size 64k];
define region BACKUP_SRAM = [from 0x38800000 size 4k];

place in AHB_SRAM1 { section .ETH.bss*, section .ETH.data* }; ❷
place in AHB_SRAM4 { section .USB.bss*, section .USB.data* };

place in ITCM_SRAM { section .ramcode* }; ❸
initialize by copy { section .ramcode* };

place in BACKUP_SRAM { section .backup* }; ❹
do not initialize { section .backup* };

place in      DTCM_SRAM ❺
then AXI_SRAM
then AHB_SRAM2 + AHB_SRAM3
then AHB_SRAM1 + AHB_SRAM4
then BACKUP_SRAM { readwrite, zeroinit };
```

❶ Define regions

Configuring regions is straightforward, one definition per memory:

❷ Placing Ethernet and USB RAM

We separate Ethernet and USB RAM use over two regions using SRAM1 for Ethernet and SRAM4 for USB.

❸ Placing fast code

Code that needs to run quickly, such as interrupt routines, is best placed into the ITCM SRAM. The linker is told to initialize the code placed into the ITCM SRAM by copying it there (see *Placing code in RAM* on page 21).

❹ Placing backup data

Data to be preserved across power cycles must be assigned to the battery backup area, but also we must instruct the linker not to zero or otherwise initialize this data on startup.

❺ Placing application data

All that remains is to place application data with a specific preference. We should use the zero-wait-state DTCM first, followed by fast RAM areas, and any overflow should be allocated to the unused parts of the USB and Ethernet areas, and finally the battery-backup area as a last resort.

Any specific assignments of sections to memories should be made before the final catch-all placement.

2.8 Thread-local data

It is possible to use thread-local data in applications, but it requires some support from the real-time operating system and the linker script to work correctly. Typically the RTOS requires that thread-local read-write data and zero-initialized data are allocated to a contiguous block.

The following example shows how the thread-local read-write data and zero-initialized data are combined into two blocks, and those blocks are then allocated in a specific order to satisfy the requirements of the real-time operating system:

```
define block tbss { section .tbss, section .tbss.* };
define block tdata { section .tdata, section .tdata.* };
define block tls { block tbss, block tdata };
```

It is possible for the blocks to be declared inline:

```
define block tls {
    block tbss { section .tbss, section .tbss.* },
    block tdata { section .tdata, section .tdata.* }
};
```

The specific arrangement for thread-local data will be documented in the real-time operating system manual. You will need to refer to this manual when using the SEGGER linker with your RTOS.

Chapter 3

Linker script reference

3.1 Memory ranges and regions

3.1.1 define memory statement

Syntax

```
define memory [ name ] [ with size=expr ] ;
```

Description

Defines the memory *name* with size *expr*; if *name* is omitted, it defaults to "mem" and if the **size** attribute is omitted, the memory size is set to "4G".

There must be at exactly one memory defined by the script before any memory regions are declared.

3.1.2 define region statement

Syntax

```
define region name = region-expr ;
```

Description

Defines the named region *name* to the region expression *region-expr*.

See also

Regions on page 14

3.2 Sections and symbols

3.2.1 define block statement

Syntax

```
define block name [ with attr, attr... ] {  
    section-selectors  
} [ except {  
    section-selectors  
} ] ;
```

where *attr* is one of:

```
size=expr  
alignment=expr  
auto order  
fixed order  
size order  
alignment order  
alphabetical order  
size then alignment order  
alignment then size order  
alignment then alphabetical order  
reverse size order  
reverse alphabetical order  
[ readwrite | rw ] access  
[ readexec | rx ] access  
mpu ranges
```

3.2.1.1 Size

The size of the block can be set using the **size** attribute. If the block size is set with this attribute, the block size is fixed and does not expand. If the block size is not set with a **size** attribute, the block size is as large as required to contain its inputs.

3.2.1.2 Alignment

The minimum alignment of the block can be set using the **alignment** attribute. The final alignment of the block is the maximum of the alignment set by the alignment attribute (if any) and the maximum alignment of any input section.

3.2.1.3 Section ordering

The section ordering attributes are:

auto order

All inputs in the block are automatically ordered to reduce inter-object gaps. This is the default if no order is defined by the **define block** directive.

fixed order

All inputs in the block are placed in the same order as they appear in the section selector part of the **define block** directive.

size order

All inputs in the block are ordered by size, largest to smallest.

alignment order

All inputs in the block are ordered by alignment, largest to smallest alignment.

alphabetical order

All inputs in the block are ordered by section name in alphabetical order.

size then alignment order

All inputs in the block are ordered by size, largest to smallest, and equal-size blocks are ordered by alignment, largest to smallest.

alignment then size order

All inputs in the block are ordered by alignment largest to smallest and for blocks of equal alignments, then ordered by size, largest to smallest.

alignment then alphabetical order

All inputs in the block are ordered by alignment largest to smallest and for blocks of equal alignments, then ordered by name.

reverse size order

All inputs in the block are ordered by reverse size, smallest to largest.

reverse alphabetical order

All inputs in the block are ordered by section name in reverse alphabetical order.

maximum packing [order]

All inputs in the block are ordered to reduce inter-section gaps caused by alignment and to reduce the size of the associated unwind tables (if any).

minimum size order

Identical to **maximum packing**.

3.2.1.4 Access attributes

The block section flags are set to the union of the section flags of all block inputs. The default block section flags, before being modified by additional block inputs, can be set by the following:

readwrite access

The block is given read-write access.

readexec access

The block is given read-execute access.

By default the block only has "allocation access."

3.2.1.5 Use with the MPU

The attribute **mpu ranges** instructs the linker to set the size and alignment of the block so that it can be used by the Cortex-M MPU. The linker computes the alignment and size of all input sections and then sets the block size and alignment to an appropriate power of two.

The following table shows some examples of this behavior:

Size	Align	Assigned block size and alignment
3	4	4 — block is padded to alignment
4	4	4 — exact fit for size and alignment
5	4	8 — size and alignment are increased to accomodate block

Please refer to the *ARMv7-M Architecture Reference Manual* for further information on the Cortex-M MPU.

3.2.2 define symbol statement

Syntax

```
define symbol name = number | symbol
```

Description

Defines the symbol *name* to be the value *number* or the value of the symbol *symbol*. Once defined, symbols cannot be redefined by a subsequent **define symbol** statement.

3.2.3 initialize statement

Syntax

```
initialize by copy [ with attr, attr... ] ;
```

where *attr* is one of:

```
packing=algorithm  
simple ranges  
complex ranges
```

3.2.3.1 Packing

The linker is capable of compressing initialized data or code copied to RAM using different packing algorithms. The algorithm can be one of:

```
none  
packbits  
zpak  
lzss | lz77  
auto | smallest
```

Each algorithm has different compression characteristics. The packing algorithms are:

none

The initialization image is stored verbatim without any compression.

packbits

The initialization image is compressed using the PackBits algorithm.

zpak

The initialization image is compressed using the SEGGER ZPak algorithm which is good for images that have many zeros in them but are otherwise not suitable for other forms of packing (PackBits and LZSS).

lzss or **lz77**

The initialization image is compressed using a Lempel-Ziv-Storer-Szymanski scheme.

auto or **smallest**

The linker chooses the packing algorithm that minimizes the initialization image size from the active set of algorithms.

3.2.3.2 Ranges

The **simple ranges** and **complex ranges** attributes are accepted but are otherwise ignored.

3.2.4 do not initialize statement

Syntax

```
do not initialize {  
    section-selectors  
};
```

Description

The sections selected by *section-selectors* are not added to the initialization table and will not be initialized on program entry.

This capability enables battery-backed data to be preserved across a reset or power cycle.

Example

```
do not initialize {  
    section .backup,  
    section .backup.*  
};
```


3.2.5 place in statement

Syntax

```
[ name: ] place in region-expr [ then region-expr... ] [ with attr, attr... ] {
    section-selectors
} [ except {
    section-selectors
} ] ;
```

where *attr* is one of:

```
auto order
fixed order
size order
alignment order
alphabetical order
size then alignment order
alignment then size order
alignment then alphabetical order
reverse size order
reverse alphabetical order
minimum size order
maximum packing [order]
```

3.2.5.1 Section ordering

The section ordering attributes are:

auto order

All inputs in the block are automatically ordered to reduce inter-object gaps. This is the default if no order is defined by the **place in** directive.

fixed order

All inputs in the block are placed in the same order as they appear in the section selector part of the **place in** directive.

size order

All inputs in the block are ordered by size, largest to smallest.

alignment order

All inputs in the block are ordered by alignment, largest to smallest alignment.

alphabetical order

All inputs in the block are ordered by section name in alphabetical order.

size then alignment order

All inputs in the block are ordered by size, largest to smallest, and equal-size blocks are ordered by alignment.

alignment then size order

All inputs in the block are ordered by alignment largest to smallest and for blocks of equal alignments, then ordered by size, largest to smallest.

alignment then alphabetical order

All inputs in the block are ordered by alignment largest to smallest and for blocks of equal alignments, then ordered by name.

reverse size order

All inputs in the block are ordered by reverse size, smallest to largest.

reverse alphabetical order

All inputs in the block are ordered by section name in reverse alphabetical order.

maximum packing [*order*]

All inputs in the block are ordered to reduce inter-section gaps caused by alignment and to reduce the size of the associated unwind tables (if any).

minimum size order

Identical to **maximum packing**.

3.2.6 place at statement

Syntax

```
[ name: ] place at [
    address expr |
    start of region-expr |
    end of region-expr ]
[ with attr, attr... ] {
    section-selectors
} [ except {
    section-selectors
} ] ;
```

where *attr* is one of:

```
auto order
fixed order
size order
reverse size order
alignment order
alphabetical order
alignment then alphabetical order
reverse alphabetical order
minimum size order
maximum packing [order]
```

3.2.6.1 Section ordering

The section ordering attributes are:

auto order

All inputs in the block are automatically ordered to reduce inter-object gaps. This is the default if no order is defined by the **place at** directive.

fixed order

All inputs in the block are placed in the same order as they appear in the section selector part of the **place at** directive.

size order

All inputs in the block are ordered by size, largest to smallest.

reverse size order

All inputs in the block are ordered by reverse size, smallest to largest.

alignment order

All inputs in the block are ordered by alignment, largest to smallest alignment.

alphabetical order

All inputs in the block are ordered by section name in alphabetical order.

alignment then alphabetical order

All inputs in the block are ordered by alignment largest to smallest and for blocks of equal alignments, then ordered by name.

reverse alphabetical order

All inputs in the block are ordered by section name in reverse alphabetical order.

maximum packing [*order*]

All inputs in the block are ordered to reduce inter-section gaps caused by alignment and to reduce the size of the associated unwind tables (if any).

minimum size order

Identical to **maximum packing**.

3.2.6.2 Placing at an address

The **place at address** statement creates an internal block for the selected sections, sorts them according to the user preference, and attempts place the block at the specified address.

Example

```
place at address 0x001FF800 { section .bootloader };
```

3.2.6.3 Placing at the start of a range

The **place at start of** statement creates an internal block for the selected sections, sorts them according to the user preference, and attempts to place the block at the start of the range.

Example

```
define region FLASH = [from 0x00000000 size 2m];  
place at start of FLASH { section .vectors };
```

3.2.6.4 Placing at the end of a range

The **place at end of** statement creates an internal block for the selected sections, sorts them according to the user preference, and attempts to place the block such that the last address used by the block is the last byte of the range.

Example

```
define region RAM = [from 0x20000000 size 192k];  
place at end of RAM { section .stack };
```

3.2.7 keep statement

Syntax

```
keep {  
    section-selectors  
};
```

Description

The **keep** statement requests that the linker keep sections in the generated output that would otherwise be discarded.

Chapter 4

Command-line options

Command line option naming is generally compatible with the following toolsets:

- GNU linker `ld`
- Arm linker `armlink`
- IAR linker `ilink`

Equivalence of command line options

The following sections describe the syntax of command line options. The SEGGER linker accepts command line options with either all hyphens between words (e.g. `--no-unwind-tables`) or with all underscores between words (e.g. `--no_unwind_tables`).

This manual shows all options using hyphens rather than underscores.

4.1 Input file options

4.1.1 --script

Synopsis

Set linker control script.

Syntax

`--script filename`
`--script=filename`
`-Tfilename`

Description

This option sets the linker script file name to *filename*.

4.1.2 --via

Synopsis

Read additional options and input files from file.

Syntax

```
--via filename  
--via=filename  
-f filename  
@filename
```

Description

This option reads the file *filename* for additional options and input files. Options are separated by spaces or newlines, and file names which contain special characters, such as spaces, must be enclosed in double quotation marks.

Notes

This option can only be provided on the command line and cannot appear in an indirect file.

4.2 Output file options

4.2.1 --bare

Summary

Generate minimal output file.

Syntax

`--bare`

`--no-sections`

Description

This option eliminates all debug information, the symbol table, and the string table from the Arm ELF file, and also removes the section table.

This option is equivalent to specifying `--no-debug` and `--no-symbols`.

See also

`--no-debug` on page 49, `--no-symbols` on page 53

4.2.2 --block-section-headers

Summary

Produce minimal section table with block information.

Syntax

--block-section-headers

Description

This option produces an Arm ELF file containing minimal section information (section header string table and string table) together with sections covering any blocks created by the linker script.

See also

--bare on page 41, *--full-section-headers* on page 47, *--minimal-section-headers* on page 52

4.2.3 --debug

Summary

Include debugging information.

Syntax

`--debug`

Description

This option instructs the linker to include any relevant debug input sections from the input object files and libraries and also includes both the symbol table and string table in the Arm ELF file.

See also

`--no-debug` on page 49

4.2.4 --entry

Summary

Set target core or architecture.

Syntax

`--entry name`
`--entry=name`
`-ename`

Description

Sets the entry point to the symbol *name* and *name* is automatically kept.

The default is `--entry=reset_handler`.

See also

`--no-entry` on page 50

4.2.5 **--force-output**

Summary

Write ELF file regardless of link errors.

Syntax

--force-output

Description

This option instructs the linker to write an ELF file even if there are link errors.

4.2.6 --full-program-headers

Summary

Produce program headers for all load regions.

Syntax

`--full-program-headers`

Description

This option produces an Arm ELF file containing program headers for all sections including regions that are initialized by the runtime startup using the initialization table.

Note

This option is now deprecated and is ignored

See also

--load-program-headers on page 48

4.2.7 --full-section-headers

Summary

Produce a section table with per-object sections.

Syntax

--full-section-headers

Description

This option produces an Arm ELF file containing a detailed per-object section table.

See also

--bare on page 41, --block-section-headers on page 42, --minimal-section-headers on page 52

4.2.8 --load-program-headers

Summary

Produce program headers for implicit load regions.

Syntax

`--load-program-headers`

Description

This option produces an Arm ELF file containing program headers only for load regions that are implicitly loaded. Any region that is initialized by an `initialize` statement or is precluded from initialization by a `do not initialize` statement will be excluded from the ELF program headers.

Note

This option is now deprecated and is ignored

See also

`--full-program-headers` on page 46

4.2.9 --no-debug

Summary

Discard debugging information.

Syntax

`--no-debug`

Description

This option excludes debug information from the output file: all input debug sections are excluded and both the symbol table and string table are removed. With debug information removed, the resulting Arm ELF file is smaller, but debugging at source level is impossible.

Note

Discarding debug information only affects the image size as loaded into the debugger—it has *no effect* on the size of the executable image that is loaded into the target.

See also

`--debug` on page 43

4.2.10 --no-entry

Summary

Set the entry point to zero.

Syntax

`--no-entry`

Description

The entry point in the Arm ELF file is set to zero. To link code into the final application, there must be at least one root symbol--see *--keep* on page 90.

Note

The entry point may be used by debuggers to configure applications ready for execution once downloaded. Removing the entry point may, therefore, require manual setup or scripting specific to the debugger to correctly configure the application for execution.

See also

--entry on page 44

4.2.11 --no-force-output

Summary

Do not write ELF file when link errors.

Syntax

`--no-force-output`

Description

This option instructs the linker not to write an ELF file in the presence of errors.

4.2.12 --minimal-section-headers

Summary

Produce minimal section header table.

Syntax

--minimal-section-headers

Description

This option produces an Arm ELF file containing minimal section information (section header string table and string table if symbols are required).

This is the default option for section headers.

See also

--bare on page 41, *--block-section-headers* on page 42, *--full-section-headers* on page 47

4.2.13 --no-symbols

Summary

Discard symbol table.

Syntax

`--no-symbols`

Description

This option removes the symbol table from the Arm ELF file.

See also

`--symbols` on page 56

4.2.14 --output

Summary

Set output file name.

Syntax

`--output filename`

`-o filename`

`--output=filename`

`-o=filename`

Description

This option sets the Arm ELF output filename, typically with extension “elf”.

4.2.15 --strip

Summary

Remove debug information and symbols.

Syntax

--strip

Description

This option eliminates all debug information, the symbol table, and the string table from the Arm ELF file, but does not remove the section table.

This option is equivalent to specifying `--no-debug` and `--no-symbols`.

See also

`--no-debug` on page 49, `--no-symbols` on page 53

4.2.16 --symbols

Summary

Include symbol table.

Syntax

`--symbols`

Description

This option includes the symbol table in the Arm ELF file.

See also

`--no-symbols` on page 53

4.3 Map file options

4.3.1 --map-all

Summary

Include all map file sections in the generated map file.

Syntax

`--map-all`

See also

--map-defaults on page 58, *--map-none* on page 65

4.3.2 --map-defaults

Summary

Set map file options to defaults.

Syntax

```
--map-defaults  
-M
```

Description

This option sets the following defaults:

```
--map-detail  
--map-init-table  
--no-map-listing  
--map-modules  
--map-narrow  
--map-placement  
--map-summary  
--map-symbols
```

See also

--map-all on page 57, *--map-none* on page 65

4.3.3 --map-detail

Summary

Include a detailed section breakdown in the generated map file.

Syntax

--map-detail

Example

```
*****
***                                     ***
***                               SECTION DETAIL                               ***
***                                     ***
*****

BSP_IP.o                                Code  RO Data  RW Data  ZI Data
      ENET_Receive_IRQHandler           88
      ENET_Transmit_IRQHandler           88
      ENET_Error_IRQHandler              88
      L_MergedGlobals                                     12
      -----
      File total:                264                                     12
      -----

system_MK66F18.o                       Code  RO Data  RW Data  ZI Data
      SystemInit                         2
      -----
      File total:                2
      -----

...more sections...

Grand total:                =====
                        3988                4                68                165
                        =====
```

See also

--no-map-detail on page 69

4.3.4 --map-file

Summary

Generate a linker map file.

Syntax

`--map-file filename`

`--Map filename`

`--map-file=filename`

`--Map=filename`

Description

Generates a linker map file to the given filename.

4.3.5 --map-init-table

Summary

Include an initialization table map section in the generated map file.

Syntax

--map-init-table

Example

```
*****
***
***          INITIALIZATION TABLE          ***
***
*****

Zero (__SEGGER_init_zero)
    1 destination range, total size 0xa5
      [0x1fff0444 to 0x1fff04e8]

Copy packing=copy (__SEGGER_init_copy)
    1 source range, total size 0x44 (100% of destination)
      [0x00000fd0 to 0x00001013]
    1 destination range, total size 0x44
      [0x1fff0400 to 0x1fff0443]

Copy packing=copy (__SEGGER_init_copy)
    1 source range, total size 0x10 (100% of destination)
      [0x00001014 to 0x00001023]
    1 destination range, total size 0x10
      [0x1fff04ea to 0x1fff04f9]

Totals
    Table size:      0x30 bytes
    Image size:      0x54 bytes
```

See also

--no-map-init-table on page 70

4.3.6 --map-listing

Summary

Include an absolute program listing in the generated map file.

Syntax

--map-listing

Example

```
*****
***
***          ABSOLUTE LISTING          ***
***
*****

;=====
; Section .init from thumb_crt0.o, alignment 4
;
_start:
0x0000051C  4924      LDR      R1, [PC, #0x90]
0x0000051E  4825      LDR      R0, [PC, #0x94]
0x00000520  1A0A      SUBS     R2, R1, R0
0x00000522  D002      BEQ      0x0000052A
0x00000524  2207      MOVS     R2, #7
0x00000526  4391      BICS     R1, R2
0x00000528  468D      MOV      SP, R1
0x0000052A  4923      LDR      R1, [PC, #0x8C]
0x0000052C  4823      LDR      R0, [PC, #0x8C]
0x0000052E  1A0A      SUBS     R2, R1, R0
0x00000530  D006      BEQ      0x00000540
0x00000532  2207      MOVS     R2, #7
0x00000534  4391      BICS     R1, R2
```

See also

--no-map-listing on page 71

4.3.7 --map-modules

Summary

Include a module breakdown in the generated map file.

Syntax

--map-modules
--map_modules

Example

```
*****
***                                     ***
***                                MODULE SUMMARY                                ***
***                                     ***
*****

-----
File      Code  RO Data  RW Data  ZI Data
-----
      BSP_IP.o      264
      system_MK66F18.o      2
      Kinetis_K60_Startup.o      68
      MK66F18_Vectors.o    1240
      thumb_crt0.o      388
      Main.o      128
                        4

...more modules...

      libc2.o (libc_v7em_fpv4_sp_d16_t_le_eabi.a)      40
libc2_asm.o (libc_v7em_fpv4_sp_d16_t_le_eabi.a)      84
                        132
Total:      3988      4      68      165
=====
```

See also

--no-map-modules on page 72

4.3.8 --map-narrow

Summary

Enable narrow name fields in map file.

Syntax

`--map-narrow`

See also

`--map-wide` on page 68

4.3.9 --map-none

Summary

Exclude all map file sections in the generated map file.

Syntax

`--map-none`

Description

This can be used to easily select a small subset of map sections, for example to include only symbols and a summary use:

```
--map-none --map-symbols --map-summary
```

See also

--map-all on page 57, *--map-defaults* on page 58

4.3.10 --map-placement

Summary

Include a section placement section in the generated map file.

Syntax

--map-placement

Example

```
*****
***                                     ***
***                               PLACEMENT SUMMARY                               ***
***                                     ***
*****
```

place at 0x00000000

Section	Type	Address	Size	Object
-----	----	-----	----	-----
.vectors	Code	00000000	0x410	MK66F18_Vectors.o

"RAM": place in [0x1fff0000 to 0x1fffffff] | [0x20000000 to 0x2002ffff]

Section	Type	Address	Size	Object
-----	----	-----	----	-----
.data.OS_Global	Init	1fff0400	0x40	OS_Global.o (OS.a)
.data.x	Init	1fff0440	0x4	Main.o
.bss..L_MergedGlobals	Zero	1fff0444	0xc	BSP_IP.o
.bss.OS_pTickHookRoot	Zero	1fff0450	0x4	OS_Global.o (OS.a)
.bss.OS_pMainContext	Zero	1fff0454	0x4	OS_Global.o (OS.a)

...more sections...

"FLASH": place in [0x00000000 to 0x001fffff]

Section	Type	Address	Size	Object
-----	----	-----	----	-----
.init	Code	00000410	0x44	Kinetis_K60_Startup.o
.init	Code	00000454	0xc8	MK66F18_Vectors.o
.init	Code	0000051c	0xcc	thumb_crt0.o
.rodata.OS_JLINKMEM_BufferSize	Cnst	000005e8	0x4	RTOSInit_K66F_CMSIS.o
.text	Code	000005ec	0x198	RTOS.o (OS.a)
.text.libc._execute_at_exit_fns	Code	000007b4	0x28	libc2.o (libc_v7em_fpv4_sp...
.text.libc.memcpy	Code	000007dc	0x54	libc2_asm.o (libc_v7em_fpv...
.text.main	Code	00000830	0x1c	Main.o

...more sections...

See also

--no-map-placement on page 73

4.3.11 --map-symbols

Summary

Include a symbol map section in the generated map file.

Syntax

--map-symbols

Example

```
*****
***                                     ***
***                               SYMBOL LIST                               ***
***                                     ***
*****
```

Symbols by value

Symbol	Value	Size	Type	Object
-----	-----	----	----	-----
_vectors	00000000	0x410	Code	Gb MK66F18_Vectors.o
__FLASH_segment_used_start__	00000000		----	Gb - Linker created -
Reset_Handler	00000411	0x44	Code	Gb Kinetis_K60_Startup.o
NMI_Handler	00000455	0xc8	Code	Wk MK66F18_Vectors.o
HardFault_Handler	00000457	0xc8	Code	Wk MK66F18_Vectors.o

...more symbols...

Symbols by name

Symbol	Value	Size	Type	Object
-----	-----	----	----	-----
ADC0_IRQHandler	000004ad	0xc8	Code	Wk MK66F18_Vectors.o
main	00000831	0x1c	Code	Gb Main.o

...more symbols...

4.3.12 --map-wide

Summary

Enable wide name fields in map file.

Syntax

`--map-wide`

See also

`--map-narrow` on page 64

4.3.13 --no-map-detail

Summary

Exclude a detailed section breakdown from the generated map file.

Syntax

`--no-map-detail`

See also

--map-detail on page 59

4.3.14 --no-map-init-table

Summary

Exclude an initialization table map section from the generated map file.

Syntax

`--no-map-init-table`

See also

--map-init-table on page 61

4.3.15 --no-map-listing

Summary

Exclude an absolute program listing from the generated map file.

Syntax

--no-map-listing

See also

--map-listing on page 62

4.3.16 --no-map-modules

Summary

Exclude a module breakdown from the generated map file.

Syntax

`--no-map-modules`

See also

--map-modules on page 63

4.3.17 --no-map-placement

Summary

Exclude a section placement section from the generated map file.

Syntax

`--no-map-placement`

See also

--map-placement on page 66

4.3.18 --no-map-symbols

Summary

Exclude a symbol map section from the generated map file.

Syntax

`--no-map-symbols`

See also

--map-symbols on page 67

4.4 Log file options

4.4.1 --log-file

Summary

Generate a linker log file.

Syntax

`--log-file filename`

`--log-file=filename`

Description

Generates a linker log file to the given filename.

4.5 Symbol and section options

4.5.1 --add-region

Summary

Add memory region.

Syntax

`--add-region region-name=size@addr`

Description

Add a region in addition to those defined in the linker script. This option allows you to write a generic linker script file for your selected toolset and to parameterize the target's memory regions (typically RAM and flash) from the command line.

It is possible to define a non-contiguous memory region from the command line using two `--add-region` options, for example:

```
--add-region RAM=64k@0x1fff0000 --add-region RAM=192k@0x20000000
```

is equivalent to the linker script fragment:

```
define region RAM = [0x1fff0000 size 64k] | [0x20000000 size 192k];
```

See also

define region statement on page 27

4.5.2 --autoat

Summary

Enable automatic placement of sections.

Syntax

`--autoat`

Description

When enabled, the linker places sections that conform to the "auto-at" naming convention at the designated position in the linked image.

See also

`--no-autoat` on page 97

4.5.3 --autokeep

Summary

Enable automatic retention of sections.

Syntax

--autokeep

Description

When enabled, the linker retains all initializer and finalizer sections in all input files so they can be selected and placed as appropriate for the runtime system. The linker defaults to autokeep enabled.

Enabling autokeep is identical to placing the following in the linker script:

```
keep { init_array, fini_array };
```

See also

--no-autokeep on page 98

4.5.4 --auto-arm-symbols

Summary

Enable automatic generation of Armlink symbols.

Syntax

`--auto-arm-symbols`

Description

Turns on automatic generation of Armlink symbols. The symbols defined in this mode are:

`B$$Start`

`B$$Limit`

`B$$Length`

where *B* is the name of a block declared using `define block`.

See also

`--no-auto-arm-symbols` on page 99

4.5.5 --auto-es-symbols

Summary

Enable automatic generation of Embedded Studio symbols.

Syntax

--auto-es-symbols

Description

Turns on automatic generation of Embedded Studio symbols. This option is equivalent to specifying **--auto-es-block-symbols** and **--auto-es-region-symbols**.

The symbols defined in this mode are:

```
__B_start__  
__B_end__  
__B_size__  
__B_start  
__B_end  
__B_size
```

where *B* is the name of a block declared using **define block**, and:

```
__R_segment_start__  
__R_segment_end__  
__R_segment_size__  
__R_segment_used_start__  
__R_segment_used_end__  
__R_segment_used_size__
```

where *R* is the name of a memory region created by a **define region** script command or by the **--add-region** command line option.

See also

--auto-es-block-symbols on page 81, *--auto-es-region-symbols* on page 82, *--no-auto-es-symbols* on page 100

4.5.6 --auto-es-block-symbols

Summary

Enable automatic generation of Embedded Studio block symbols.

Syntax

--auto-es-block-symbols

Description

The symbols defined in this mode are:

```
__B_start__  
__B_end__  
__B_size__  
__B_start  
__B_end  
__B_size
```

where *B* is the name of a block declared using **define block**.

See also

--no-auto-es-block-symbols on page 101

4.5.7 --auto-es-region-symbols

Summary

Enable automatic generation of Embedded Studio region symbols.

Syntax

--auto-es-region-symbols

Description

Turns on automatic generation of Embedded Studio region symbols. The symbols defined in this mode are:

```
__R_segment_start__  
__R_segment_end__  
__R_segment_size__  
__R_segment_used_start__  
__R_segment_used_end__  
__R_segment_used_size__
```

where *R* is the name of a memory region created by a **define region** script command or by the **--add-region** command line option.

See also

--no-auto-es-region-symbols on page 102

4.5.8 --define-symbol

Summary

Define linker symbol.

Syntax

`--define-symbol name=number | name`

`--defsym:name=number | name`

`--defsym=name=number | name`

`-defsym:name=number | name`

`-defsym=name=number | name`

`-Dname=number | name`

`-Dname=number | name`

Description

Define the symbol *name* as either a number or the value of an existing symbol.

4.5.9 --disable-lzss

Summary

Disable LZSS algorithm in auto packing selection.

Syntax

--disable-lzss

Description

When this option is specified, the LZSS compression algorithm is not a candidate algorithm when selecting a packing algorithm using `packing=auto`.

Note that disabling LZSS compression in this manner does not prevent it from being selected as a packing algorithm using **with `packing=lzss`** in an **initialize by copy** statement.

See also

--disable-packbits on page 85

4.5.10 **--disable-packbits**

Summary

Disable PackBits algorithm in auto packing selection.

Syntax

--enable-packbits

Description

When this option is specified, the PackBits compression algorithm is not a candidate algorithm when selecting a packing algorithm using `packing=auto`.

Note that disabling PackBits compression in this manner does not prevent it from being selected as a packing algorithm using **with** `packing=packbits` in an **initialize by copy** statement.

See also

--disable-packbits on page 85

4.5.11 --disable-zpak

Summary

Disable ZPak algorithm in auto packing selection.

Syntax

--disable-zpak

Description

When this option is specified, the SEGGER ZPak compression algorithm is not a candidate algorithm when selecting a packing algorithm using `packing=auto`.

Note that disabling ZPak compression in this manner does not prevent it from being selected as a packing algorithm using **with `packing=zipak`** in an **initialize by copy** statement.

See also

--enable-zpak on page 89

4.5.12 **--enable-lzss**

Summary

Enable LZSS algorithm in auto packing selection.

Syntax

--enable-lzss

Description

When this option is specified, the LZSS compression algorithm is a candidate algorithm when selecting a packing algorithm using `packing=auto`. This is the default.

See also

--disable-lzss on page 84

4.5.13 **--enable-packbits**

Summary

Enable PackBits algorithm in auto packing selection.

Syntax

--enable-packbits

Description

When this option is specified, the PackBits compression algorithm is a candidate algorithm when selecting a packing algorithm using `packing=auto`. This is the default.

See also

--disable-packbits on page 85

4.5.14 **--enable-zpak**

Summary

Enable ZPak algorithm in auto packing selection.

Syntax

--enable-zpak

Description

When this option is specified, the SEGGER ZPak compression algorithm is a candidate algorithm when selecting a packing algorithm using `packing=auto`. This is the default.

See also

--disable-zpak on page 86

4.5.15 --keep

Summary

Keep symbol.

Syntax

`--keep name`

`--keep=name`

Description

The linker keeps code and data reachable from root symbols, such as the entry point symbol, and discards all other sections. If an application image must contain some code or data (such as configuration or personalization data) that is not directly reachable from the root symbols, use this option to instruct the linker to treat additional symbols as roots.

4.5.16 --min-code-align

Summary

Set minimum code section alignment.

Description

Please see *--min-rx-align* on page 95.

4.5.17 --min-data-align

Summary

Set minimum data section alignment.

Syntax

`--min-data-align=value`

Description

This option overrides the alignment of all data section (read-only, read-write, zero-initialized) from input object files and libraries such that each section has a minimum alignment requirement of *value* bytes.

This options can be useful to force data that would be aligned on an odd byte address to an even or word-aligned address. Some example applications from silicon vendors have made assumptions regarding alignment of data and will fail to work correctly, usually failing with a misaligned load or store resulting in a hard fault. This is not a fault of the linker, which honors all alignment requirements, but increasing the alignment of all data to at least four bytes can quickly identify if a hard fault is caused by alignment assumptions.

See also

`--min-ro-align` on page 93, `--min-rw-align` on page 94, `--min-rx-align` on page 95

4.5.18 --min-ro-align

Summary

Set minimum read-only data section alignment.

Syntax

`--min-ro-align=value`

Description

This option overrides the alignment of all read-only data sections from input object files and libraries such that each section has a minimum alignment requirement of *value* bytes.

This options can be useful to force data that would be aligned on an odd byte address to an even or word-aligned address. Some example applications from silicon vendors have made assumptions regarding alignment of data and will fail to work correctly, usually failing with a misaligned load or store resulting in a hard fault. This is not a fault of the linker, which honors all alignment requirements, but increasing the alignment of all data to at least four bytes can quickly identify if a hard fault is caused by alignment assumptions.

See also

`--min-data-align` on page 92, `--min-rw-align` on page 94, `--min-zi-align` on page 96

4.5.19 --min-rw-align

Summary

Set minimum read-write data section alignment.

Syntax

`--min-rw-align=value`

Description

This option overrides the alignment of all read-write data sections from input object files and libraries such that each section has a minimum alignment requirement of *value* bytes.

This options can be useful to force data that would be aligned on an odd byte address to an even or word-aligned address. Some example applications from silicon vendors have made assumptions regarding alignment of data and will fail to work correctly, usually failing with a misaligned load or store resulting in a hard fault. This is not a fault of the linker, which honors all alignment requirements, but increasing the alignment of all data to at least four bytes can quickly identify if a hard fault is caused by alignment assumptions.

See also

`--min-data-align` on page 92, `--min-ro-align` on page 93, `--min-zi-align` on page 96

4.5.20 --min-rx-align

Summary

Set minimum code section alignment.

Syntax

```
--min-rx-align=value  
--min-code-align=value
```

Description

This option overrides the alignment of read-execute sections from input object files and libraries such that each section has a minimum alignment requirement of *value* bytes.

This options can be useful to force functions that would have two-byte alignment to be aligned on four-byte boundaries which may increase the effectiveness of flash accelerators and thereby increase execution speed (but at the expense of some wasted flash storage for the alignment).

It can also be used to increase the alignment of all functions to 16-byte boundaries that some flash accelerators benefit from.

4.5.21 --min-zi-align

Summary

Set minimum zero-initialized data section alignment.

Syntax

`--min-zi-align=value`

Description

This option overrides the alignment of all zero-initialized data sections from input object files and libraries such that each section has a minimum alignment requirement of *value* bytes.

This options can be useful to force data that would be aligned on an odd byte address to an even or word-aligned address. Some example applications from silicon vendors have made assumptions regarding alignment of data and will fail to work correctly, usually failing with a misaligned load or store resulting in a hard fault. This is not a fault of the linker, which honors all alignment requirements, but increasing the alignment of all data to at least four bytes can quickly identify if a hard fault is caused by alignment assumptions.

See also

`--min-data-align` on page 92, `--min-ro-align` on page 93, `--min-rw-align` on page 94

4.5.22 --no-autoat

Summary

Disable automatic placement of sections.

Syntax

`--no-autoat`

Description

Turns off automatic placement of sections which use the “auto-at” naming convention.

See also

`--autoat` on page 77

4.5.23 --no-autokeep

Summary

Disable automatic retention of sections.

Syntax

`--no-autokeep`

Description

The linker does not automatically keep initialization and finalization sections and it is the user's responsibility to keep and select them in the linker script.

See also

`--autokeep` on page 78

4.5.24 --no-auto-arm-symbols

Summary

Disable automatic generation of Armlink symbols.

Syntax

`--no-auto-arm-symbols`

Description

Turns off automatic generation of Armlink symbols.

See also

--auto-arm-symbols on page 79

4.5.25 **--no-auto-es-symbols**

Summary

Disable automatic generation of Embedded Studio symbols.

Syntax

--no-auto-es-symbols

Description

Turns off automatic generation of Embedded Studio symbols.

See also

--auto-es-symbols on page 80

4.5.26 **--no-auto-es-block-symbols**

Summary

Disable automatic generation of Embedded Studio block symbols.

Syntax

--no-auto-es-block-symbols

Description

Turns off automatic generation of Embedded Studio block symbols.

See also

--auto-es-block-symbols on page 81

4.5.27 **--no-auto-es-region-symbols**

Summary

Disable automatic generation of Embedded Studio region symbols.

Syntax

--no-auto-es-region-symbols

Description

Turns off automatic generation of Embedded Studio region symbols.

See also

--auto-es-region-symbols on page 82

4.5.28 **--no-unwind-tables**

Summary

Remove exception unwinding tables.

Syntax

--no-unwind-tables

Description

Remove all references to exception unwinding entries and do not create an unwinding table for stack unwinding. In this case it is not possible for C or C++ code to throw exceptions or unwind the stack.

See also

--unwind-tables on page 104

4.5.29 --unwind-tables

Summary

Include and generate exception unwinding tables.

Syntax

`--unwind-tables`

Description

Include all references exception unwinding entries and create an unwinding table for stack unwinding according to the Arm EABI. This is the default.

Notes

If there are no sections that have associated unwinding information, the table is empty and takes no space in the application even with this option selected.

See also

`--no-unwind-tables` on page 103

4.6 Control options

4.6.1 --cpu=name

Summary

Set target core or architecture.

Syntax

`--cpu=name`

`-cpu=name`

`-mcpu=name`

Description

This option selects the target processor for the application and controls the construction of appropriate veneers when required.

The core names accepted are:

- `cortex-m0`
- `cortex-m0plus`
- `cortex-m1`
- `cortex-m3`
- `cortex-m4`
- `cortex-m7`

The architecture names accepted are:

- `6-M`
- `7-M`
- `7E-M`

The default is `--cpu=cortex-m0`.

4.6.2 --list-all-undefs

Summary

Issue one error for each undefined symbol.

Syntax

--list-all-undefs

Description

This option instructs the linker to issue one error per undefined symbol which is the preferred option when running the linker inside an integrated development environment.

See also

--no-list-all-undefs on page 107

4.6.3 --no-list-all-undefineds

Summary

Issue one error covering all undefined symbols.

Syntax

--no-list-all-undefineds

Description

This option instructs the linker to issue a list of undefined symbols and a single error message indicating that there are undefined symbols. This is the default option for undefined symbols and is intended to produce clear output when the linker is run interactively from the command line.

See also

--list-all-undefineds on page 106

4.6.4 --no-warnings

Summary

Suppress warnings.

Syntax

--no-warnings

Description

This option disables all warning diagnostics issued by the linker. Although warnings are suppressed, the total number of warnings that are suppressed by the linker is shown at the end of linking:

```
C:> segger-ld --via=app.ind
Copyright (c) 2017-2018 SEGGER Microcontroller GmbH    www.segger.com
SEGGER Linker 2.14 compiled Apr 11 2018 10:50:34

Link complete: 0 errors, 1 warnings suppressed, 0 remarks

C:> _
```

See also

--warnings on page 115, *--warnings-are-errors* on page 116

4.6.5 --no-remarks

Summary

Suppress remarks.

Syntax

--no-remarks

Description

This option disables all remark diagnostics issued by the linker. Although remarks are suppressed, the total number of remarks that are suppressed by the linker is shown at the end of linking:

```
C:> segger-ld --via=app.ind
Copyright (c) 2017-2018 SEGGER Microcontroller GmbH    www.segger.com
SEGGER Linker 2.14 compiled Apr 11 2018 10:50:34

Link complete: 0 errors, 0 warnings, 2 remarks suppressed

C:> _
```

See also

--remarks-are-warnings on page 112, *--remarks-are-errors* on page 111

4.6.6 --remarks

Summary

Issue remarks.

Syntax

--remarks

Description

This option instructs the linker to issue remarks for potential issues during linking. This is the default.

See also

--no-remarks on page 109, --remarks-are-warnings on page 112, --remarks-are-errors on page 111

4.6.7 --remarks-are-errors

Summary

Elevate remarks to errors.

Syntax

`--remarks-are-errors`

Description

This option elevates all remark diagnostics issued by the linker to errors.

See also

`--no-remarks` on page 109, `--remarks` on page 110, , `--remarks-are-warnings` on page 112

4.6.8 --remarks-are-warnings

Summary

Elevate remarks to warnings.

Syntax

`--remarks-are-warnings`

Description

This option elevates all remark diagnostics issued by the linker to warnings.

See also

`--no-remarks` on page 109, `--remarks` on page 110, , `--remarks-are-errors` on page 111

4.6.9 --silent

Summary

Do not show output.

Syntax

`--silent`

`-q`

Description

This option inhibits all linker status messages; only diagnostic messages are shown.

See also

`--verbose` on page 114

4.6.10 --verbose

Summary

Increase verbosity.

Syntax

--verbose
-b

Description

This option increase the verbosity of the linker by one level.

See also

--*silent* on page 113

4.6.11 --warnings

Summary

Issue warnings.

Syntax

`--warnings`

Description

This option instructs the linker to issue warnings for dubious use or inputs. This is the default.

See also

`--no-warnings` on page 108, `--warnings-are-errors` on page 116

4.6.12 --warnings-are-errors

Summary

Elevate warnings to errors.

Syntax

`--warnings-are-errors`
`--fatal-warnings`

Description

This option elevates all warning diagnostics issued by the linker to errors.

See also

`--no-warnings` on page 108, `--warnings` on page 115

4.7 Compatibility options

4.7.1 --begin-group

Synopsis

Start input file group.

Syntax

```
--begin-group  
-(
```

Description

This option is accepted for GNU `ld` compatibility and is otherwise ignored. The SEGGER Linker will automatically resolve references from object files and libraries and does not require library grouping to do so.

4.7.2 --end-group

Synopsis

End input file group.

Syntax

```
--end-group  
-)
```

Description

This option is accepted for GNU ld compatibility and is otherwise ignored.

The SEGGER Linker will automatically resolve references from object files and libraries and does not require library grouping to do so.

4.7.3 --emit-relocs

Synopsis

Emit relocations.

Syntax

--emit-relocs

Description

This option is accepted for GNU ld compatibility and is otherwise ignored.

4.7.4 --allow-multiple-definition

Synopsis

Allow multiple definitions of the same symbol.

Syntax

--allow-multiple-definition

Description

This option is accepted for GNU ld compatibility and is otherwise ignored.

The SEGGER Linker does not support multiple definitions of identical symbols: a symbol is required to have exactly one strong definition.

4.7.5 --gc-sections

Synopsis

Garbage collect sections.

Syntax

--allow-multiple-definition

Description

This option is accepted for GNU ld compatibility and is otherwise ignored.

The SEGGER Linker only includes sections that are reachable from root symbols and, therefore, this option is redundant.

4.7.6 -EL

Synopsis

Little-endian byte ordering.

Syntax

-EL

Description

This option is accepted for GNU ld compatibility and is otherwise ignored.

The SEGGER Linker only links inputs and generates outputs with little-endian byte order.

4.7.7 --omagic

Synopsis

Set NMAGIC flag.

Syntax

`--omagic`

Description

This option is accepted for GNU ld compatibility and is otherwise ignored.

4.7.8 --discard-locals

Synopsis

Discard local symbols.

Syntax

--discard-locals

-X

Description

This option is accepted for GNU ld compatibility and is otherwise ignored.

Chapter 5

Indexes

5.1 Subject index

- add-region (linker option), 76
- allow-multiple-definition (linker option), 120
- auto-arm-symbols (linker option), 79
- auto-es-block-symbols (linker option), 81
- auto-es-region-symbols (linker option), 82
- auto-es-symbols (linker option), 80
- autoat (linker option), 77
- autokeep (linker option), 78
- bare (linker option), 41
- begin-group (linker option), 117
- block,
 - alignment, 28, 29
 - calculated size, 28, 29
 - inline, 19
 - input section ordering, 28
 - use with MPU, 29
- block-section-headers (linker option), 42
- command line,
 - add-region, 76
 - allow-multiple-definition, 120
 - auto-arm-symbols, 79
 - auto-es-block-symbols, 81
 - auto-es-region-symbols, 82
 - auto-es-symbols, 80
 - autoat, 77
 - autokeep, 78
 - bare, 41
 - begin-group, 117
 - block-section-headers, 42
 - cpu=name, 105
 - debug, 43
 - define-symbol, 83
 - disable-lzss, 84
 - disable-packbits, 85
 - disable-zpak, 86
 - discard-locals, 124
 - EL, 122
 - emit-relocs, 119
 - enable-lzss, 87
 - enable-packbits, 88
 - enable-zpak, 89
 - end-group, 118
 - entry, 44
 - force-output, 45
 - full-program-headers, 46
 - full-section-headers, 47
 - gc-sections, 121
 - keep, 90
 - list-all-undefineds, 106
 - load-program-headers, 48
 - log-file, 75
 - map-all, 57
 - map-defaults, 58
 - map-detail, 59
 - map-file, 60
 - map-init-table, 61
 - map-listing, 62
 - map-modules, 63
 - map-narrow, 64
 - map-none, 65
 - map-placement, 66
 - map-symbols, 67
 - map-wide, 68
 - min-code-align, 91
 - min-data-align, 92
 - min-ro-align, 93
 - min-rw-align, 94
 - min-rx-align, 95
 - min-zi-align, 96
 - minimal-section-headers, 52
 - no-auto-arm-symbols, 99
 - no-auto-es-block-symbols, 101
 - no-auto-es-region-symbols, 102
 - no-auto-es-symbols, 100
 - no-autoat, 97
 - no-autokeep, 98
 - no-debug, 49
 - no-entry, 50
 - no-force-output, 51
 - no-list-all-undefineds, 107
 - no-map-detail, 69
 - no-map-init-table, 70
 - no-map-listing, 71
 - no-map-modules, 72
 - no-map-placement, 73
 - no-map-symbols, 74
 - no-remarks, 109
 - no-symbols, 53
 - no-unwind-tables, 103
 - no-warnings, 108
 - omagic, 123
 - output, 54
 - remarks, 110
 - remarks-are-errors, 111
 - remarks-are-warnings, 112
 - script, 39
 - silent, 113
 - strip, 55
 - symbols, 56
 - unwind-tables, 104
 - verbose, 114
 - via, 40
 - warnings, 115
 - warnings-are-errors, 116
 - cpu=name (linker option), 105
- data,
 - thread-local, 24
- debug (linker option), 43
- define block statement, 28
- define memory statement, 26
- define region statement, 27
- define symbol statement, 30
- define-symbol (linker option), 83
- disable-lzss (linker option), 84
- disable-packbits (linker option), 85
- disable-zpak (linker option), 86
- discard-locals (linker option), 124
- do not initialize statement, 32
- EL (linker option), 122
- emit-relocs (linker option), 119
- enable-lzss (linker option), 87
- enable-packbits (linker option), 88
- enable-zpak (linker option), 89
- end-group (linker option), 118
- entry (linker option), 44
- force-output (linker option), 45
- full-program-headers (linker option), 46
- full-section-headers (linker option), 47
- gc-sections (linker option), 121
- initialize statement, 31
- keep (linker option), 90
- keep statement, 37
- list-all-undefineds (linker option), 106
- load-program-headers (linker option), 48
- log-file (linker option), 75
- map-all (linker option), 57
- map-defaults (linker option), 58
- map-detail (linker option), 59
- map-file (linker option), 60

- map-init-table (linker option), 61
- map-listing (linker option), 62
- map-modules (linker option), 63
- map-narrow (linker option), 64
- map-none (linker option), 65
- map-placement (linker option), 66
- map-symbols (linker option), 67
- map-wide (linker option), 68
- min-code-align (linker option), 91
- min-data-align (linker option), 92
- min-ro-align (linker option), 93
- min-rw-align (linker option), 94
- min-rx-align (linker option), 95
- min-zi-align (linker option), 96
- minimal-section-headers (linker option), 52
- no-auto-arm-symbols (linker option), 99
- no-auto-es-block-symbols (linker option), 101
- no-auto-es-region-symbols (linker option), 102
- no-auto-es-symbols (linker option), 100
- no-autoat (linker option), 97
- no-autokeep (linker option), 98
- no-debug (linker option), 49
- no-entry (linker option), 50
- no-force-output (linker option), 51
- no-list-all-undefineds (linker option), 107
- no-map-detail (linker option), 69
- no-map-init-table (linker option), 70
- no-map-listing (linker option), 71
- no-map-modules (linker option), 72
- no-map-placement (linker option), 73
- no-map-symbols (linker option), 74
- no-remarks (linker option), 109
- no-symbols (linker option), 53
- no-unwind-tables (linker option), 103
- no-warnings (linker option), 108
- omagic (linker option), 123
- output (linker option), 54
- place at statement, 35
 - place at address, 35
 - place at end, 36
 - place at start, 36
- place in statement, 33
- placement,
 - at fixed address, 20
 - by section name, 20
 - code in RAM, 21
 - preference order, 22
 - using auto-at, 20
- region, 14
 - intersection, 16
 - memory ranges, 14
 - repeated ranges, 14
 - subtraction, 16
 - union, 15
- remarks (linker option), 110
- remarks-are-errors (linker option), 111
- remarks-are-warnings (linker option), 112
- script (linker option), 39
- silent (linker option), 113
- statement,
 - define block, 28
 - define memory, 26
 - define region, 27
 - define symbol, 30
 - do not initialize, 32
 - initialize, 31
 - keep, 37
 - place at, 35
 - place in, 33
- strip (linker option), 55
- symbols (linker option), 56
- thread-local data, 24
- unwind-tables (linker option), 104
- verbose (linker option), 114
- via (linker option), 40
- warnings (linker option), 115
- warnings-are-errors (linker option), 116