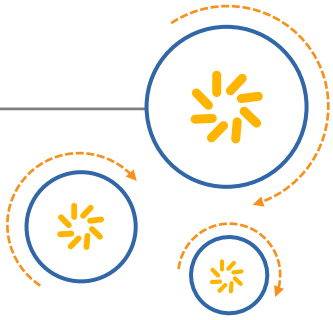
Qualcomm Technologies, Inc.



# Snapdragon ARM LLVM Linker

## User Guide

80-VB419-102 Rev. B

September 2, 2015

**Questions or comments:** developer.qualcomm.com/llvm-forum



Qualcomm Technologies, Inc.
5775 Morehouse Drive
San Diego, CA 92121
U.S.A.

# Contents


**1 Introduction ..... 6**

1.1 Overview ..... 6
1.2 Using the document ..... 7
1.3 Notation ..... 7
1.4 Feedback ..... 7

**2 Using the Linker ..... 8**

2.1 Overview ..... 8
2.2 Linker command ..... 9
2.3 Starting the linker ..... 9
2.4 Options ..... 10
2.5 Link-time optimization ..... 19
2.6 Known issues ..... 20

**3 Link Maps ..... 21**

3.1 Overview ..... 21
3.2 Using link maps ..... 22
3.3 Example link map file ..... 23
3.4 Archive section ..... 24
3.5 Common symbols section ..... 24
3.6 Linker script section ..... 25
3.7 Memory map section ..... 25

**4 Linker Scripts ..... 26**

4.1 Overview ..... 26
4.2 Using linker scripts ..... 27
4.3 Example script file ..... 28
4.4 Basic script syntax ..... 30
4.5 Script commands ..... 31
4.5.1 PHDRS ..... 32
4.5.2 SECTIONS ..... 34
4.5.3 ENTRY ..... 38
4.5.4 OUTPUT_FORMAT ..... 38
4.5.5 OUTPUT_ARCH ..... 38
4.5.6 SEARCH_DIR ..... 38

4.5.7 INCLUDE ..... 39
4.5.8 OUTPUT ..... 39
4.5.9 GROUP ..... 39
4.5.10 ASSERT ..... 39
4.6 Expressions ..... 40
4.7 Symbol assignment ..... 41
4.8 Linker script examples ..... 43
4.8.1 Exclude file in archive ..... 43
4.8.2 Exclude all files in archive ..... 44
4.8.3 Exclude multiple files ..... 45
4.8.4 Exclude archive and non-archive files ..... 46
4.8.5 Conflicting wildcards ..... 47

# Tables

Table 3-1 Link map sections ..... 22
Table 4-1 Linker script commands ..... 27
Table 4-2 Linker script expression functions ..... 40
Table 4-3 Linker script symbol assignments ..... 42

# 1 Introduction

## 1.1 Overview

The Snapdragon ARM linker merges object and archive files into executable images, relocating the program data to their final locations in memory, and resolving the symbol references both within and between files.

The linker accepts files built for ARM architecture and AArch64 architecture.

## 1.2 Using the document

This document contains four chapters:

- Chapter 1, *Introduction*, presents an overview of the linker.
- Chapter 2, *Using the Linker*, describes the linker command and options.
- Chapter 3, *Link Maps*, describes linker map files, which are optionally generated by the linker to provide information on how a program was linked.
- Chapter 4, *Linker Scripts*, describes linker script files, which are used to provide a detailed specification of how files are to be linked.

## 1.3 Notation

This document uses italics for terms and document names:

*executable object file*

*Snapdragon ARM LLVM Compiler User Guide*

Courier font is used for computer text:

```
ld.qcld -help
```

The following notation is used to define command syntax:

- Square brackets enclose optional items (e.g., [**label**]).
- **Bold** is used to indicate literal symbols (e.g., **[***comment***]**).
- The vertical bar character | is used to indicate a choice of items.
- Parentheses are used to enclose a choice of items (e.g., (**add**|**del**)).
- An ellipsis, . . . , follows items that can appear more than once.

## 1.4 Feedback

If you have any comments or suggestions on how to improve the linker (or this document), please send them to:

developer.qualcomm.com/llvm-forum

# 2 Using the Linker

## 2.1 Overview

The linker merges object and archive files into executable images, relocating the program data to their final locations in memory, and resolving the symbol references both within and between files.

This chapter covers the following topics:

- Linker command
- Starting the linker
- Linker options
- Link-time optimization
- Known issues

## 2.2 Linker command

To start the linker from a command line, type:

```
ld.qcld [option...] [input_object_file...]
```

The linker is invoked similarly to the GNU linker, and supports the most commonly-used options in the GNU linker.

Arguments that are specified on the command line without an option switch are treated as the names of input object files. If the linker does not recognize an input file as an object file, it treats the file as linker script (Chapter 4).

The linker can read its command arguments from a text file rather than from the command line. To do this, start the linker with the following command:

```
ld.qcld @file
```

In this command the linker reads its command arguments from the text file specified after the `@` symbol.

**NOTE** The command name `arm-link` is defined as a symbolic link to `ld.qcld`, and thus can be used as an alternate command name for starting the linker.

## 2.3 Starting the linker

The linker supports multiple target architectures, which are specified on the command line with the `-march` option.

The default target architecture is ARM. To explicitly specify this architecture, start the linker with the following command:

```
ld.qcld -march=arm other_arguments...
```

To link for the AArch64 architecture, the linker must be started with the following command:

```
ld.qcld -march=aarch64 other_arguments...
```

**NOTE** In the Snapdragon ARM LLVM compiler, the Clang driver recognizes the option `-fuse-ld=qcld` as a directive to use `ld.qcld` as the system linker.

## 2.4 Options

The linker accepts a large number of command options.

Option names are prefixed on the command line with a "-" character.

**NOTE** Any linker option name can be prefixed with either a single or double dash. Thus, `-shared` and `--shared` are considered identical.

To list all the linker options, type:

```
ld.qcld -help
```

The following linker options have one or more aliases defined for them. The aliases are used as short forms of the options:

**-Bdynamic** | **-dy** | **-call_shared**
**-Bshareable** | **-shared**
**-Bstatic** | **-dn** | **-non_shared** | **-static**
**--entry=***entry* | **-e** *entry*
**--export-dynamic** | **-E**
**--library-path=***searchdir* | **-L***searchdir*
**--output=***output* | **-o** *output*
**--print-map** | **-M**
**--relocatable** | **-r**
**--rpath=***pathname* | **-R** *pathname*
**--script=***file* | **-T** *file*
**-soname=***name* | **-h** *name*
**--strip-debug** | **-S**
**--trace=***type* | **-t=***type*
**--undefined=***symbol* | **-u** *symbol*
**-version** | **-V**

All of the linker options are listed below, along with their syntax and description.

**--allow-multiple-definition**
Allow multiple definitions of a symbol to exist in the files being linked. Only the first definition is used – the additional ones are ignored. Normally the linker flags multiple symbol definitions with an error. This option is disabled by default.

**--allow-shlib-undefined**
This option has no effect - it is provided for compatibility with the GNU linker.

**-add-needed**
**-no-add-needed**
Add a `DT_NEEDED` tag to every shared library specified on the command line. This option is enabled by default. `-no-add-needed` disables the option.

**--as-needed**
**--no-as-needed**
Limit the adding of DT_NEEDED tags to libraries that satisfy a symbol reference (from regular objects) which is undefined when the library was linked, or – if the library is not found in the DT_NEEDED lists of the other libraries linked up to that point – a reference from another shared library. This option is disabled by default. --as-needed enables the option.

This option affects the ELF DT_NEEDED tags for shared libraries specified on the command line after the --as-needed option. Normally the linker adds a DT_NEEDED tag for every shared library specified on the command line.

**-Bdynamic**
**-dy**
**-call_shared**
Link against dynamic libraries. This option can be used multiple times – it affects the library searching performed by subsequent occurrences of the -l option.

**-Bgroup**
Instruct the dynamic linker to perform lookups only inside the group.

**-Bshareable**
**-shared**
Create shared library.

**-Bstatic**
**-dn**
**-non_shared**
**-static**
Link against static libraries.

**-Bsymbolic**
Bind global symbol references to the symbol definition in a shared library.

**-build-id=***style*
This option has no effect – it is provided for compatibility with the GNU linker.

**-cref**
Generate cross-reference table. The table is written to the standard output.

**-d**
Allocate space for common symbols. This is done even if the output is specified as relocatable (using the -r option).

**-default-script=***file*
Specify the default linker script. The specified linker script is loaded after all other linker options are processed.

**--defsym** *symbol=expression*
Create a global symbol in the output file, containing the absolute address given by *expression*. This option can be used multiple times. The expression is limited to a hexadecimal constant or existing symbol name, with the optional addition or subtraction of a second hexadecimal constant or symbol.

**-discard-all**
Delete all local symbols from the output file.

**-discard-locals**
Delete all temporary local symbols from the output file.

**--dynamic-linker=***file*
Specify dynamic linker.

**--dynamic-list=***file*
Add symbols to the dynamic symbol table of a dynamic executable. The specified text file contains one or more lists of symbol names, with each list having the following form:

```
{
  symbol1;
  symbol2;
  symbol3;
};
```

This option can be used multiple times.

**-e** *entry*
**--entry=***entry*
Use the specified symbol as the program entry point.

**-exclude-libs=***lib1*,*lib2*,...
Exclude the specified libraries from automatic export.

**--export-dynamic**
**-E**
Export all dynamic symbols. By default the only dynamic symbols exported are those referenced by dynamic objects that are explicitly specified during linking.

This option automatically enables `--force-dynamic`.

**-eh-frame-hdr**
Create the ELF segment header "`PT_GNU_EH_FRAME`" and the section ".`eh_frame_hdr`".

**-emit-gnu-compat-relocs**
Generate GNU linker-compatible relocations, which use the relocation target address as the VMA of the executable (instead of the offset within sections). When this option is enabled, `-emit-relocs` is automatically enabled. This option is disabled by default.

**-emit-relocs**
Leave relocation sections and entries in executables. This option is disabled by default.

**-extern-list=***file*
Add symbols to the external symbol table of the output file. The specified text file contains one or more lists of symbol names, with each list having the following form:

```
{
  symbol1;
  symbol2;
  symbol3;
};
```

This option can be used multiple times.

**-fatal-warnings**
**-no-fatal-warnings**
Convert warnings into fatal errors. `-no-fatal-warnings` restores the default behavior.

**-filetype=**(**obj**|**dso**|**exe**)
Output file type (not all types are supported on all targets).

**obj**

Generate relocatable object file (`.o`)

**dso**

Generate dynamic shared object file (`.so`)

**exe**

Generate executable object file (`.exe`)

**-fini=***function*
Specify the function called when an executable or shared object is unloaded, by setting `DT_FINI`. The default function name is `_fini`.

**-fix-cortex-a8**
**-no-fix-cortex-a8**
This option has no effect – it is provided for compatibility with the GNU linker.

**-flto**
Enable link-time optimization (LTO). For more information see Section 2.5.

**NOTE** When an LLVM bitcode object is specified as an input file, the linker automatically enables LTO even if this option is not specified.

**-flto-options=(codegen=***option*`[,`*option*`...]`|**preserve=***symbol***)**
Specify options that affect link-time optimization (LTO).

**codegen=***option*`[,`*option*`...]`

Pass the specified `-mllvm` compiler options to the compiler during LTO. For example, `-flto-options=codegen=-arm-opt-memcopy`

**preserve=***symbol*

Preserve the specified symbol during LTO.

**--force-dynamic**
Force the output file to include dynamic sections.

Force the linker to create dynamic sections. This makes a dynamic executable.

**-fPIC**
Set relocation model to PIC. Equivalent to `-relocation-model=pic`.

**-g**
Not used.

**-G** *size*
**-gpsize** *size*
Not used.

**--gc-sections**
**--no-gc-sections**
Delete all unused input sections from the output file (*garbage collection*).

Sections are not considered unused if they contain the entry symbol, undefined symbols, or symbols used with dynamic objects or shared libraries.

The deleted sections can be displayed with `--print-gc-sections`.

**-h** *name*
**-soname=***nam*
Set the internal name of a shared library, by setting `DT_SONAME`.

**-hash-size=***size*
This option has no effect – it is provided for compatibility with the GNU linker.

**-hash-style=**(**sysv**|**gnu**|**both**)
Specify hash table type used in the linker.

**sysv**

Classic ELF `.hash` section

**gnu**

New-style GNU `.gnu.hash` section

**both**

Both ELF and GNU hash tables

**--help**
Print list of linker command options.

**-init=***function*
Specify the function called when an executable or shared object is loaded, by setting `DT_INIT`. The default function name is `_init`.

**-l***archive*
Add the specified archive file to the list of files to link. This option can be used multiple times.

**-L***searchdir*
**--library-path=***searchdir*
Add the specified pathname to the list of paths used to search for libraries and scripts. This option can be used multiple times.

**-m=***emulation*
This option has no effect – it is provided for compatibility with the GNU linker.

**-M**
**--print-map**
Generate link map file which provides information on how the object files were linked. For more information see Chapter 3.

**-Map=***mapfile*
Generate link map file with the specified file name.

**-march=**[**arm**|**aarch64**]
Specify the target architecture. The default is `arm`.

**-mcpu=***cpu-name*
Specify the ARM/AArch64 processor version. When `-march` is used, this option is ignored.

**-merge-strings**
Remove duplicate instances of character strings from the output file. This option is enabled by default.

**-mtriple=***triple*
Specify the architecture, ABI, and operating system of the linker output file (for example, `-mtriple=arm-gnu-linux`). When this option is used, the `-march` option must be specified with a null option value (i.e., "`-march=`").

**NOTE** `-mtriple` is not recommended. Instead, use `-march` to specify the target architecture.

**-noinhibit-exec**
Do not delete output file after linker error.

**-nostdlib**
Search only the library directories that are specified on the command line.

**-no-trampolines**
Do not add trampolines to the linked code. Anything that requires a trampoline will be considered an error.

**-no-undefined**
Do not allow unresolved references in the linked code.

**-no-warn-mismatch**
Allow linking together mismatched input files.

**-o** *output*
**--output=***output*
Specify the linker output file. The default name is `a.out`.

**-p**
This option has no effect – it is provided for compatibility with the ARM linker.

**-pie**
Generate a position-independent executable file.

**--print-gc-sections**
Display all sections that were removed during linking by garbage collection. This option should be used with `--gc-sections`.

**-Qy**
This option is ignored for SVR4 compatibility.

**--relocatable**
**-r**
Create a relocatable object. The default is an absolute file.

**-relocation-model=**(**default**|**static**|**pic**|**dynamic-no-pic**)
Specify relocation model.

**default**

Target default relocation model

**static**

Non-relocatable code

**pic**

Fully relocatable, position-independent code

**dynamic-no-pic**

Relocatable external references, non-relocatable code (not supported)

**NOTE** `-relocation-model` is parsed only, and has no effect on linking.

**-rosegment**
Disable the merging of read-only sections with read/executable sections.

**--rpath-link=***pathname*
This option has no effect – it is provided for compatibility with the GNU linker.

**-R** *pathname*
**--rpath=***pathname*
Add the specified pathname to the search path for the runtime library.

**-save-temps**
Preserve the assembler file in link-time optimization (LTO).

The option `-t=lto` can be used to display the pathname of the file.

**--strip-all**
Do not include any symbol information in the output file.

**-S**
**--strip-debug**
Do not include debugger symbols in the output file.

**--section-start** *section=org*
Set the start address of the specified section. The value *org* must be a hexadecimal integer. This option can be repeated multiple times.

**-sysroot**=*pathname*
Specify the system root directory, overriding the configure-time default. This option is useful only in Linux.

**-t=***type*
**--trace=***type*
Display trace information indicating how the files are being linked. The following types of information can be displayed. This option can be repeated multiple times.

**command-line**

Display linker options specified on the command line.

**file**

Display files specified for linking.

**garbage-collection**

Display garbage collection performed during linking.

**lto**

Display additional trace information related to link-time optimization (LTO).

**symbols**

Display program symbols processed by the linker, and how they are resolved.

**sym-reloc**

Display symbol relocation performed by the linker.

**trampolines**

Display details on trampolines created by the linker.

**-T** *file*
**--script=***file*
Specify the linker script file (Chapter 4). The order that the linker script is loaded in (relative to when the other linker options are processed) is determined by the order that the options are specified on the command line.

**-Tbss=***address*
Specify the address of the bss segment.

**-Tdata=***address*
Specify the address of the data segment.

**-Ttext=***address*
Specify the address of the text segment.

**-u** *symbol*
**--undefined=***symbol*
Include the specified symbol in the output file as an undefined symbol. This is typically done to force additional object files to be linked into a program. This option can be repeated multiple times.

**-verbose=**(**1**|**2**)
Display linker-related information, including the linker release version. The option argument specifies how much information is displayed.

**-V**
**-version**
Display the linker release version.

**-version-script**
This option has no effect - it is provided for compatibility with the GNU linker.

**--whole-archive**
**--no-whole-archive**

Link into the program every object file that is contained in the specified archives.

Archives are specified by appearing between the `--whole-archive` and `--no-whole-archive` options.

**NOTE** `--whole-archive` is parsed only, and has no effect on linking.

**--wrap=***symbol*

Create wrapper for the specified symbol. Resolve any undefined references to the symbol as references to the symbol `__wrap_`*symbol*, and any undefined references to the symbol `__real_`*symbol* as references to *symbol*. This option can be repeated multiple times.

**-warn-common**

Warn when a common symbol is combined with another common symbol or with a symbol definition.

**NOTE** `-warn-common` is parsed only, and generates no warning messages.

**-warn-shared-textrel**
**-no-warn-shared-textrel**

Warn when an object compiled without the `-fpic` option is added to a shared library.

**-Y=***default-search-path*

Add the specified path to the default library search path.

**-z** *keyword*

This option has no effect – it is provided for compatibility with the GNU linker.

**-(** *archive* **... -)**
**--start-group=***archive* **... --end-group**

Resolve circular symbol references in the specified files and libraries. Because the linker must repeatedly search the libraries to resolve circular references, this option significantly affects the linker performance – it should only be used when circular symbol references are present.

Files and libraries are specified by appearing between the `-(` and `-)` options, or the `--start-group` and `--end-group` options.

## 2.5 Link-time optimization

Link-time optimization (LTO) is a type of code optimization which is performed by the linker. In LTO the linker drives the compiler to perform inter-procedure optimizations and apply global program information to improve the code optimizations performed by the compiler.

Figure 2-1 shows the flow of LTO as it is performed by the linker.

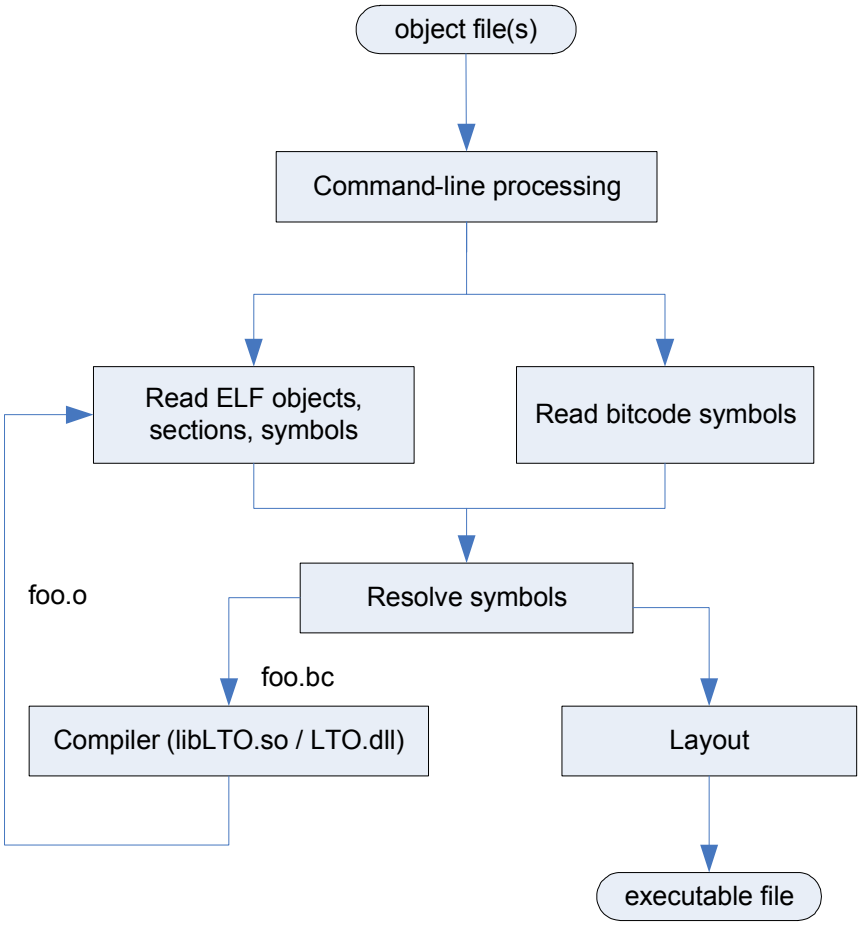



**Figure 2-1 LTO flow diagram**

LTO can be performed by linking with the `-flto` option (Section 2.4). However, if any of the linker inputs are in LLVM bitcode format, the linker automatically enables LTO even if this option is not specified.

To achieve the best results from LTO, all the linker input files should be in LLVM bitcode format. This can be done by compiling all the source files with the `-flto` compiler option.

**NOTE** For more information on compiling files for LTO, see the *Snapdragon ARM LLVM Compiler User Guide*.

When performing LTO the linker must be able to invoke the compiler. This requires the following system configuration:

- **Linux** – The library file `libLTO.so` (which is the Linux shared library that contains the compiler) is linked with the linker.
- **Windows** – The library file `LTO.dll` (which is the Windows shared library that contains the compiler) must be stored in the same directory as the linker executable (`ld.qcld`).

**NOTE** On both platforms this system configuration is performed by default as part of the installation, so no explicit setup steps are required.

The linker option `-flto-options=codegen` can be used to pass the `-mllvm` options to the compiler as part of LTO. Because the compiler in LTO is being driven by the linker and not Clang, any options defined in Clang are not supported.

By default the linker uses the LLVM integrated assembler to generate bit code directly to a binary object file. To preserve the assembler file, use the linker option `-save-temps` (along with the option `-t=lto` to show where the file gets stored).

## 2.6 Known issues

The current version of the linker has the following known issues:

- It cannot handle input object files that contain more than 65280 sections.
- When linking ARM object files compiled with `-O0`, C++ exception handling may fail due to a known bug in the `ARM.exidx` section generation.

These issues will be fixed in the next release of the linker.

# 3 Link Maps

## 3.1 Overview

Link maps are optionally generated by the linker. They provide the following information on how the files were linked:

- Archive and object files accessed
- Common symbols allocated
- Linker script (if specified)
- Memory map of sections and symbols

## 3.2 Using link maps

A link map is generated as a text file. The file can be specified on the command line with either the `-Map` or **`--print-map`** option. For more information see Section 2.4.

Link maps are divided into four sections: the archive section, the common symbols section, the linker script section, and the memory map section.

Table 3-1 lists the link map sections and the format for each section entry.

**Table 3-1 Link map sections**

| Section | Section Entry Format |
|---|---|
| Archive | *archive_file* (*symbol_define_object_file*)<br>*symbol_reference_object_file* (*symbol*) |
| Common symbols | *symbol size archive_file* (*symbol_define_object_file*) |
| Linker script | *linker_script_command* |
| Memory map | *output_section addr size*<br>*input_section addr size object_file*<br>*symbol addr*<br>...<br>... |

**NOTE** Link map section entries are described in detail in the following sections.

## 3.3 Example link map file

The following code example shows a typical link map file. It was generated by linking the following C program for the AArch64 target architecture:

```
#include "stdio.h"
int common_var;

int main() {
  common_var = 1;
  printf("var = %d\n", common_var);
  return 0;
}
```

**NOTE** The map file has been shortened for readability.

```
Archive member included because of file (symbol)
/sysroot/aarch64-linux-gnu/libc/usr/lib64/libc_nonshared.a(elf-init.oS)
        /sysroot/aarch64-linux-gnu/libc/usr/lib/../lib64/crt1.o (__libc_csu_init)

Allocating common symbols
Common symbolsizefile

common_var0x4/tmp/t-57e15d.o

Linker Script and memory map
LOAD /sysroot/aarch64-linux-gnu/libc/usr/lib/../lib64/crt1.o
LOAD /sysroot/aarch64-linux-gnu/libc/usr/lib/../lib64/crti.o
LOAD /sysroot/lib/gcc/aarch64-linux-gnu/4.9.0/crtbegin.o
LOAD /tmp/t-57e15d.o
LOAD /sysroot/lib/gcc/aarch64-linux-gnu/4.9.0/../../../../aarch64-linux-
gnu/lib/../lib64/libgcc_s.so
START GROUP
LOAD /sysroot/aarch64-linux-gnu/libc/lib64/libc.so.6
LOAD /sysroot/aarch64-linux-gnu/libc/usr/lib64/libc_nonshared.a(elf-init.oS)
LOAD /sysroot/aarch64-linux-gnu/libc/lib/ld-linux-aarch64.so.1
END GROUP
SKIPPED /sysroot/lib/gcc/aarch64-linux-gnu/4.9.0/../../../../aarch64-linux-
gnu/lib/../lib64/libgcc_s.so (ELF)
LOAD /sysroot/lib/gcc/aarch64-linux-gnu/4.9.0/crtend.o
LOAD /sysroot/aarch64-linux-gnu/libc/usr/lib/../lib64/crtn.o

Linker scripts used (including INCLUDE command)
/sysroot/aarch64-linux-gnu/libc/usr/lib/../lib64/libc.so

__ehdr__0x00x40

__pHdr__0x400x150

.interp0x1900x1b

…

.text0x4680x21c
.text0x4680x48/sysroot/aarch64-linux-gnu/libc/usr/lib/../lib64/crt1.o
    0x468   .text
    0x468   $x
    0x468   _start
    0x498   $d
...
```

```
.text0x5c00x48/tmp/t-57e15d.o
    0x5c0   $x.0
    0x5c0   .text
    0x5c0   main
…

.shstrtab0x12880xf1

.symtab0x13800x7f8

.strtab0x1b780x351
...
```

## 3.4 Archive section

The archive section of a link map lists each archive file that was accessed by the linker, along with the symbol reference that caused the archive file to be accessed.

Each entry in the archive section contains the following items:

- The full pathname of the archive file accessed by the linker
- The name of the archived object file that defines the symbol (in parentheses)
- The full pathname of the object file that contains the symbol reference
- The name of the referenced symbol (in parentheses)

In the following example, the symbol `__libc_csu_init` is referenced in object file `crt1.o` and defined in `_elf-init.oS`, which in turn is stored in archive file `libc_nonshared.a`:

```
/sysroot/aarch64-linux-gnu/libc/usr/lib64/libc_nonshared.a(elf-init.oS)
       /sysroot/aarch64-linux-gnu/libc/usr/lib/../lib64/crt1.o (__libc_csu_init)
```

## 3.5 Common symbols section

The common symbols section of a link map lists the common symbols that were allocated in memory by the linker.

Each entry in the common symbols section contains the following item:

- The name of the symbol
- The size of the memory area allocated for the symbol
- The full pathname of the archive file accessed by the linker
- The name of the archived object file that defines the symbol (in parentheses)

In the following example, the common symbol `common_var` has size `0x4` and is defined in object file `t-57e15d.o`, which in turn is compiled from the C code shown in Section 3.3:

```
common_var  0x4     /tmp/t-57e15d.o 3
```

## 3.6 Linker script section

The linker script section of a link map lists the complete linker script that was specified for the link. For more information on linker scripts see Chapter 4.

**NOTE** Linker scripts are optional – if a script is not specified on the linker command line, the link map will not include a linker script.

The following example shows the initial lines of a linker script section:

```
…
START GROUP
LOAD /sysroot/aarch64-linux-gnu/libc/lib64/libc.so.6
LOAD /sysroot/aarch64-linux-gnu/libc/usr/lib64/libc_nonshared.a(elf-init.oS)
LOAD /sysroot/aarch64-linux-gnu/libc/lib/ld-linux-aarch64.so.1
END GROUP
SKIPPED /sysroot/lib/gcc/aarch64-linux-gnu/4.9.0/../../../../aarch64-linux-
gnu/lib/../lib64/libgcc_s.so (ELF)
LOAD /sysroot/lib/gcc/aarch64-linux-gnu/4.9.0/crtend.o
LOAD /sysroot/aarch64-linux-gnu/libc/usr/lib/../lib64/crtn.o

Linker scripts used (including INCLUDE command)
/sysroot/aarch64-linux-gnu/libc/usr/lib/../lib64/libc.so
```

## 3.7 Memory map section

The memory map section of a link map lists how symbols and assembly language sections are assigned to memory in the output file.

The memory map section lists one or more output sections in the order they are assigned by the linker. Each entry in the memory map section contains the following items:

- The output section (including its start address and section size)
- Each input section that is mapped to the output section (including its start address, section size, and the full pathname of the object file containing the section)
- Each symbol defined in the input section (including its assigned value)

In the following example the output section `.text` has start address `0x468` and size `0x21c`. Multiple input sections (named `.text`) are mapped to this output section – two are shown below. The first has start address `0x468` and size `0x48`, and the second has start address `0x5c0` and size `0x48`. Following each section descriptor is a list of the symbols in the section.

```
.text0x4680x21c
.text0x4680x48/sysroot/aarch64-linux-gnu/libc/usr/lib/../lib64/crt1.o
    0x468   .text
    0x468   $x
    0x468   _start
    0x498   $d
…

.text0x5c00x48/tmp/t-57e15d.o
    0x5c0   $x.0
    0x5c0   .text
    0x5c0   main
…
```

# 4 Linker Scripts

## 4.1 Overview

Linker scripts are used to provide a detailed specification of how files are to be linked. They offer greater control over linking than is available using just the linker command options (Section 2.4).

Linker scripts control the following properties:

- ELF program header
- Program entry point
- Control input and output files and searching paths
- Section memory placement and runtime properties
- Section removal
- Define symbols

**NOTE** Linker scripts are optional – in most cases the linker's default behavior is sufficient (with input sections merged according to their section names, and segments ordered by sections sharing similar permissions).

## 4.2 Using linker scripts

A linker script consists of a sequence of commands stored in a text file. The script file can be specified on the command line either with the `-T` option, or by specifying the file as one of the input files. The linker is able to distinguish between script files and object files, and handles each accordingly.

**NOTE** To generate a map file which shows how a linker script controlled linking, use the `-M` option.

Table 4-1 lists the commands that are supported in linker script files.

**Table 4-1 Linker script commands**

| Command | Description |
|---|---|
| PHDRS | ELF program header definition |
| SECTIONS | Section mapping and memory placement |
| ENTRY | Program execution entry point |
| OUTPUT_FORMAT | Parsed, but no effect on linking |
| OUTPUT_ARCH | Parsed, but no effect on linking |
| SEARCH_DIR | Add additional searching directory for libraries |
| INCLUDE | Include linker script file |
| OUTPUT | Define output filename |
| GROUP | Define files that will be searched repeatedly |
| ASSERT | Linker script assertion |

**NOTE** The syntax used in linker scripts is similar to the scripts used with the GNU linker. However, only the features and syntax described in this document are supported.

## 4.3 Example script file

The following example shows a linker script which demonstrates the scripting features:

```
ENTRY (main)
SEARCH_DIR("./")

PHDRS
{
CODE PT_LOAD ;
DATA PT_LOAD ;
}

SECTIONS
{
  .text.qcldfn (0x2000) : { *(.text.qcldfn*) } : CODE
  . = ALIGN(0x1000);
  PROVIDE(__etext = .);
  __text_start = . + 0x1000 - 0x1000;
  .text : { EXCLUDE_FILE(*notused.o*) *(.text.*) } : CODE
  .data : { *(.data.*) } : DATA
  .init : { KEEP (*(.init)) }
  . = SEGMENT_START(".bss", 0x80000);
  .bss : { *(.bss.*) }
  __bss_start = .;
  __bss_end = .;

}
```

The script contains several commands which specify various link properties. The `PHDRS` command defines two loadable ELF segment, while the `SECTIONS` command specifies how input sections are mapped to output sections, and where output sections are located in memory.

Wildcard characters in the `SECTIONS` command indicate that multiple input sections are mapped to a single output section. A period (`.`) is used to indicate the current location counter – note that it is assigned several different values in the `SECTIONS` command.

### .text.qcldfn

All input sections whose names begin with `.text.qcldfn` are mapped to an output section named `.text.qcldfn`. This output section is put into the first place in segment `CODE` and the merged output section is located at the virtual memory address 0x2000.

The current location counter is then advanced to the next 0x1000 address boundary past the end of the output section `.text.qcldfn` (using the `ALIGN` directive).

The symbol `__etext` is conditionally defined with the current location counter value if any unresolved symbol references exist for the symbol (using the `PROVIDE` command).

The symbol `__text_start` is assigned the current location counter value plus the following expression. (Arithmetic expressions can be used in linker script statements.)

### .text

All input sections that begin with `.text` are mapped to the output section `.text`. It is put into the segment `CODE`, and the merged output section is located at the current location counter.

**NOTE** The previously-referenced section `.text.qcldfn` is not affected by this statement (even though it matches the section name wildcard `.text.*`) because it was already merged in the previous link script statement.

### .data

Next, the current location counter is assigned the base address of the output section `.data` (using the `SEGMENT_START` directive). If this section is not defined, the default value of 0x50000 is used instead.

All input sections whose names begin with `.data` are mapped to the output section `.data`. This section is put into segment `DATA`, and the merged output section is located at the current location counter (as specified by the preceding statement in the script).

### .init

All input sections whose names begin with `.init` are mapped to the output section `.init`. Because no segment is defined, it uses the previous segment if available. In this script, it is put into the segment `DATA`, and the merged output section is located at the current location counter (as specified by the preceding statement in the script).

### .bss

Next, the current location counter is assigned the base address of the output section `.bss` (using the `SEGMENT_START` directive). If this section is not defined, the default value of 0x80000 is used instead.

All input sections whose names begin with `.bss` are mapped to the output section `.bss`, and the merged output section is located at the current location counter (as specified by the preceding statement in the script).

In addition, none of the input sections will ever be removed from memory if garbage collection is enabled (as specified by the `KEEP` directive).

Finally, the symbols `__bss_start` and `__bss_end` are both assigned the current location counter value.

## 4.4 Basic script syntax

This section covers the basic syntax for link scripts:

- Symbols
- Comments
- Strings
- Expressions
- Location counter
- Symbol assignment

### Symbols

Symbol names must begin with a letter, underscore, or period. They can include letters, numbers, underscores, hyphens, or periods.

### Comments

Comments can appear in linker scripts:

```
/* comment */
```

### Strings

Character strings can be specified as parameters with or without delimiter characters.

### Expressions

Expressions are similar to C, and support all C arithmetic operators. They are evaluated as type `long` or `unsigned long`. For more information see Section 4.6.

### Location counter

A period ( `.` ) is used as a symbol to indicate the current location counter – it is used only in the `SECTIONS` command, where it designates locations in the output section:

```
. = ALIGN(0x1000);
. = . + 0x1000;
```

Assigning a value to the location counter symbol changes the location counter to the specified value. The location counter can be moved forward by arbitrary amounts to create gaps in an output section. However, it cannot be moved backwards.

### Symbol assignment

Symbols (including the location counter) can be assigned constants or expressions:

```
__text_start = . + 0x1000;
```

Assignment statements are similar to C, and support all C assignment operators. They must be terminated with a semicolon ( `;` ).

## 4.5 Script commands

This section describes the commands supported in linker scripts:

- PHDRS
- SECTIONS
- ENTRY
- OUTPUT_FORMAT
- OUTPUT_ARCH
- SEARCH_DIR
- INCLUDE
- OUTPUT
- GROUP
- ASSERT

The `SECTIONS` command must be specified in a linker script – all the other script commands are optional.

An example linker script (which demonstrates the use of several commands) is presented in Section 4.3.

Expressions can be used in several linker script commands. For more information on expressions see Section 4.6.

### 4.5.1 PHDRS

```
PHDRS
{
name type [FILEHDR] [PHDRS] [AT (address)] [FLAGS (flags)]
}
```

Script command which sets information in the program header of an ELF output file.

*name* is used to specify the program header in the SECTIONS command (Section 4.5.2).

*type* specifies the program header type:

- PT_LOAD – Loadable segment
- PT_NULL – Linker does not include section in a segment
- PT_DYNAMIC – Segment where dynamic linking information is stored
- PT_INTERP – Segment where the name of the dynamic linker is stored
- PT_NOTE – Segment where note information is stored
- PT_SHLIB – Reserved program header type
- PT_PHDR – Segment where program headers are stored

**NOTE** No loadable section should be set to PT_NULL.

The options FILEHDR, PHDRS, and AT are not supported – they are parsed but otherwise have no effect on linking.

The FLAGS option specifies the p_flags field in the ELF header. The following values can be used:

- PF_R – Read
- PF_W – Write
- PF_X – Execute

Multiple values can be specified in p_flags. For example, the value "PF_R | PF_W" specifies the flag setting "read/write".

**NOTE** If the sections in an output file have different flag settings than what is specified in PHDRS, the linker chooses the least-restrictive settings for the output file.

More than one program header specification can be assigned to a given section. For example, the following linker script generates a linker error indicating that the same section cannot be included in two different segments:

```
PHDRS {
phdr1 PT_LOAD;
phdr2 PT_LOAD;
}
.text : {
*(*.text*)
} : phdr1 :phdr2
```

**NOTE** The `PHDRS` command overrides the linker's default program header settings.

For multiple program headers, only the first one can have an LMA or VMA definition.

## 4.5.2 SECTIONS

```
SECTIONS
{
  section_statement
  section_statement
  ...
}
```

Script command which specifies how input sections are mapped to output sections, and where output sections are located in memory.

**NOTE** The `SECTIONS` command must be specified once – and only once – in a linker script.

An example linker script (which demonstrates the use of the `SECTIONS` command) is presented in Section 4.3.

### 4.5.2.1 Section statements

A `SECTIONS` command contains one or more *section statements*, each of which can be one of the following:

- An `ENTRY` command (Section 4.5.3)
- A *symbol assignment statement*, which is used to set the location counter (Section 4.7). The location counter specifies the default address in any subsequent section-mapping statements that do not explicitly specify an address.
- An *output section description*, which specifies one or more input sections in one or more library files, and maps those sections to the specified output section. The virtual memory address of the output section can additionally be specified, using attribute keywords.

#### 4.5.2.2 Output section description

A SECTIONS command may contain one or more *output section descriptions*.

An output section description has the following syntax:

```
section-name [virtual_addr] [(type)] :
     [AT(load_addr)]
     [ALIGN(section_align)]
     [SUBALIGN(subsection_align)]
     [constraint]
{
    output-section-command
    output-section-command
    ...
} [>region] [AT>lma_region] [:phdr ...] [=fillexp]
```

*section-name* specifies the name of the output section.

*virtual-addr* specifies the optional virtual address of the output section. The address value can be an expression (Section 4.6).

*type* specifies the optional section load property:

- NOLOAD – Mark section as not loadable
- DSECT – Parsed only, no effect on linking
- COPY – Parsed only, no effect on linking
- INFO – Parsed only, no effect on linking
- OVERLAY – Parsed only, no effect on linking

*load-addr* specifies the optional load address of the output section. The address value can be specified as an expression (Section 4.6).

*section-align* specifies the optional section alignment of the output section. The alignment value can be an expression (Section 4.6).

*subsection-align* specifies the optional subsection alignment of the output section. The alignment value can be an expression (Section 4.6).

*constraint* specifies the optional access type of the input sections:

- NOLOAD – All input sections are read-only
- DSECT – All input sections are read/write (default)

*output-section-command* specifies an output section command (Section 4.5.2.3). An output section description contains one or more output section commands.

*region* specifies the optional region of the output section. The region is expressed as a string. This option is parsed, but has no effect on linking.

*lma-region* specifies the optional load memory address (LMA) region of the output section. The value can be an expression. This option is parsed, but has no effect on linking.

*fillexp* specifies the optional fill value of the output section. The value can be an expression. This option is parsed, but has no effect on linking.

*phdr* specifies an optional program segment for the output section. This option can appear more than once in an output section description, to assign multiple program segments to an output section.

**NOTE** Setting `phdr` in an output section description will affect any subsequent output sections. This behavior differs from the GNU linker.

#### 4.5.2.3 Output section commands

An output section description contains one or more *output section commands*.

An output section command consists of one or more statements separated by semicolon characters. The statements can be any of the following:

- Output section data
- Output section keyword
- A *symbol assignment statement*, which is used to set the location counter (Section 4.7). The location counter specifies the default address in any subsequent section-mapping statements that do not explicitly specify an address.
- An *input section description*, which specifies one or more input sections in one or more library files (Section 4.5.2.6).

#### 4.5.2.4 Output section data

The `OUTPUT_SECTION_DATA` operator can be used to include specific bytes of data from an expression value (Section 4.6). It has the following syntax:

**`OUTPUT_SECTION_DATA`**_*`keyword`*`(`*`expression`*`)`

... where *`keyword`* can have the literal values `BYTE`, `SHORT`, `LONG`, `QUAD`, or `SQUAD`.

**NOTE** Output section data is currently not supported by the linker. The keyword is parsed but the generated data value is undefined, and the linker does not generate any warning message. Because of this, all output section data references should be replaced with fill expressions (Section 4.5.2.2).

#### 4.5.2.5 Output section keyword

For compatibility with the GNU linker, the keywords `CREATE_OBJECT_SYMBOLS`, `CONSTRUCTORS`, and `SORT_BY_NAME(CONSTRUCTORS)` are parsed, but have no effect on linking.

#### 4.5.2.6 Input section descriptions

An output section command may contain one or more input section descriptions. An input section description specifies the sections to be linked, and what files they are contained in.

An input section description has the following basic syntax for specifying a section in an object file:

```
file_name(section_name)
```

A single input section description can specify multiple files or sections:

```
file_name [:file_name ]... (section_name [section_name]...)
```

Multiple file names are separated by colon characters, while multiple section names are separated by space characters.

The wildcard characters `*` and `?` can be used in both file names and section names. For example, the following input file description specifies all the input sections named `.text*` that are contained in all the linker input files (`*`):

```
*(.text*)
```

The `EXCLUDE_FILE` operator can be used to reduce the number of items matched by a wildcard expression. Excluded items can be files, archives, or archive members. For example:

```
*(EXCLUDE_FILE(*crtend.o) .text .data)
```

This specifies all the linker input files (`*`) except for any files named `*crtend.o`.

**NOTE** If an exclusion is used in a list of section names, it applies only to the immediately following section name in the list (`.text` in the example above).

For several examples of using exclusions, see Section 4.8.

The `KEEP` operator can be used to prevent the linker from performing garbage collection on unused sections when the linker option `-gc-sections` is used. For example:

```
KEEP(*(.init))
```

This specifies that the input section named `.init` will be retained by the linker even if it is not referenced in the program being linked.

**NOTE** For compatibility with the GNU linker, the sort operators `SORT_NONE`, `SORT_BY_NAME`, `SORT_BY_ALIGNMENT`, and `SORT_BY_INIT_PRIORITY` are all parsed, but have no effect on linking.

### 4.5.3 ENTRY

```
ENTRY (symbol)
```

Script command which specifies the program execution entry point. The entry point is the first instruction that is executed after a program is loaded.

This command is equivalent to the linker command-line option `-e`.

### 4.5.4 OUTPUT_FORMAT

```
OUTPUT_FORMAT (string)
```

Script command which specifies the output file properties.

For compatibility with the GNU linker, this command is parsed but has no effect on linking.

### 4.5.5 OUTPUT_ARCH

```
OUTPUT_ARCH ("aarch64")
```

Script command which specifies the target processor architecture.

For compatibility with the GNU linker, this command is parsed but has no effect on linking.

### 4.5.6 SEARCH_DIR

```
SEARCH_DIR (path)
```

Script command which specifies which adds the specified path to the list of paths that the linker uses to search for libraries.

This command is equivalent to the linker command-line option `-L`.

### 4.5.7 INCLUDE

**INCLUDE** (*file*)

Script command which specifies the contents of the specified text file at the current location in the linker script.

The specified file is searched for in the current directory and in any directory that the linker uses to search for libraries.

**NOTE** Include files can be nested.

### 4.5.8 OUTPUT

**OUTPUT** (*file*)

Script command which specifies defines the location and file name where the linker will use to write output data. Only one output is allowed per linking.

### 4.5.9 GROUP

**GROUP** (*file, file, …*)

Script command which specifies which includes a list of achieve file names. The achieves defined in the list are searched repeatedly until all defined references are resolved.

### 4.5.10 ASSERT

**ASSERT**(*expression, string*)

Script command which specifies adds an assertion to the linker script.

If the specified expression evaluates to zero, the linker displays the specified string as an error message, and then exits with an error result code.

## 4.6 Expressions

Expressions in linker scripts are identical to C expressions. They are evaluated in the same size: 32-bit for the ARM architecture, and 64-bit for the AArch64 architecture.

Besides the `SECTION` command operators (Section 4.5.2), the linker defines a number of functions which can be used in linker script expressions.

Table 4-2 lists the script expression functions.

**Table 4-2 Linker script expression functions**

| Function | Description |
|---|---|
| . | Return location counter value (representing current virtual address). |
| ABSOLUTE (*expression*) | Return absolute value of expression. |
| ADDR (*string*) | Return virtual address of symbol or section.<br>Dot (.) is supported. |
| ALIGN (*expression*) | Return value when current location counter is aligned to next expression boundary.<br>Value of current location counter is not changed. |
| ALIGN (*expression1*, *expression2*) | Return value when value of expression1 is aligned to next expression2 boundary. |
| ALIGNOF (*string*) | Return align information of symbol or section. |
| ASSERT (*expression*, *string*) | Throw assertion if expression result is zero. |
| BLOCK (*expression*) | Synonym for ALIGN (*expression*). |
| DATA_SEGMENT_ALIGN (*maxpagesize*, *commonpagesize*) | Equivalent to:<br>(ALIGN(maxpagesize) + ( . & (maxpagesize - 1)))<br>or<br>(ALIGN(maxpagesize) + ( . & (maxpagesize - commonpagesize)))<br>Linker computes both of these values, returns the larger one. |
| DATA_SEGMENT_END (*expression*) | Not used, returns value of expression. |
| DATA_SEGMENT_RELRO_END (*expression*) | Not used, returns value of expression. |
| DEFINED (*symbol*) | Return 1 if symbol defined in linker's global symbol table.<br>Otherwise return 0. |
| LOADADDR (*string*) | Synonym for ADDR (*string*). |
| MAX (*expression1*, *expression2*) | Return maximum value of two expressions. |
| MIN (*expression1*, *expression2*) | Return minimum value of two expressions. |
| SEGMENT_START (*string*, *expression*) | If string matches known segment, return start address of that segment.<br>If nothing found, return value of expression. |
| SIZEOF (*string*) | Return size of symbol or section. |
| SIZEOF_HEADERS | Return section start file offset. |
| CONSTANT (MAXPAGESIZE) | Return defined default page size required by ABI. |
| CONSTANT (COMMONPAGESIZE) | Return defined common page size. |

## 4.7 Symbol assignment

Any symbol defined in a linker script becomes a global symbol. The following C assignment operators are supported to assign a value to a symbol:

- *symbol* = *expression*;
- *symbol* += *expression*;
- *symbol* -= *expression*;
- *symbol* *= *expression*;
- *symbol* /= *expression*;
- *symbol* &= *expression*;
- *symbol* |= *expression*;
- *symbol* <<= *expression*;
- *symbol* >>= *expression*;

The first statement above defines *symbol* and assigns it the value of *expression*. In the other statements, *symbol* must already be defined.

**NOTE** All the statements above must be terminated with a semicolon (;) character.

A common way to create an empty space in memory is to use the expression "`. += space_size`". For example:

```
BSS1 { . += 0x2000 }
```

This statement generates a section named *BSS1* with size 0x2000.

Linker scripts support several functions which perform symbol assignment. Table 4-3 lists the symbol assignment functions.

**Table 4-3 Linker script symbol assignments**

| Function | Description |
|---|---|
| HIDDEN (*symbol* = *expression*) | Hide the defined symbol so it is not exported.<br>This statement must be terminated with a semicolon (;). |
| FILL (*expression*) | Specify the fill value for the current section.<br>The fill length can be 1, 2, 4, or 8. The linker determines the length by selecting the minimum fit length. In the following example, the fill length is 8:<br><br>FILL( 0xdeadc0de )<br><br>A FILL statement covers memory locations from the point at which it occurs to the end of the current section.<br>Multiple FILL statements can be used in an output section definition to fill different parts of the section with different patterns. |
| ASSERT (*expression*, *string*) | When the specified expression is zero, the linker throws an assertion with the specified message string.<br>Standard ASSERT operation. |
| PROVIDE (*symbol* = *expression*) | Similar to symbol assignment, but does not perform checking for an unresolved reference.<br>This statement must be terminated with a semicolon (;).<br><br>NOTE - This behavior is different from the GNU linker, where the symbol is defined with the specified value only if any unresolved references exist for the symbol during linking. |
| PROVIDE_HIDDEN (*symbol* = *expression*) | Similar to PROVIDE, but hides the defined symbol so it will not be exported. |
| PRINT (*symbol* = *expression*) | Instruct linker to print symbol name and expression value to standard output during parsing.<br>NOTE - This operation may not work in certain situations. |

## 4.8 Linker script examples

This section presents several example linker scripts which show how to specify input files for linking. The examples all use the `EXCLUDE_FILE` operator which is defined for use in input section descriptions (Section 4.5.2.6).

### 4.8.1 Exclude file in archive

To exclude a file in an archive from being linked, specify the archive as part of the input expression, and specify the file to be excluded as the parameter of the following `EXCLUDE_FILE` operator.

**NOTE** Any sections whose names match the exclusion will still be included in the link, except if they are stored in the excluded archive.

The following example uses four relocatable files: `a1.o`, `a2.o`, `a3.o`, `a4.o`. Each file contains functions named `foo_`*N* and `bar_`*N*, where *N* indicates the digit in the file name.

The linker script excludes `foo_2` but not `bar_2` from `.text1` (because `EXCLUDE_FILE` applies only to the immediately following section name):

```
script.t :
SECTIONS {
  .text1 : {
    *lib23.a:(EXCLUDE_FILE(a2.o) .text.foo* .text.bar*)
  }
  .text2 : {
  *(*)
  }
}
```

```
clang -ffunction-sections -c a1.c a2.c a3.c a4.c
arm-ar cr lib23.a a2.o a3.o
arm-ar cr lib4.a a4.o
arm-link -T script.t -o mcld.out a1.o --whole-archive lib23.a lib4.a --no-whole-archive
```

Section headers (starting at offset `0x22a0`):

```
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text1            PROGBITS        00000000 001000 00002c 00  AX  0   0 16
  [ 2] .text2            PROGBITS        00000030 002030 00009c 00 WAX  0   0 16
```

Symbol table:

```
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     8: 00000040    12 FUNC    GLOBAL DEFAULT    2 bar_1
     9: 00000000    12 FUNC    GLOBAL DEFAULT    1 bar_2
    10: 00000020    12 FUNC    GLOBAL DEFAULT    1 bar_3
    11: 000000c0    12 FUNC    GLOBAL DEFAULT    2 bar_4
    12: 00000030    12 FUNC    GLOBAL DEFAULT    2 foo_1
    13: 000000a0    12 FUNC    GLOBAL DEFAULT    2 foo_2
    14: 00000010    12 FUNC    GLOBAL DEFAULT    1 foo_3
    15: 000000b0    12 FUNC    GLOBAL DEFAULT    2 foo_4
```

### 4.8.2 Exclude all files in archive

To exclude all files in an archive from being linked, specify the archive to be excluded as the parameter of the EXCLUDE_FILE operator.

**NOTE** Any sections whose names match the exclusion will still be included in the link, except if they are stored in the excluded archive.

The following example uses four relocatable files: a1.o, a2.o, a3.o, a4.o. Each file contains functions named foo_*N* and bar_*N*, where *N* indicates the digit in the file name.

The linker script excludes foo_2/3 but not bar_2/3 from .text1 (because EXCLUDE_FILE applies only to the immediately following section name):

```
script.t :
SECTIONS {
  .text1 : {
    *lib*:(EXCLUDE_FILE(*lib23.a) .text.foo* .text.bar*)
  }
  .text2 : {
  *(*)
  }
}
```

```
clang -ffunction-sections -c a1.c a2.c a3.c a4.c
arm-ar cr lib23.a a2.o a3.o
arm-ar cr lib4.a a4.o
arm-link -T script.t -o mcld.out a1.o --whole-archive lib23.a lib4.a --no-whole-archive
```

Section headers (starting at offset 0x22a0):

```
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text1            PROGBITS        00000000 001000 00003c 00  AX  0   0 16
  [ 2] .text2            PROGBITS        00000040 002040 00008c 00 WAX  0   0 16
```

Symbol table:

```
Num:    Value  Size Type    Bind   Vis      Ndx Name
  8: 00000050    12 FUNC    GLOBAL DEFAULT    2 bar_1
  9: 00000000    12 FUNC    GLOBAL DEFAULT    1 bar_2
 10: 00000010    12 FUNC    GLOBAL DEFAULT    1 bar_3
 11: 00000030    12 FUNC    GLOBAL DEFAULT    1 bar_4
 12: 00000040    12 FUNC    GLOBAL DEFAULT    2 foo_1
 13: 000000b0    12 FUNC    GLOBAL DEFAULT    2 foo_2
 14: 000000c0    12 FUNC    GLOBAL DEFAULT    2 foo_3
 15: 00000020    12 FUNC    GLOBAL DEFAULT    1 foo_4
```

### 4.8.3 Exclude multiple files

To exclude multiple files from being linked, specify the files as parameters of the `EXCLUDE_FILE` operator.

**NOTE** `EXCLUDE_FILE` accepts multiple file name parameters.

The following example uses four relocatable files: `a1.o`, `a2.o`, `a3.o`, `a4.o`. Each file contains functions named `foo_`*N* and `bar_`*N*, where *N* indicates the digit in the file name.

The linker script excludes `foo_2/3` but not `bar_2/3` from `.text1` (because `EXCLUDE_FILE` applies only to the immediately following section name):

```
script.t :
SECTIONS {
  .text1 : {
    *lib*:(EXCLUDE_FILE(a2.0 a3.0) .text.foo* .text.bar*)
  }
  .text2 : {
  *(*)
  }
}
```

```
clang -ffunction-sections -c a1.c a2.c a3.c a4.c
arm-ar cr lib23.a a2.o a3.o
arm-ar cr lib4.a a4.o
arm-link -T script.t -o mcld.out a1.o --whole-archive lib23.a lib4.a --no-whole-archive
```

Section headers (starting at offset `0x2260`):

```
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text1            PROGBITS        00000000 001000 00003c 00  AX  0   0 16
  [ 2] .text2            PROGBITS        00000040 002040 000040 00 WAX  0   0 16
```

Symbol table:

```
Num:    Value  Size Type    Bind   Vis      Ndx Name
  8: 00000050    12 FUNC    GLOBAL DEFAULT    2 bar_1
  9: 00000000    12 FUNC    GLOBAL DEFAULT    1 bar_2
 10: 00000010    12 FUNC    GLOBAL DEFAULT    1 bar_3
 11: 00000030    12 FUNC    GLOBAL DEFAULT    1 bar_4
 12: 00000040    12 FUNC    GLOBAL DEFAULT    2 foo_1
 13: 00000060    12 FUNC    GLOBAL DEFAULT    2 foo_2
 14: 00000070    12 FUNC    GLOBAL DEFAULT    2 foo_3
 15: 00000020    12 FUNC    GLOBAL DEFAULT    1 foo_4
```

### 4.8.4 Exclude archive and non-archive files

To exclude both archive and a non-archive files from being linked, specify the files as parameters of the `EXCLUDE_FILE` operator.

**NOTE** `EXCLUDE_FILE` searches both inside and outside archives for files to exclude.

The following example uses four relocatable files: `a1.o`, `a2.o`, `a3.o`, `a4.o`. Each file contains functions named `foo_N` and `bar_N`, where *N* indicates the digit in the file name.

The linker script excludes `foo_1/2` but not `bar_1/2` from `.text1` (because `EXCLUDE_FILE` applies only to the immediately following section name):

```
script.t :
SECTIONS {
  .text1 : {
    *:(EXCLUDE_FILE(a[12].o) .text.foo* .text.bar*)
  }
  .text2 : {
  *(*)
  }
}
```

```
clang -ffunction-sections -c a1.c a2.c a3.c a4.c
arm-ar cr lib23.a a2.o a3.o
arm-ar cr lib4.a a4.o
arm-link -T script.t -o mcld.out a1.o --whole-archive lib23.a lib4.a --no-whole-archive
```

Section headers (starting at offset `0x2260`):

```
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text1            PROGBITS        00000000 001000 00004c 00  AX  0   0 16
  [ 2] .text2            PROGBITS        00000050 002050 000030 00 WAX  0   0 16
```

Symbol table:

```
Num:    Value  Size Type    Bind   Vis      Ndx Name
  8: 00000060    12 FUNC    GLOBAL DEFAULT    2 bar_1
  9: 00000000    12 FUNC    GLOBAL DEFAULT    1 bar_2
 10: 00000020    12 FUNC    GLOBAL DEFAULT    1 bar_3
 11: 00000040    12 FUNC    GLOBAL DEFAULT    1 bar_4
 12: 00000050    12 FUNC    GLOBAL DEFAULT    2 foo_1
 13: 00000070    12 FUNC    GLOBAL DEFAULT    2 foo_2
 14: 00000010    12 FUNC    GLOBAL DEFAULT    1 foo_3
 15: 00000030    12 FUNC    GLOBAL DEFAULT    1 foo_4
```

### 4.8.5 Conflicting wildcards

If a section is both included and excluded in an input file specification, the exclusion has no effect, and any following section name will act like a normal wildcard.

**NOTE** This is a common error when using linker scripts.

The following example uses four relocatable files: `a1.o`, `a2.o`, `a3.o`, `a4.o`. Each file contains functions named `foo_`*N* and `bar_`*N*, where *N* indicates the digit in the file name.

The linker script does not exclude `foo_2` from `.text1` (because `EXCLUDE_FILE` has no effect in this case):

```
script.t :
SECTIONS {
  .text1 : {
    *lib23.a:(.text.foo* EXCLUDE_FILE(a2.o) .text.foo* .text.bar*)
  }
  .text2 : {
  *(*)
  }
}
```

```
clang -ffunction-sections -c a1.c a2.c a3.c a4.c
arm-ar cr lib23.a a2.o a3.o
arm-ar cr lib4.a a4.o
arm-link -T script.t -o mcld.out a1.o --whole-archive lib23.a lib4.a --no-whole-archive
```

Section headers (starting at offset `0x22a0`):

```
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text1            PROGBITS        00000000 001000 00003c 00  AX  0   0 16
  [ 2] .text2            PROGBITS        00000040 002040 00008c 00 WAX  0   0 16
```

Symbol table:

```
Num:    Value  Size Type    Bind   Vis      Ndx Name
  8: 00000050    12 FUNC    GLOBAL DEFAULT    2 bar_1
  9: 00000010    12 FUNC    GLOBAL DEFAULT    1 bar_2
 10: 00000030    12 FUNC    GLOBAL DEFAULT    1 bar_3
 11: 000000c0    12 FUNC    GLOBAL DEFAULT    2 bar_4
 12: 00000040    12 FUNC    GLOBAL DEFAULT    2 foo_1
 13: 00000000    12 FUNC    GLOBAL DEFAULT    1 foo_2
 14: 00000020    12 FUNC    GLOBAL DEFAULT    1 foo_3
 15: 000000b0    12 FUNC    GLOBAL DEFAULT    2 foo_4
```