



Arm[®] Compiler

Version 6.6

fromelf User Guide

Non-Confidential

Copyright © 2014–2017, 2019–2020, 2023 Arm Limited (or its affiliates).
All rights reserved.

Issue

DUI0805_I_en



Arm® Compiler fromelf User Guide

Copyright © 2014–2017, 2019–2020, 2023 Arm Limited (or its affiliates). All rights reserved.

Release information

Document history

Issue	Date	Confidentiality	Change
A	14 March 2014	Non-Confidential	Arm Compiler v6.00 Release
B	15 December 2014	Non-Confidential	Arm Compiler v6.01 Release
C	30 June 2015	Non-Confidential	Arm Compiler v6.02 Release
D	18 November 2015	Non-Confidential	Arm Compiler v6.3 Release
E	24 February 2016	Non-Confidential	Arm Compiler v6.4 Release
F	29 June 2016	Non-Confidential	Arm Compiler v6.5 Release
G	4 November 2016	Non-Confidential	Arm Compiler v6.6 Release
H	8 May 2017	Non-Confidential	Arm Compiler v6.6.1 Release
I	29 November 2017	Non-Confidential	Arm Compiler v6.6.2 Release
J	28 August 2019	Non-Confidential	Arm Compiler v6.6.3 Release
K	26 August 2020	Non-Confidential	Arm Compiler v6.6.4 Release
L	31 January 2023	Non-Confidential	Arm Compiler v6.6.5 Release

Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

THIS DOCUMENT IS PROVIDED “AS IS”. ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm’s customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its affiliates) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm’s trademark usage guidelines at <https://www.arm.com/company/policies/trademarks>.

Copyright © 2014–2017, 2019–2020, 2023 Arm Limited (or its affiliates). All rights reserved.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Feedback

Arm® welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on <https://support.developer.arm.com>

To provide feedback on the document, fill the following survey: <https://developer.arm.com/documentation-feedback-survey>.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

List of Figures..... 9

List of Tables..... 10

1. Introduction.....	11
1.1 Conventions.....	11
1.2 Other information.....	12
2. Overview of the fromelf Image Converter.....	13
2.1 About the fromelf image converter.....	13
2.2 fromelf execution modes.....	14
2.3 Getting help on the fromelf command.....	14
2.4 fromelf command-line syntax.....	14
2.5 Support level definitions.....	15
3. Using fromelf.....	20
3.1 General considerations when using fromelf.....	20
3.2 Examples of processing ELF files in an archive.....	20
3.3 Options to protect code in image files with fromelf.....	21
3.4 Options to protect code in object files with fromelf.....	22
3.5 Option to print specific details of ELF files.....	24
3.6 Using fromelf to find where a symbol is placed in an executable ELF image.....	25
4. fromelf Command-line Options.....	27
4.1 --base [[object_file::]load_region_ID=]num.....	27
4.2 --bin.....	28
4.3 --bincombined.....	29
4.4 --bincombined_base=address.....	30
4.5 --bincombined_padding=size,num.....	31
4.6 --cad.....	32
4.7 --cadcombined.....	33
4.8 --compare=option[,option,...].....	34
4.9 --continue_on_error.....	35
4.10 --cpu=list.....	35
4.11 --cpu=name.....	36
4.12 --datasymbols.....	38
4.13 --debugonly.....	38
4.14 --decode_build_attributes.....	39
4.15 --diag_error=tag[,tag,...].....	40
4.16 --diag_remark=tag[,tag,...].....	40
4.17 --diag_style=arm ide gnu.....	41

4.18 --diag_suppress=tag[,tag,...]	42
4.19 --diag_warning=tag[,tag,...]	42
4.20 --disassemble	43
4.21 --dump_build_attributes	43
4.22 --elf	44
4.23 --emit=option[,option,...]	45
4.24 --expandarrays	47
4.25 --extract_build_attributes	48
4.26 --fieldoffsets	49
4.27 --fpu=list	51
4.28 --fpu=name	51
4.29 --globalize=option[,option,...]	52
4.30 --help	53
4.31 --hide=option[,option,...]	53
4.32 --hide_and_localize=option[,option,...]	54
4.33 --i32	54
4.34 --i32combined	55
4.35 --ignore_section=option[,option,...]	56
4.36 --ignore_symbol=option[,option,...]	57
4.37 --in_place	58
4.38 --info=topic[,topic,...]	58
4.39 input_file	59
4.40 --interleave=option	61
4.41 --linkview, --no_linkview	62
4.42 --localize=option[,option,...]	63
4.43 --m32	63
4.44 --m32combined	64
4.45 --only=section_name	65
4.46 --output=destination	66
4.47 --privacy	67
4.48 --qualify	67
4.49 --relax_section=option[,option,...]	68
4.50 --relax_symbol=option[,option,...]	69
4.51 --rename=option[,option,...]	69
4.52 --select=select_options	70
4.53 --show=option[,option,...]	71

4.54 --show_and_globalize=option[,option,...]	72
4.55 --show_cmdline	73
4.56 --source_directory=path	73
4.57 --strip=option[,option,...]	73
4.58 --symbolversions, --no_symbolversions	75
4.59 --text	76
4.60 --version_number	78
4.61 --vhx	79
4.62 --via=file	80
4.63 --vsr	80
4.64 -w	81
4.65 --wide64bit	81
4.66 --widthxbanks	82
5. Via File Syntax	84
5.1 Overview of via files	84
5.2 Via file syntax rules	84

List of Figures

Figure 2-1: Integration boundaries in Arm Compiler for Embedded 6.....	17
--	----

List of Tables

Table 3-1: Effect of fromelf -privacy and -strip options on images files..... 22

Table 3-2: Effect of fromelf -privacy and -strip options on object files.....23

Table 4-1: Examples of using -base.....28

Table 4-2: Supported Arm architectures..... 36

1. Introduction

Arm® Compiler fromelf User Guide provides information on how to use the `fromelf` utility.

1.1 Conventions

The following subsections describe conventions used in Arm documents.





Glossary



The Arm Glossary is a list of terms used in Arm documentation, together with definitions for those terms. The Arm Glossary does not contain terms that are industry standard unless the Arm meaning differs from the generally accepted meaning.

See the Arm® Glossary for more information: developer.arm.com/glossary.

Typographic conventions

Arm documentation uses typographical conventions to convey specific meaning.

Convention	Use
<i>italic</i>	Citations.
bold	Interface elements, such as menu names. Terms in descriptive lists, where appropriate.
monospace	Text that you can enter at the keyboard, such as commands, file and program names, and source code.
monospace <u>underline</u>	A permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
<and>	Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example: <pre>MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2></pre>
SMALL CAPITALS	Terms that have specific technical meanings as defined in the <i>Arm® Glossary</i> . For example, IMPLEMENTATION DEFINED , IMPLEMENTATION SPECIFIC , UNKNOWN , and UNPREDICTABLE .
 Caution	Recommendations. Not following these recommendations might lead to system failure or damage.
 Warning	Requirements for the system. Not following these requirements might result in system failure or damage.
 Danger	Requirements for the system. Not following these requirements will result in system failure or damage.
 Note	An important piece of information that needs your attention.

Convention	Use
 Tip	A useful tip that might make it easier, better or faster to perform a task.
 Remember	A reminder of something important that relates to the information you are reading.

1.2 Other information

See the Arm website for other relevant information.

- [Arm® Developer](#).
- [Arm® Documentation](#).
- [Technical Support](#).
- [Arm® Glossary](#).

2. Overview of the fromelf Image Converter

Gives an overview of the `fromelf` image converter provided with Arm® Compiler.

2.1 About the fromelf image converter

The `fromelf` image conversion utility allows you to modify ELF image and object files, and to display information on those files.

`fromelf` allows you to:

- Process Arm ELF object and image files that the compiler, assembler, and linker generate.
- Process all ELF files in an archive that `armar` creates, and output the processed files into another archive if necessary.
- Convert ELF images into other formats for use by ROM tools or for direct loading into memory. The formats available are:
 - Plain binary.
 - Motorola 32-bit S-record. (AArch32 state only).
 - Intel Hex-32. (AArch32 state only).
 - Byte oriented (Verilog Memory Model) hexadecimal.
- Display information about the input file, for example, disassembly output or symbol listings, to either `stdout` or a text file. Disassembly is generated in `armasm` assembler syntax and not GNU assembler syntax.



If your image is produced without debug information, `fromelf` cannot:

- Translate the image into other file formats.
- Produce a meaningful disassembly listing.



The command-line option descriptions and related information in the individual Arm® Compiler tools documents describe all the features that Arm Compiler supports. Any features not documented are not supported and are used at your own risk. You are responsible for making sure that any generated code using [Support level definitions](#) is operating correctly.

Related information

[fromelf execution modes](#) on page 14

[Options to protect code in image files with fromelf](#) on page 21

[Options to protect code in object files with fromelf](#) on page 22

[fromelf command-line syntax](#) on page 14

[fromelf Command-line Options](#) on page 27

2.2 fromelf execution modes

You can run `fromelf` in various execution modes.

The execution modes are:

- ELF mode (`--elf`), to resave a file as ELF.
- Text mode (`--text`, and others), to output information about an object or image file.
- Format conversion mode (`--bin`, `--m32`, `--i32`, `--vbx`).

Related information

[--bin](#) on page 28

[--elf](#) on page 44

[--i32](#) on page 54

[--m32](#) on page 63

[--text](#) on page 75

[--vbx](#) on page 79

2.3 Getting help on the fromelf command

Use the `--help` option to display a summary of the main command-line options.

This is the default if you do not specify any options or files.

To display the help information, enter:

```
fromelf --help
```

Related information

[fromelf command-line syntax](#) on page 14

[--help](#) on page 53

2.4 fromelf command-line syntax

You can specify an ELF file or library of ELF files on the `fromelf` command-line.

Syntax

```
fromelf options input_file
```

options

fromelf command-line options.

input_file

The ELF file or library file to be processed. When some options are used, multiple input files can be specified.

Related information

[fromelf Command-line Options](#) on page 27

[input_file](#) on page 59

2.5 Support level definitions

This describes the levels of support for various Arm® Compiler 6 features.

Arm Compiler 6 is built on Clang and LLVM technology. Therefore, it has more functionality than the set of product features described in the documentation. The following definitions clarify the levels of support and guarantees on functionality that are expected from these features.

Arm welcomes feedback regarding the use of all Arm Compiler 6 features, and intends to support users to a level that is appropriate for that feature. You can contact support at <https://developer.arm.com/support>.

Identification in the documentation

All features that are documented in the Arm Compiler 6 documentation are product features, except where explicitly stated. The limitations of non-product features are explicitly stated.

Product features

Product features are suitable for use in a production environment. The functionality is well tested, and is expected to be stable across feature and update releases.

- Arm intends to give advance notice of significant functionality changes to product features.
- If you have a support and maintenance contract, Arm provides full support for use of all product features.
- Arm welcomes feedback on product features.
- Any issues with product features that Arm encounters or is made aware of are considered for fixing in future versions of Arm Compiler.

In addition to fully supported product features, some product features are only alpha or beta quality.

Beta product features

Beta product features are implementation complete, but have not been sufficiently tested to be regarded as suitable for use in production environments.

Beta product features are identified with [BETA].

- Arm endeavors to document known limitations on beta product features.
- Beta product features are expected to eventually become product features in a future release of Arm Compiler 6.
- Arm encourages the use of beta product features, and welcomes feedback on them.
- Any issues with beta product features that Arm encounters or is made aware of are considered for fixing in future versions of Arm Compiler.

Alpha product features

Alpha product features are not implementation complete, and are subject to change in future releases, therefore the stability level is lower than in beta product features.

Alpha product features are identified with [ALPHA].

- Arm endeavors to document known limitations of alpha product features.
- Arm encourages the use of alpha product features, and welcomes feedback on them.
- Any issues with alpha product features that Arm encounters or is made aware of are considered for fixing in future versions of Arm Compiler.

Community features

Arm Compiler 6 is built on LLVM technology and preserves the functionality of that technology where possible. This means that there are more features available in Arm Compiler that are not listed in the documentation. These extra features are known as community features. For information on these community features, see the [Clang Compiler User's Manual](#).

Where community features are referenced in the documentation, they are identified with [COMMUNITY].

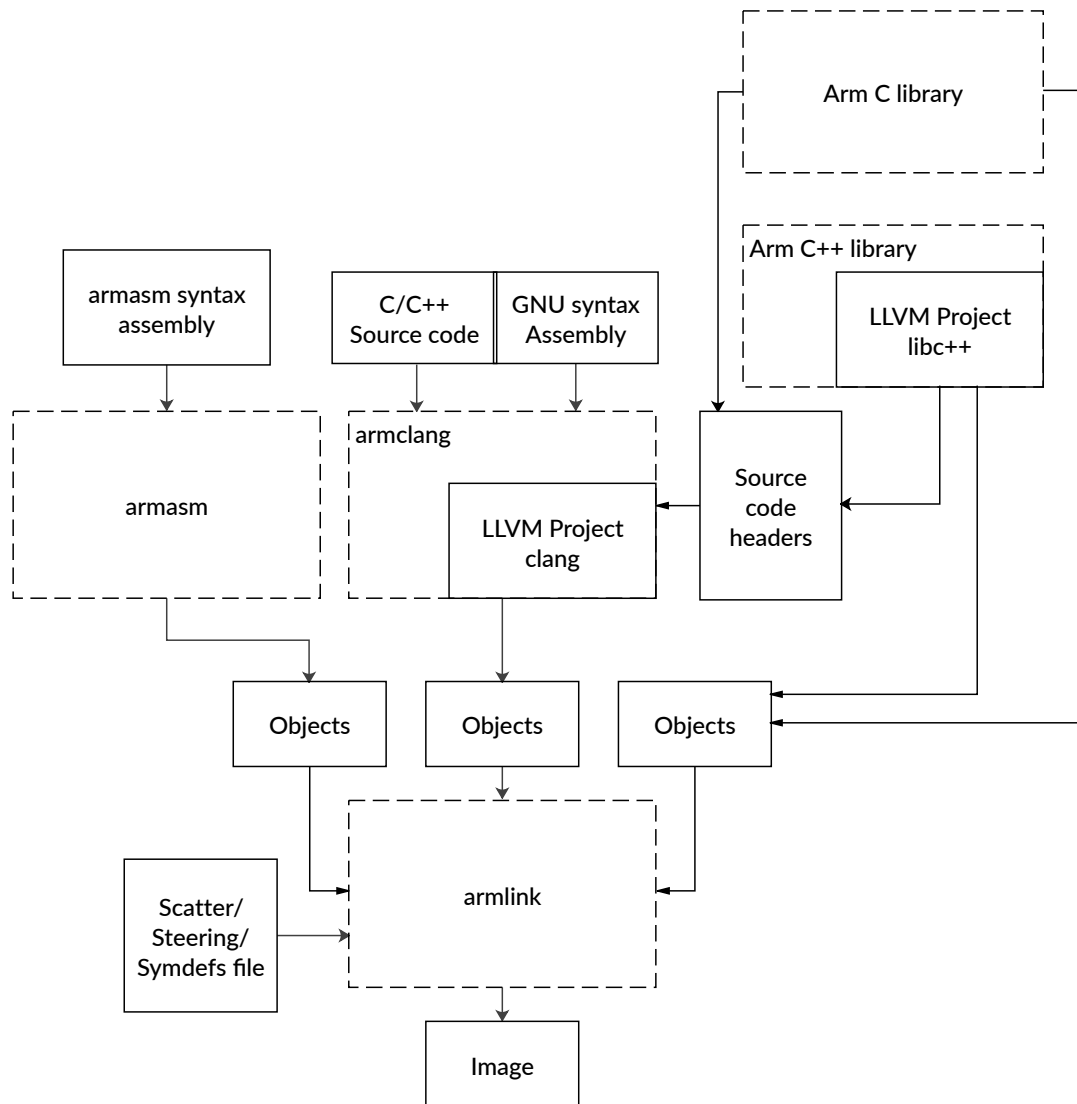
- Arm makes no claims about the quality level or the degree of functionality of these features, except when explicitly stated in this documentation.
- Functionality might change significantly between feature releases.
- Arm makes no guarantees that community features remain functional across update releases, although changes are expected to be unlikely.

Some community features might become product features in the future, but Arm provides no roadmap for such features. Arm is interested in understanding your use of these features, and welcomes feedback on them. Arm supports customers using these features on a best-effort basis, unless the features are unsupported. Arm accepts defect reports on these features, but does not guarantee that these issues are to be fixed in future releases.

Guidance on use of community features

There are several factors to consider when assessing the likelihood of a community feature being functional:

- The following figure shows the structure of the Arm Compiler 6 toolchain:

Figure 2-1: Integration boundaries in Arm Compiler for Embedded 6.

The dashed boxes are toolchain components, and any interaction between these components is an integration boundary. Community features that span an integration boundary might have significant limitations in functionality. The exception to such features is if the interaction is codified in one of the standards supported by Arm Compiler 6. See [Application Binary Interface \(ABI\)](#). Community features that do not span integration boundaries are more likely to work as expected.

- Features primarily used when targeting hosted environments such as Linux or BSD might have significant limitations, or might not be applicable, when targeting bare-metal environments.

- The Clang implementations of compiler features, particularly those features that have been present for a long time in other toolchains, are likely to be mature. The functionality of new features, such as support for new language features, is likely to be less mature and therefore more likely to have limited functionality.

Deprecated features

A deprecated feature is one that Arm plans to remove from a future release of Arm Compiler. Arm does not make any guarantee regarding the testing or maintenance of deprecated features. Therefore, Arm does not recommend using a feature after it is deprecated.

For information on replacing deprecated features with supported features, see the Arm Compiler documentation and Release Notes. Where appropriate, each Arm Compiler document includes notes for features that are deprecated, and also provides entries in the changes appendix of that document.

Unsupported features

With both the product and community feature categories, specific features and use-cases are known not to function correctly, or are not intended for use with Arm Compiler 6.

Limitations of product features are stated in the documentation. Arm cannot provide an exhaustive list of unsupported features or use-cases for community features. The known limitations on community features are listed in [Community features](#).

List of known unsupported features

The following is an incomplete list of unsupported features, and might change over time:

- The Clang option `-stdlib=libstdc++` is not supported.
- C++ static initialization of local variables is not thread-safe when linked against the standard C++ libraries. For thread-safety, you must provide your own implementation of thread-safe functions as described in [Standard C++ library implementation definition](#).



This restriction does not apply to the [ALPHA]-supported multithreaded C++ libraries.

-
- Use of C11 library features is unsupported.
 - Any community feature that is exclusively related to non-Arm architectures is not supported.
 - Except for Armv6-M, compilation for targets that implement architectures lower than Armv7 is not supported.
 - The `long double` data type is not supported for AArch64 state because of limitations in the current Arm C library.
 - C complex arithmetic is not supported, because of limitations in the current Arm C library.
 - Complex numbers are defined in C++ as a template, `std::complex`. Arm Compiler supports `std::complex` with the `float` and `double` types, but not the `long double` type because of limitations in the current Arm C library.



For C code that uses complex numbers, it is not sufficient to recompile with the C++ compiler to make that code work. How you can use complex numbers depends on whether you are building for Armv8-M architecture-based processors.

- You must take care when mixing translation units that are compiled with and without the [COMMUNITY] `-fsigned-char` option, and that share interfaces or data structures.



The Arm ABI defines `char` as an unsigned byte, and this is the interpretation used by the C libraries supplied with the Arm compilation tools.

Alternatives to C complex numbers not being supported

If you are building for Armv8-M architecture-based processors, consider using the free and Open Source CMSIS-DSP library that includes a data type and library functions for complex number support in C. For more information about CMSIS-DSP and complex number support see the following sections of the CMSIS documentation:

- [Complex Math Functions](#)
- [Complex Matrix Multiplication](#)
- [Complex FFT Functions](#)

If you are not building for Armv8-M architecture-based processors, consider modifying the affected part of your project to use the C++ standard template library type `std::complex` instead.

3. Using fromelf

Describes how to use the `fromelf` image converter provided with Arm® Compiler.

3.1 General considerations when using fromelf

There are some changes that you cannot make to an image with `fromelf`.

When using `fromelf` you cannot:

- Change the image structure or addresses, other than altering the base address of Motorola S-record or Intel Hex output with the `--base` option.
- Change a scatter-loaded ELF image into a non scatter-loaded image in another format. Any structural or addressing information must be provided to the linker at link time.

Related information

`--base [[object_file::]load_region_ID=num]` on page 27

`input_file` on page 59

3.2 Examples of processing ELF files in an archive

Examples of how you can process all ELF files in an archive, or a subset of those files. The processed files together with any unprocessed files are output to another archive.

Examples

Consider an archive, `test.a`, containing the following ELF files:

```
bmw.o  
bmwl.o  
call_c_code.o  
newtst.o  
shapes.o  
strmtst.o
```

Example of processing all files in the archive

This example removes all debug, comments, notes and symbols from all the files in the archive:

```
fromelf --elf --strip=all test.a -o strip_all/
```

This creates an output archive with the name `test.a` in the subdirectory `strip_all`.

Example of processing a subset of files in the archive

To remove all debug, comments, notes and symbols from only the `shapes.o` and the `strmtst.o` files in the archive, enter:

```
fromelf --elf --strip=all test.a(s*.o) -o subset/
```

This creates an output archive with the name `test.a` in the subdirectory `subset`. The archive contains the processed files together with the remaining files that are unprocessed.

To process the `bmw.o`, `bmw1.o`, and `newtst.o` files in the archive, enter:

```
fromelf --elf --strip=all test.a(??w*) -o subset/
```

Example of displaying a disassembled version of files in an archive

To display the disassembled version of `call_c_code.o` in the archive, enter:

```
fromelf --disassemble test.a(c*)
```



Note

On Unix systems your shell typically requires the parentheses to be escaped with backslashes. Alternatively, enclose the complete section specifier in double quotes, for example:

```
--entry="8+startup.o(startupseg) "
```

Related information

[--disassemble](#) on page 42

[--elf](#) on page 44

[input_file](#) on page 59

[--output=destination](#) on page 65

[--strip=option\[,option,...\]](#) on page 73

3.3 Options to protect code in image files with fromelf

If you are delivering images to third parties, then you might want to protect the code they contain.

To help you to protect this code, `fromelf` provides the `--strip` option and the `--privacy` option. These options remove or obscure the symbol names in the image. The option that you choose depends on how much information you want to remove. The effect of these options is different for image files.

Restrictions

You must use `--elf` with these options. Because you have to use `--elf`, you must also use `--output`.

Effect of the options for protecting code in image files

For image files:

Table 3-1: Effect of fromelf -privacy and -strip options on images files

Option	Effect
fromelf --elf --privacy	Removes the whole symbol table. Removes the <code>.comment</code> section name. This section is marked as [Anonymous Section] in the fromelf --text output. Gives section names a default value. For example, changes code section names to <code>'.text'</code> .
fromelf --elf --strip=symbols	Removes the whole symbol table. Section names remain the same.
fromelf --elf --strip=localsymbols	Removes local and mapping symbols. Retains section names and build attributes.

Example

To produce a new ELF executable image with the complete symbol table removed and with the various section names changed, enter:

```
fromelf --elf --privacy --output=outfile.axf infile.axf
```

Related information

[Options to protect code in object files with fromelf](#) on page 22

[fromelf command-line syntax](#) on page 14

[--elf](#) on page 44

[--output=destination](#) on page 65

[--privacy](#) on page 66

[--strip=option\[,option,...\]](#) on page 73

3.4 Options to protect code in object files with fromelf

If you are delivering objects to third parties, then you might want to protect the code they contain.

To help you to protect this code, fromelf provides the `--strip` option and the `--privacy` option. These options remove or obscure the symbol names in the object. The option you choose depends on how much information you want to remove. The effect of these options is different for object files.

Restrictions

You must use `--elf` with these options. Because you have to use `--elf`, you must also use `--output`.

Effect of the options for protecting code in object files

For object files:

Table 3-2: Effect of fromelf -privacy and -strip options on object files

Option	Local symbols	Section names	Mapping symbols	Build attributes
fromelf --elf --privacy	Removes those local symbols that can be removed without loss of functionality. Symbols that cannot be removed, such as the targets for relocations, are kept. For these symbols, the names are removed. These are marked as [Anonymous Symbol] in the fromelf --text output.	Gives section names a default value. For example, changes code section names to '.text'	Present	Present
fromelf --elf --strip=symbols	Removes those local symbols that can be removed without loss of functionality. Symbols that cannot be removed, such as the targets for relocations, are kept. For these symbols, the names are removed. These are marked as [Anonymous Symbol] in the fromelf --text output.	Section names remain the same	Present	Present
fromelf --elf --strip=localsymbols	Removes those local symbols that can be removed without loss of functionality. Symbols that cannot be removed, such as the targets for relocations, are kept. For these symbols, the names are removed. These are marked as [Anonymous Symbol] in the fromelf --text output.	Section names remain the same	Present	Present

Example

To produce a new ELF object with the complete symbol table removed and various section names changed, enter:

```
fromelf --elf --privacy --output=outfile.o infile.o
```

Related information

[Options to protect code in image files with fromelf](#) on page 21

[fromelf command-line syntax](#) on page 14

[--elf](#) on page 44

[--output=destination](#) on page 65

[--privacy](#) on page 66

[--strip=option\[,option,...\]](#) on page 73

3.5 Option to print specific details of ELF files

You can specify the elements of an ELF object that you want to appear in the textual output with the `--emit` option.

The output includes ELF header and section information. You can specify these elements as a comma separated list.



You can specify some of the `--emit` options using the `--text` option.

Examples

To print the contents of the data sections of an ELF file, `infile.axf`, enter:

```
fromelf --emit=data infile.axf
```

To print relocation information and the dynamic section contents for the ELF file `infile2.axf`, enter:

```
fromelf --emit=relocation_tables,dynamic_segment infile2.axf
```

Related information

[fromelf command-line syntax](#) on page 14

[--emit=option\[,option,...\]](#) on page 45

[--text](#) on page 75

3.6 Using fromelf to find where a symbol is placed in an executable ELF image

You can find where a symbol is placed in an executable ELF image.

About this task

To find where a symbol is placed in an ELF image file, use the `--text -s -v` options to view the symbol table and detailed information on each segment and section header.

The symbol table identifies the section where the symbol is placed.

Procedure

1. Create the file `s.c` containing the following source code:

```
long long arr[10] __attribute__((section("ARRAY")));
int main()
{
    return sizeof(arr);
}
```

2. Compile the source:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -c s.c -o s.o
```

3. Link the object `s.o` and keep the `ARRAY` symbol:

```
armlink --cpu=8-A.32 --keep=s.o(ARRAY) s.o --output=s.axf
```

4. Run the `fromelf` command to display the symbol table and detailed information on each segment and section header:

```
fromelf --text -s -v s.o
```

5. Locate the `arr` symbol in the `fromelf` output, for example:

```
...
=====
** Section #10

Name       : .symtab
Type       : SHT_SYMTAB (0x00000002)
Flags      : None (0x00000000)
Addr       : 0x00000000
File Offset : 312 (0x138)
Size       : 96 bytes (0x60)
Link       : Section 1 (.strtab)
Info       : Last local symbol no = 3
Alignment  : 4
Entry Size : 16

Symbol table .symtab (5 symbols, 3 local)

#  Symbol Name                Value      Bind  Sec  Type  Vis  Size
=====
...
4  arr                      0x00000000   Gb    6   Data  Hi   0x50
...
```

The `sec` column shows the section where the stack is placed. In this example, section 6.

6. Locate the section identified for the symbol in the `fromelf` output, for example:

```
...
```

```
=====
** Section #6

Name      : ARRAY
Type      : SHT_PROGBITS (0x00000001)
Flags     : SHF_ALLOC + SHF_WRITE (0x00000003)
Addr      : 0x00000000
File Offset : 88 (0x58)
Size      : 80 bytes (0x50)
Link      : SHN_UNDEF
Info      : 0
Alignment : 8
Entry Size : 0

=====
...
```

This example shows that the symbols are placed in an `ARRAY` section.

Related information

[--text](#) on page 75

4. fromelf Command-line Options

Describes the command-line options of the `fromelf` image converter provided with Arm® Compiler.

4.1 --base [[object_file::]load_region_ID=]num

Enables you to alter the base address specified for one or more load regions in Motorola S-record and Intel Hex file formats.



Not supported for AArch64 state inputs.

Syntax

```
--base [[object_file::]load_region_ID=]num
```

Where:

object_file

An optional ELF input file.

load_region_ID

An optional load region. This can either be a symbolic name of an execution region belonging to a load region or a zero-based load region number, for example #0 if referring to the first region.

num

Either a decimal or hexadecimal value.

You can:

- Use wildcard characters `?` and `*` for symbolic names in *object_file* and *load_region_ID* arguments.
- Specify multiple values in one *option* followed by a comma-separated list of arguments.

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

Restrictions

You must use one of the output formats `--i32`, `--i32combined`, `--m32`, or `--m32combined` with this option. Therefore, you cannot use this option with object files.

Examples

The following table shows examples:

Table 4-1: Examples of using -base

<code>--base 0</code>	decimal value
<code>--base 0</code>	decimal value
<code>--base 0x8000</code>	hexadecimal value
<code>--base #0=0</code>	base address for the first load region
<code>--base foo.o::*=0</code>	base address for all load regions in <code>foo.o</code>
<code>--base #0=0,#1=0x8000</code>	base address for the first and second load regions

Related information

[General considerations when using fromelf](#) on page 20

[--i32](#) on page 54

[--i32combined](#) on page 55

[--m32](#) on page 63

[--m32combined](#) on page 64

4.2 --bin

Produces plain binary output, one file for each load region. You can split the output from this option into multiple files with the `--widthxbanks` option.

Restrictions

The following restrictions apply:

- You cannot use this option with object files.
- You must use `--output` with this option.

Considerations when using --bin

If you convert an ELF image containing multiple load regions to a binary format, `fromelf` creates an output directory named *destination* and generates one binary output file for each load region in the input image. `fromelf` places the output files in the *destination* directory.



For multiple load regions, the name of the first non-empty execution region in the corresponding load region is used for the filename.

A file is only created when the load region describes code or data that is present in the ELF file. For example a load region containing only execution regions with ZI data in them does not result in an output file.

Example

To convert an ELF file to a plain binary file, for example `outfile.bin`, enter:

```
fromelf --bin --output=outfile.bin infile.axf
```

Related information

[--output=destination](#) on page 65

[--widthxbanks](#) on page 82

4.3 --bincombined

Produces plain binary output. It generates one output file for an image containing multiple load regions.

Usage

By default, the start address of the first load region in memory is used as the base address. `fromelf` inserts padding between load regions as required to ensure that they are at the correct relative offset from each other. Separating the load regions in this way means that the output file can be loaded into memory and correctly aligned starting at the base address.

Use this option with `--bincombined_base` and `--bincombined_padding` to change the default values for the base address and padding.

Restrictions

The following restrictions apply:

- You cannot use this option with object files.
- You must use `--output` with this option.

Considerations when using --bincombined

Use this option with `--bincombined_base` to change the default value for the base address.

The default padding value is `0xFF`. Use this option with `--bincombined_padding` to change the default padding value.

If you use a scatter file that defines two load regions with a large address space between them, the resulting binary can be very large because it contains mostly padding. For example, if you have a load region of size `0x100` bytes at address `0x00000000` and another load region at address `0x30000000`, the amount of padding is `0x2FFFFFF00` bytes.

Arm recommends that you use a different method of placing widely spaced load regions, such as `--bin`, and make your own arrangements to load the multiple output files at the correct addresses.

Examples

To produce a binary file that can be loaded at start address `0x1000`, enter:

```
fromelf --bincombined --bincombined_base=0x1000 --output=out.bin in.axf
```

To produce plain binary output and fill the space between load regions with copies of the 32-bit word 0x12345678, enter:

```
fromelf --bincombined --bincombined_padding=4,0x12345678 --output=out.bin in.axf
```

Related information

[--bincombined_base=address](#) on page 30

[--bincombined_padding=size,num](#) on page 30

[--output=destination](#) on page 65

[--widthxbanks](#) on page 82

[Input sections, output sections, regions, and Program Segments](#)

4.4 --bincombined_base=address

Enables you to lower the base address used by the `--bincombined` output mode. The output file generated is suitable to be loaded into memory starting at the specified address.

Default

By default the start address of the first load region in memory is used as the base address.

Syntax

```
--bincombined_base=address
```

Where is the start address where the image is to be loaded:

- If the specified address is lower than the start of the first load region, `fromelf` adds padding at the start of the output file.
- If the specified address is higher than the start of the first load region, `fromelf` gives an error.

Restrictions

You must use `--bincombined` with this option. If you omit `--bincombined`, a warning message is displayed.

Example

```
--bincombined --bincombined_base=0x1000
```

Related information

[--bincombined](#) on page 29

[--bincombined_padding=size,num](#) on page 30

[Input sections, output sections, regions, and Program Segments](#)

4.5 --bincombined_padding=size,num

Enables you to specify a different padding value from the default used by the `--bincombined` output mode.

Default

The default is `--bincombined_padding=1,0xFF`.

Syntax

`--bincombined_padding=size,num`

Where:

size

Is 1, 2, or 4 bytes to define whether it is a byte, halfword, or word.

num

The value to be used for padding. If you specify a value that is too large to fit in the specified size, a warning message is displayed.



fromelf expects that 2-byte and 4-byte padding values are specified in the appropriate endianness for the input file. For example, if you are translating a big endian ELF file into binary, the specified padding value is treated as a big endian word or halfword.

Restrictions

You must use `--bincombined` with this option. If you omit `--bincombined`, a warning message is displayed.

Examples

The following examples show how to use `--bincombined_padding`:

`--bincombined --bincombined_padding=4,0x12345678`

This example produces plain binary output and fills the space between load regions with copies of the 32-bit word `0x12345678`.

`--bincombined --bincombined_padding=2,0x1234`

This example produces plain binary output and fills the space between load regions with copies of the 16-bit halfword `0x1234`.

`--bincombined --bincombined_padding=2,0x01`

This example when specified for big endian memory, fills the space between load regions with `0x0100`.

Related information

[--bincombined](#) on page 29

[--bincombined_base=address](#) on page 30

4.6 --cad

Produces a C array definition or C++ array definition containing binary output.

Usage

You can use each array definition in the source code of another application. For example, you might want to embed an image in the address space of another application, such as an embedded operating system.

If your image has a single load region, the output is directed to `stdout` by default. To save the output to a file, use the `--output` option together with a filename.

If your image has multiple load regions, then you must also use the `--output` option together with a directory name. Unless you specify a full path name, the path is relative to the current directory. A file is created for each load region in the specified directory. The name of each file is the name of the corresponding execution region.

Use this option with `--output` to generate one output file for each load region in the image.

Restrictions

You cannot use this option with object files.

Considerations when using --cad

A file is only created when the load region describes code or data that is present in the ELF file. For example a load region containing only execution regions with ZI data in them does not result in an output file.

Example

The following examples show how to use `--cad`:

- To produce an array definition for an image that has a single load region, enter:

```
fromelf --cad myimage.axf
unsigned char LR0[] = {
    0x00,0x00,0x00,0xEB,0x28,0x00,0x00,0xEB,0x2C,0x00,0x8F,0xE2,0x00,0x0C,0x90,0xE8,
    0x00,0xA0,0x8A,0xE0,0x00,0xB0,0x8B,0xE0,0x01,0x70,0x4A,0xE2,0x0B,0x00,0x5A,0xE1,
    0x00,0x00,0x00,0x1A,0x20,0x00,0x00,0xEB,0x0F,0x00,0xBA,0xE8,0x18,0xE0,0x4F,0xE2,
    0x01,0x00,0x13,0xE3,0x03,0xF0,0x47,0x10,0x03,0xF0,0xA0,0xE1,0xAC,0x18,0x00,0x00,
    0xBC,0x18,0x00,0x00,0x00,0x30,0xB0,0xE3,0x00,0x40,0xB0,0xE3,0x00,0x50,0xB0,0xE3,
    0x00,0x60,0xB0,0xE3,0x10,0x20,0x52,0xE2,0x78,0x00,0xA1,0x28,0xFC,0xFF,0xFF,0x8A,
    0x82,0x2E,0xB0,0xE1,0x30,0x00,0xA1,0x28,0x00,0x30,0x81,0x45,0x0E,0xF0,0xA0,0xE1,
    0x70,0x00,0x51,0xE3,0x66,0x00,0x00,0x0A,0x64,0x00,0x51,0xE3,0x38,0x00,0x00,0x0A,
    0x00,0x00,0xB0,0xE3,0x0E,0xF0,0xA0,0xE1,0x1F,0x40,0x2D,0xE9,0x00,0x00,0xA0,0xE1,
```



```
.
.
.

0x3A,0x74,0x74,0x00,0x43,0x6F,0x6E,0x73,0x74,0x72,0x75,0x63,0x74,0x65,0x64,0x20,
0x41,0x20,0x23,0x25,0x64,0x20,0x61,0x74,0x20,0x25,0x70,0x0A,0x00,0x00,0x00,0x00,
0x44,0x65,0x73,0x74,0x72,0x6F,0x79,0x65,0x64,0x20,0x41,0x20,0x23,0x25,0x64,0x20,
0x61,0x74,0x20,0x25,0x70,0x0A,0x00,0x00,0x00,0x0C,0x99,0x00,0x00,0x0C,0x99,0x00,0x00,
0x50,0x01,0x00,0x00,0x44,0x80,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
};
```

- For an image that has multiple load regions, the following commands create a file for each load region in the directory `root\myprojects\multiload\load_regions`:

```
cd root\myprojects\multiload
fromelf --cad image_multiload.axf --output load_regions
```

If `image_multiload.axf` contains the execution regions `EXEC_ROM` and `RAM`, then the files `EXEC_ROM` and `RAM` are created in the `load_regions` subdirectory.

Related information

[--cadcombined](#) on page 33

[--output=destination](#) on page 65

[Input sections, output sections, regions, and Program Segments](#)

4.7 --cadcombined

Produces a C array definition or C++ array definition containing binary output.

Usage

You can use each array definition in the source code of another application. For example, you might want to embed an image in the address space of another application, such as an embedded operating system.

The output is directed to `stdout` by default. To save the output to a file, use the `--output` option together with a filename.

Restrictions

You cannot use this option with object files.

Example

The following commands create the file `load_regions.c` in the directory `root\myprojects\multiload`:

```
:guilabel:cd root  \\myprojects\\multiload
```

```
fromelf --cadcombined image_multiload.axf --output load_regions.c
```

Related information

[--cad](#) on page 32

[--output=destination](#) on page 65

4.8 --compare=option[,option,...]

Compares two input files and prints a textual list of the differences.

Usage

The input files must be the same type, either two ELF files or two library files. Library files are compared member by member and the differences are concatenated in the output.

All differences between the two input files are reported as errors unless specifically downgraded to warnings by using the `--relax_section` option.

Syntax

`--compare=option[,option,...]`

Where *option* is one of:

section_sizes

Compares the size of all sections for each ELF file or ELF member of a library file.

section_sizes::object_name

Compares the sizes of all sections in ELF objects with a name matching *object_name*.

section_sizes::section_name

Compares the sizes of all sections with a name matching *section_name*.

sections

Compares the size and contents of all sections for each ELF file or ELF member of a library file.

sections::object_name

Compares the size and contents of all sections in ELF objects with a name matching *object_name*.

sections::section_name

Compares the size and contents of all sections with a name matching *section_name*.

function_sizes

Compares the size of all functions for each ELF file or ELF member of a library file.

function_sizes::object_name

Compares the size of all functions in ELF objects with a name matching *object_name*.

function_size::function_name

Compares the size of all functions with a name matching *function_name*.

global_function_sizes

Compares the size of all global functions for each ELF file or ELF member of a library file.

global_function_sizes::function_name

Compares the size of all global functions in ELF objects with a name matching *function_name*.

You can:

- Use wildcard characters ? and * for symbolic names in *section_name*, *function_name*, and *object_name* arguments.
- Specify multiple values in one *option* followed by a comma-separated list of arguments.

Related information

[--ignore_section=option\[,option,...\]](#) on page 56

[--ignore_symbol=option\[,option,...\]](#) on page 57

[--relax_section=option\[,option,...\]](#) on page 68

[--relax_symbol=option\[,option,...\]](#) on page 68

4.9 --continue_on_error

Reports any errors and then continues.

Usage

Use `--diag_warning=error` instead of this option.

Related information

[--diag_warning=tag\[,tag,...\]](#) on page 42

4.10 --cpu=list

Lists the architecture and processor names that are supported by the `--cpu=name` option.

Syntax

`--cpu=list`

Related information

[--cpu=name](#) on page 35

4.11 --cpu=name

Affects the way machine code is disassembled by options such as `-c` or `--disassemble`, so that it is disassembled in the same way that the specified processor interprets it.

Default

If you do not specify a `--cpu` option, then `fromelf` disassembles machine instructions in an architecture-independent way. This means that `fromelf` disassembles anything that it recognizes as an instruction by some architecture.

Syntax

`--cpu=name`

Where *name* is the name of a processor or architecture.

Processor and architecture names are not case-sensitive.

Wildcard characters are not accepted.

The following table shows the supported architectures. For a complete list of the supported architecture and processor names, specify the `--cpu=list` option.

Table 4-2: Supported Arm architectures

Architecture name	Description
6-M	Arm®v6 architecture microcontroller profile.
6S-M	Armv6 architecture microcontroller profile with OS extensions.
7-A	Armv7 architecture application profile.
7-A.security	Armv7-A architecture profile with Security Extensions and includes the SMC instruction (formerly SMI).
7-R	Armv7 architecture real-time profile.
7-M	Armv7 architecture microcontroller profile.
7E-M	Armv7-M architecture profile with DSP extension.
8-A.32	Armv8-A architecture profile, AArch32 state.
8-A.32.crypto	Armv8-A architecture profile, AArch32 state with cryptographic instructions.
8-A.64	Armv8-A architecture profile, AArch64 state.
8-A.64.crypto	Armv8-A architecture profile, AArch64 state with cryptographic instructions.
8.1-A.32	Armv8.1, for Armv8-A architecture profile, AArch32 state.
8.1-A.32.crypto	Armv8.1, for Armv8-A architecture profile, AArch32 state with cryptographic instructions.
8.1-A.64	Armv8.1, for Armv8-A architecture profile, AArch64 state.
8.1-A.64.crypto	Armv8.1, for Armv8-A architecture profile, AArch64 state with cryptographic instructions.
8.2-A.32	Armv8.2, for Armv8-A architecture profile, AArch32 state.

Architecture name	Description
8.2-A.32.crypto	Armv8.2, for Armv8-A architecture profile, AArch32 state with cryptographic instructions.
8.2-A.64	Armv8.2, for Armv8-A architecture profile, AArch64 state.
8.2-A.64.crypto	Armv8.2, for Armv8-A architecture profile, AArch64 state with cryptographic instructions.
8.3-A.32	Armv8.3, for Armv8-A architecture profile, AArch32 state.
8.3-A.32.crypto	Armv8.3, for Armv8-A architecture profile, AArch32 state with cryptographic instructions.
8.3-A.64	Armv8.3, for Armv8-A architecture profile, AArch64 state.
8.3-A.64.crypto	Armv8.3, for Armv8-A architecture profile, AArch64 state with cryptographic instructions.
8-R	Armv8-R architecture profile.
8-M.Base	Armv8-M baseline architecture profile. Derived from the Armv6-M architecture.
8-M.Main	Armv8-M mainline architecture profile. Derived from the Armv7-M architecture.
8-M.Main.dsp	Armv8-M mainline architecture profile with DSP extension.



The full list of supported architectures and processors depends on your license.

Usage

The following general points apply to processor and architecture options:

Processors

Selecting the processor selects the appropriate architecture, Floating-Point Unit (FPU), and memory organization.

Architectures

If you specify an architecture name for the `--cpu` option, machine code is disassembled by options such as `-c` or `--disassemble` for that architecture. If you specify `--disassemble`, then the disassembly can be assembled for any processor supporting that architecture.

For example, `--cpu=7-A --disassemble` produces disassembly that can be assembled for the Cortex®-A7 processor.

FPU

- Some specifications of `--cpu` imply an `--fpu` selection.



Any explicit FPU, set with `--fpu` on the command line, overrides an *implicit* FPU.

- If no `--fpu` option is specified and no `--cpu` option is specified, `--fpu=softvfp` is used.

Example

To specify the Cortex-M4 processor, use:

```
--cpu=Cortex-M4
```

Related information

[--cpu=list](#) on page 35

[--disassemble](#) on page 42

[--info=topic\[,topic,...\]](#) on page 58

[--text](#) on page 75

4.12 --datasymbols

Modifies the output information of data sections so that symbol definitions are interleaved.

Usage

You can use this option only with `--text -d`.

Related information

[--text](#) on page 75

4.13 --debugonly

Removes the content of any code or data sections.

Usage

This option ensures that the output file contains only the information required for debugging, for example, debug sections, symbol table, and string table. Section headers are retained because they are required to act as targets for symbols.

Restrictions

You must use `--elf` with this option.

Example

To create an ELF file, `debugout.axf`, from the ELF file `infile.axf`, containing only debug information, enter:

```
fromelf --elf --debugonly --output=debugout.axf infile.axf
```

Related information

[--elf](#) on page 44

4.14 --decode_build_attributes

Prints the contents of the build attributes section in human-readable form for standard build attributes or raw hexadecimal form for nonstandard build attributes.



The standard build attributes are documented in the *Application Binary Interface for the Arm Architecture*.

Restrictions

This option has no effect for AArch64 state inputs.

Example

The following example shows the output for `--decode_build_attributes`:

```
armclang --target=arm-arm-eabi-none -march=armv8-a -c hello.c -o hello.o
fromelf -v --decode_build_attributes hello.o

...
** Section #6

Name      : .ARM.attributes
Type      : SHT_ARM_ATTRIBUTES (0x70000003)
Flags     : None (0x00000000)
Addr      : 0x00000000
File Offset : 112 (0x70)
Size      : 74 bytes (0x4a)
Link      : SHN_UNDEF
Info      : 0
Alignment : 1
Entry Size : 0

'aeabi' file build attributes:
0x000000:  43 32 2e 30 39 00 05 63 6f 72 74 65 78 2d 61 35      C2.09..cortex-a5
0x000010:  33 00 06 0e 07 41 08 01 09 02 0a 07 0c 03 0e 00      3....A.....
0x000020:  11 01 12 04 14 01 15 01 17 03 18 01 19 01 1a 02      .....
0x000030:  22 00 24 01 26 01 2a 01 44 03                        ".$.&.*.D.
Tag_conformance = "2.09"
Tag_CPU_name = "cortex-a53"
Tag_CPU_arch = ARM v8 (=14)
Tag_CPU_arch_profile = The application profile 'A' (e.g. for Cortex A8)
(=65)
Tag_ARM_ISA_use = ARM instructions were permitted to be used (=1)
Tag_THUMB_ISA_use = Thumb2 instructions were permitted (implies Thumb in
structions permitted) (=2)
Tag_VFP_arch = Use of the ARM v8-A FP ISA was permitted (=7)
Tag_NEON_arch = Use of the ARM v8-A Advanced SIMD Architecture (Neon) wa
s permitted (=3)
Tag_ABI_PCS_R9_use = R9 used as V6 (just another callee-saved register)
(=0)
Tag_ABI_PCS_GOT_use = Data are imported directly (=1)
Tag_ABI_PCS_wchar_t = Size of wchar_t is 4 (=4)
Tag_ABI_FP_denormal = This code was permitted to require IEEE 754 denorm
al numbers (=1)
Tag_ABI_FP_exceptions = This code was permitted to check the IEEE 754 in
exact exception (=1)
Tag_ABI_FP_number_model = This code may use all the IEEE 754-defined FP
```

```

encodings (=3)
    Tag_ABI_align8_needed = Code was permitted to depend on the 8-byte align-
ment of 8-byte data items (=1)
    Tag_ABI_align8_preserved = Code was required to preserve 8-byte alignmen-
t of 8-byte data objects (=1)
    Tag_ABI_enum_size = Enum containers are 32-bit (=2)
    Tag_CPU_unaligned_access = The producer was not permitted to make unalign-
ed data accesses (=0)
    Tag_VFP_HP_extension = The producer was permitted to use the VFPv3/Advan-
ced SIMD optional half-precision extension (=1)
    Tag_ABI_FP_16bit_format = The producer was permitted to use IEEE 754 for-
mat 16-bit floating point numbers (=1)
    Tag_MPextension_use = Use of the ARM v7 MP extension was permitted (=1)
    Tag_Virtualization_use = Use of TrustZone and virtualization extensions
was permitted (=3)
...

```

Related information

[--dump_build_attributes](#) on page 43

[--emit=option\[,option,...\]](#) on page 45

[--extract_build_attributes](#) on page 48

[Application Binary Interface for the ARM Architecture](#)

4.15 --diag_error=tag[,tag,...]

Sets diagnostic messages that have a specific tag to Error severity.

Syntax

```
--diag_error=tag[, tag, ...]
```

Where *tag* can be:

- A diagnostic message number to set to error severity. This is the four-digit number, *nnnn*, with the tool letter prefix, but without the letter suffix indicating the severity.
- `warning`, to treat all warnings as errors.

Related information

[--diag_remark=tag\[,tag,...\]](#) on page 40

[--diag_style=arm|ide|gnu](#) on page 41

[--diag_suppress=tag\[,tag,...\]](#) on page 41

[--diag_warning=tag\[,tag,...\]](#) on page 42

4.16 --diag_remark=tag[,tag,...]

Sets diagnostic messages that have a specific tag to Remark severity.

Syntax

```
--diag_remark=tag[, tag, ...]
```


Where *tag* is a comma-separated list of diagnostic message numbers. This is the four-digit number, *nnnn*, with the tool letter prefix, but without the letter suffix indicating the severity.

Related information

[--diag_error=tag\[,tag,...\]](#) on page 40

[--diag_style=arm|ide|gnu](#) on page 41

[--diag_suppress=tag\[,tag,...\]](#) on page 41

[--diag_warning=tag\[,tag,...\]](#) on page 42

4.17 --diag_style=arm|ide|gnu

Specifies the display style for diagnostic messages.

Default

The default is `--diag_style=arm`.

Syntax

`--diag_style=string`

Where *string* is one of:

arm

Display messages using the legacy Arm® compiler style.

ide

Include the line number and character count for any line that is in error. These values are displayed in parentheses.

gnu

Display messages in the format used by `gcc`.

Usage

`--diag_style=gnu` matches the format reported by the GNU Compiler, `gcc`.

`--diag_style=ide` matches the format reported by Microsoft Visual Studio.

Related information

[--diag_error=tag\[,tag,...\]](#) on page 40

[--diag_remark=tag\[,tag,...\]](#) on page 40

[--diag_suppress=tag\[,tag,...\]](#) on page 41

[--diag_warning=tag\[,tag,...\]](#) on page 42

4.18 --diag_suppress=tag[,tag,...]

Suppresses diagnostic messages that have a specific tag.

Syntax

```
--diag_suppress=tag[, tag, ...]
```

Where *tag* can be:

- A diagnostic message number to be suppressed. This is the four-digit number, *nnnn*, with the tool letter prefix, but without the letter suffix indicating the severity.
- `error`, to suppress all errors that can be downgraded.
- `warning`, to suppress all warnings.

Related information

[--diag_error=tag\[,tag,...\]](#) on page 40

[--diag_remark=tag\[,tag,...\]](#) on page 40

[--diag_style=arm|ide|gnu](#) on page 41

[--diag_warning=tag\[,tag,...\]](#) on page 42

4.19 --diag_warning=tag[,tag,...]

Sets diagnostic messages that have a specific tag to Warning severity.

Syntax

```
--diag_warning=tag[, tag, ...]
```

Where *tag* can be:

- A diagnostic message number to set to warning severity. This is the four-digit number, *nnnn*, with the tool letter prefix, but without the letter suffix indicating the severity.
- `error`, to set all errors that can be downgraded to warnings.

Related information

[--diag_error=tag\[,tag,...\]](#) on page 40

[--diag_remark=tag\[,tag,...\]](#) on page 40

[--diag_style=arm|ide|gnu](#) on page 41

[--diag_suppress=tag\[,tag,...\]](#) on page 41

4.20 --disassemble

Displays a disassembled version of the image to stdout. Disassembly is generated in `armasm` assembler syntax and not GNU assembler syntax.

Usage

If you use this option with `--output destination`, you can reassemble the output file with `armasm`.

You can use this option to disassemble either an ELF image or an ELF object file.



The output is not the same as that from `--emit=code` and `--text -c`.

Example

To disassemble the ELF file `infile.axf` for the Cortex®-A7 processor and create a source file `outfile.asm`, enter:

```
fromelf --cpu=Cortex-A7 --disassemble --output=outfile.asm infile.axf
```

Related information

[--cpu=name](#) on page 35

[--emit=option\[,option,...\]](#) on page 45

[--interleave=option](#) on page 61

[--output=destination](#) on page 65

[--text](#) on page 75

4.21 --dump_build_attributes

Prints the contents of the build attributes section in raw hexadecimal form.

Restrictions

This option has no effect for AArch64 state inputs.

Example

The following example shows the output for `--dump_build_attributes`:

```
...
** Section #10 '.ARM.attributes' (SHT_ARM_ATTRIBUTES)
   Size   : 89 bytes

0x000000:  41 47 00 00 00 61 65 61 62 69 00 01 3d 00 00 00    AG...aeabi...=...
0x000010:  43 32 2e 30 36 00 05 38 2d 41 2e 33 32 00 06 0a    C2.06..8-A.32...
0x000020:  07 41 08 01 09 02 0a 05 0c 02 11 01 12 02 14 02    .A.....
```

```

0x000030: 17 01 18 01 19 01 1a 01 1c 01 1e 03 22 01 24 01 .....".$.
0x000040: 42 01 44 03 46 01 2c 02 11 00 00 00 41 52 4d 00 B.D.F.,.....ARM.
0x000050: 01 09 00 00 00 12 01 16 01 .....

```

Related information

- [--decode_build_attributes](#) on page 39
- [--emit=option\[,option,...\]](#) on page 45
- [--extract_build_attributes](#) on page 48
- [--text](#) on page 75

4.22 --elf

Selects ELF output mode.

Usage

Use this option whenever you have to transform an ELF file into a slightly different ELF file. You also have to provide options to indicate how you want the file to be modified. The options are:

- `--debugonly.`
- `--globalize.`
- `--hide.`
- `--hide_and_localize.`
- `--in_place.`
- `--hide.`
- `--linkview` OR `--no_linkview.` This option is deprecated.
- `--localize.`
- `--rename.`
- `--show.`
- `--show_and_globalize.`
- `--strip.`
- `--show.`
- `--symbolversions` OR `--no_symbolversions.`

Restrictions

You must use `--output` with this option.

Related information

- [--in_place](#) on page 57
- [--output=destination](#) on page 65
- [--strip=option\[,option,...\]](#) on page 73

4.23 --emit=option[,option,...]

Enables you to specify the elements of an ELF object that you want to appear in the textual output. The output includes ELF header and section information.

Syntax

```
--emit=option[,option,...]
```

Where *option* is one of:

addresses

Prints global and static data addresses (including addresses for structure and union contents). It has the same effect as `--text -a`.

This option can only be used on files containing debug information. If no debug information is present, a warning message is generated.

Use the `--select` option to output a subset of the data addresses.

If you want to view the data addresses of arrays, expanded both inside and outside structures, use the `--expandarrays` option with this text category.

build_attributes

Prints the contents of the build attributes section in human-readable form for standard build attributes or raw hexadecimal form for nonstandard build attributes. The produces the same output as the `--decode_build_attributes` option.

code

Disassembles code, alongside a dump of the original binary data being disassembled and the addresses of the instructions. It has the same effect as `--text -c`.



Note

Unlike `--disassemble`, the disassembly cannot be input to the assembler.

data

Prints contents of the data sections. It has the same effect as `--text -d`.

data_symbols

Modifies the output information of data sections so that symbol definitions are interleaved.

debug_info

Prints debug information. It has the same effect as `--text -g`.

dynamic_segment

Prints dynamic segment contents. It has the same effect as `--text -y`.

exception_tables

Decodes AArch32 exception table information for objects. It has the same effect as `--text -e`.

frame_directives

Prints the contents of `FRAME` directives in disassembled code as specified by the debug information embedded in an object module.

Use this option with `--disassemble`.

got

Prints the contents of the Global Offset Table (GOT) section.

heading_comments

Prints heading comments at the beginning of the disassembly containing tool and command-line information from `.comment` sections.

Use this option with `--disassemble`.

raw_build_attributes

Prints the contents of the build attributes section in raw hexadecimal form, that is, in the same form as data.

relocation_tables

Prints relocation information. It has the same effect as `--text -r`.

string_tables

Prints the string tables. It has the same effect as `--text -t`.

summary

Prints a summary of the segments and sections in a file. It is the default output of `fromelf --text`. However, the summary is suppressed by some `--info` options. Use `--emit summary` to explicitly re-enable the summary, if required.

symbol_annotations

Prints symbols in disassembled code and data annotated with comments containing the respective property information.

Use this option with `--disassemble`.

symbol_tables

Prints the symbol and versioning tables. It has the same effect as `--text -s`.

whole_segments

Prints disassembled executables or shared libraries segment by segment even if it has a link view.

Use this option with `--disassemble`.

You can specify multiple options in one *option* followed by a comma-separated list of arguments.

Restrictions

You can use this option only in text mode.

Related information

[--disassemble](#) on page 42

[--decode_build_attributes](#) on page 39

[--expandarrays](#) on page 47

[--text](#) on page 75

4.24 --expandarrays

Prints data addresses, including arrays that are expanded both inside and outside structures.

Restrictions

You can use this option with `--text -a` or with `--fieldoffsets`.

Example

The following example shows the output for a struct containing arrays when `--fieldoffsets --expandarrays` is specified:

```
// foo.c
struct S {
    char A[8];
    char B[4];
};
struct S s;

struct S* get()
{
    return &s;
}
```

```
> armclang -target arm-arm-none-eabi -march=armv8-a -g -c foo.c
> fromelf --fieldoffsets --expandarrays foo.o
```

```
; Structure, S , Size 0xc bytes, from foo.c
|S.A|                EQU    0          ; array[8] of char
|S.A[0]|              EQU    0          ; char
|S.A[1]|              EQU    0x1        ; char
|S.A[2]|              EQU    0x2        ; char
|S.A[3]|              EQU    0x3        ; char
|S.A[4]|              EQU    0x4        ; char
|S.A[5]|              EQU    0x5        ; char
|S.A[6]|              EQU    0x6        ; char
|S.A[7]|              EQU    0x7        ; char
|S.B|                EQU    0x8        ; array[4] of char
|S.B[0]|              EQU    0x8        ; char
|S.B[1]|              EQU    0x9        ; char
|S.B[2]|              EQU    0xa        ; char
|S.B[3]|              EQU    0xb        ; char
; End of Structure S

END
```

Related information

[--fieldoffsets](#) on page 49

[--text](#) on page 75

4.25 --extract_build_attributes

Prints only the build attributes in a form that depends on the type of attribute.

Usage

Prints the build attributes in:

- Human-readable form for standard build attributes.
- Raw hexadecimal form for nonstandard build attributes.

Restrictions

This option has no effect for AArch64 state inputs.

Example

The following example shows the output for `--extract_build_attributes`:

```
> armclang -c -mcpu=cortex-m7 --target=arm-arm-none-eabi -mfpv=vfpv3 hello.c -o
hello.o
> fromelf --cpu=Cortex-M7 --extract_build_attributes hello.o

=====

** Object/Image Build Attributes

'aeabi' file build attributes:
0x000000:  43 32 2e 30 39 00 05 63 6f 72 74 65 78 2d 6d 37      C2.09..cortex-m7
0x000010:  00 06 0d 07 4d 08 00 09 02 0a 05 0e 00 11 01 12      ....M.....
0x000020:  04 14 01 15 01 17 03 18 01 19 01 1a 02 22 00 24      .....".$.
0x000030:  01 26 01                                     .&.
      Tag_conformance = "2.09"
      Tag_CPU_name = "cortex-m7"
      Tag_CPU_arch = ARM v7E-M (=13)
      Tag_CPU_arch_profile = The microcontroller profile 'M' (e.g. for Cortex M3)
(=77)
      Tag_ARM_ISA_use = No ARM instructions were permitted to be used (=0)
      Tag_THUMB_ISA_use = Thumb2 instructions were permitted (implies Thumb
instructions permitted) (=2)
      Tag_VFP_arch = VFPv4 instructions were permitted (implies VFPv3 instructions
were permitted) (=5)
      Tag_ABI_PCS_R9_use = R9 used as V6 (just another callee-saved register) (=0)
      Tag_ABI_PCS_GOT_use = Data are imported directly (=1)
      Tag_ABI_PCS_wchar_t = Size of wchar_t is 4 (=4)
      Tag_ABI_FP_denormal = This code was permitted to require IEEE 754 denormal
numbers (=1)
      Tag_ABI_FP_exceptions = This code was permitted to check the IEEE 754
inexact exception (=1)
      Tag_ABI_FP_number_model = This code may use all the IEEE 754-defined FP
encodings (=3)
      Tag_ABI_align8_needed = Code was permitted to depend on the 8-byte alignment
of 8-byte data items (=1)
      Tag_ABI_align8_preserved = Code was required to preserve 8-byte alignment of
8-byte data objects (=1)
      Tag_ABI_enum_size = Enum containers are 32-bit (=2)
```



```

Tag_CPU_unaligned_access = The producer was not permitted to make unaligned
data accesses (=0)
Tag_VFP_HP_extension = The producer was permitted to use the VFPv3/Advanced
SIMD optional half-precision extension (=1)
Tag_ABI_FP_16bit_format = The producer was permitted to use IEEE 754 format
16-bit floating point numbers (=1)

```

Related information

[--decode_build_attributes](#) on page 39
[--dump_build_attributes](#) on page 43
[--emit=option\[,option,...\]](#) on page 45
[--text](#) on page 75

4.26 --fieldoffsets

Prints a list of `armasm` style assembly language EQU directives that equate C++ class or C structure field names to their offsets from the base of the class or structure.

Usage

The input ELF file can be a relocatable object or an image.

Use `--output` to redirect the output to a file. Use the `INCLUDE` command from `armasm` to load the produced file and provide access to C++ classes and C structure members by name from assembly language.



The `EQU` directives cannot be used with the clang-integrated assembler. To use them, you must change them to GNU syntax.

This option outputs all structure information. To output a subset of the structures, use `--select select_options`.

If you do not require a file that can be input to `armasm`, use the `--text -a` options to format the display addresses in a more readable form. The `-a` option only outputs address information for structures and static data in images because the addresses are not known in a relocatable object.

Restrictions

This option:

- Requires that the object or image file has debug information.
- Can be used in text mode and with `--expandarrays`.

Examples

The following examples show how to use `--fieldoffsets`:

- To produce an output listing to `stdout` that contains all the field offsets from all structures in the file `inputfile.o`, enter:

```
fromelf --fieldoffsets inputfile.o
```

- To produce an output file listing to `outputfile.s` that contains all the field offsets from structures in the file `inputfile.o` that have a name starting with `p`, enter:

```
fromelf --fieldoffsets --select=p* --output=outputfile.s inputfile.o
```

- To produce an output listing to `outputfile.s` that contains all the field offsets from structures in the file `inputfile.o` with names of `tools` or `moretools`, enter:

```
fromelf --fieldoffsets --select=tools.*,moretools.* --output=outputfile.s  
inputfile.o
```

- To produce an output file listing to `outputfile.s` that contains all the field offsets of structure fields whose name starts with `number` and are within structure field `top` in structure `tools` in the file `inputfile.o`, enter:

```
fromelf --fieldoffsets --select=tools.top.number* --output=outputfile.s  
inputfile.o
```

The following is an example of the output, and includes `name.` and `name...member` that arise because of anonymous structs and unions:

```
; Structure, Table , Size 0x104 bytes, from inputfile.cpp
|Table.TableSize|          EQU    0          ; int
|Table.Data|              EQU    0x4          ; array[64] of
|MyClassHandle|
; End of Structure Table
; Structure, Box2 , Size 0x8 bytes, from inputfile.cpp
|Box2.|                  EQU    0          ; anonymous
|Box2..|                 EQU    0          ; anonymous
|Box2...Min|             EQU    0          ; Point2
|Box2...Min.x|           EQU    0          ; short
|Box2...Min.y|           EQU    0x2         ; short
|Box2...Max|             EQU    0x4         ; Point2
|Box2...Max.x|           EQU    0x4         ; short
|Box2...Max.y|           EQU    0x6         ; short
; Warning: duplicate name (Box2..) present in (inputfile.cpp) and in (inputfile.cpp)
; please use the --qualify option
|Box2..|                  EQU    0          ; anonymous
|Box2...Left|            EQU    0          ; unsigned short
|Box2...Top|             EQU    0x2         ; unsigned short
|Box2...Right|           EQU    0x4         ; unsigned short
|Box2...Bottom|          EQU    0x6         ; unsigned short
; End of Structure Box2
; Structure, MyClassHandle , Size 0x4 bytes, from inputfile.cpp
|MyClassHandle.Handle|    EQU    0          ; pointer to MyClass
; End of Structure MyClassHandle
; Structure, Point2 , Size 0x4 bytes, from defects.cpp
|Point2.x|                EQU    0          ; short
|Point2.y|                EQU    0x2         ; short
; End of Structure Point2
; Structure, __fpos_t_struct , Size 0x10 bytes, from <filepath>
|__fpos_t_struct.__pos|   EQU    0          ; unsigned long long
|__fpos_t_struct.__mbstate| EQU    0x8         ; anonymous
|__fpos_t_struct.__mbstate.__state1| EQU    0x8         ; unsigned int
```

```
| __fpos_t_struct._mbstate.__state2| EQU 0xc ; unsigned int
; End of Structure __fpos_t_struct
END
```

Related information

[--expandarrays](#) on page 47

[--qualify](#) on page 67

[--select=select_options](#) on page 70

[--text](#) on page 75

[EQU](#)

[GET or INCLUDE](#)

[Miscellaneous directives](#)

4.27 --fpu=list

Lists the FPU architectures that are supported by the `--fpu=name` option.

Deprecated options are not listed.

Related information

[--fpu=name](#) on page 51

4.28 --fpu=name

Specifies the target FPU architecture.

To obtain a full list of FPU architectures use the `--fpu=list` option.

Default

The default target FPU architecture is derived from use of the `--cpu` option.

If the CPU you specify with `--cpu` has a VFP coprocessor, the default target FPU architecture is the VFP architecture for that CPU.

Syntax

`--fpu=name`

Where *name* is the name of the target FPU architecture. Specify `--fpu=list` to list the supported FPU architecture names that you can use with `--fpu=name`.

The default floating-point architecture depends on the target architecture.



Software floating-point linkage is not supported for AArch64 state.

Usage

This option selects disassembly for a specific FPU architecture. It affects how `fromelf` interprets the instructions it finds in the input files.

If you specify this option, it overrides any implicit FPU option that appears on the command line, for example, where you use the `--cpu` option.

Any FPU explicitly selected using the `--fpu` option always overrides any FPU implicitly selected using the `--cpu` option.

Related information

[--disassemble](#) on page 42

[--fpu=list](#) on page 51

[--info=topic\[,topic,...\]](#) on page 58

[--text](#) on page 75

4.29 --globalize=option[,option,...]

Converts the selected symbols to global symbols.

Syntax

```
--globalize=option[,option,...]
```

Where is one of:

***object_name* :**

All symbols in ELF objects with a name matching *object_name* are converted to global symbols.

{object_name} : symbol_name

All symbols in ELF objects with a name matching *object_name* and also a symbol name matching *symbol_name* are converted to global symbols.

symbol_name

All symbols with a symbol name matching *symbol_name* are converted to global symbols.

You can:

- Use wildcard characters `?` and `*` for symbolic names in *symbol_name* and *object_name* arguments
- Specify multiple values in one *option* followed by a comma-separated list of arguments.

Restrictions

You must use `--elf` with this option.

Related information

[--elf](#) on page 44

[--hide=option\[,option,...\]](#) on page 53

4.30 --help

Displays a summary of the main command-line options.

Default

This is the default if you specify `fromelf` without any options or source files.

Related information

[--show_cmdline](#) on page 72

[--version_number](#) on page 78

[--vsn](#) on page 80

4.31 --hide=option[,option,...]

Changes the symbol visibility property to mark selected symbols as hidden.

Syntax

`--hide=option[,option,...]`

Where *option* is one of:

***object_name* :**

All symbols in ELF objects with a name matching *object_name*.

{object_name} : symbol_name

All symbols in ELF objects with a name matching *object_name* and also a symbol name matching *symbol_name*.

symbol_name

All symbols with a symbol name matching *symbol_name*.

You can:

- Use wildcard characters `?` and `*` for symbolic names in *symbol_name* and *object_name* arguments.
- Specify multiple values in one *option* followed by a comma-separated list of arguments.

Restrictions

You must use `--elf` with this option.

Related information

[--elf](#) on page 44

[--show=option\[,option,...\]](#) on page 71

4.32 --hide_and_localize=option[,option,...]

Changes the symbol visibility property to mark selected symbols as hidden, and converts the selected symbols to local symbols.

Syntax

```
--hide_and_localize=option[,option,...]
```

Where *option* is one of:

***object_name* :**

All symbols in ELF objects with a name matching *object_name* are marked as hidden and converted to local symbols.

{*object_name* : : *symbol_name*}

All symbols in ELF objects with a name matching *object_name* and also a symbol name matching *symbol_name* are marked as hidden and converted to local symbols.

symbol_name

All symbols with a symbol name matching *symbol_name* are marked as hidden and converted to local symbols.

You can:

- Use wildcard characters `?` and `*` for symbolic names in *symbol_name* and *object_name* arguments.
- Specify multiple values in one *option* followed by a comma-separated list of arguments.

Restrictions

You must use `--elf` with this option.

Related information

[--elf](#) on page 44

4.33 --i32

Produces Intel Hex-32 format output. It generates one output file for each load region in the image.

You can specify the base address of the output with the `--base` option.

Restrictions

The following restrictions apply:

- Not supported for AArch64 state.
- You cannot use this option with object files.
- You must use `--output` with this option.

Considerations when using `--i32`

If you convert an ELF image containing multiple load regions to a binary format, `fromelf` creates an output directory named *destination* and generates one binary output file for each load region in the input image. `fromelf` places the output files in the *destination* directory.



For multiple load regions, the name of the first non-empty execution region in the corresponding load region is used for the filename.

A file is only created when the load region describes code or data that is present in the ELF file. For example a load region containing only execution regions with ZI data in them does not result in an output file.

Example

To convert the ELF file `infile.axf` to an Intel Hex-32 format file, for example `outfile.bin`, enter:

```
fromelf --i32 --output=outfile.bin infile.axf
```

Related information

`--base [[object_file::]load_region_ID=num]` on page 27

`--i32combined` on page 55

`--output=destination` on page 65

4.34 `--i32combined`

Produces Intel Hex-32 format output. It generates one output file for an image containing multiple load regions.

You can specify the base address of the output with the `--base` option.

Restrictions

The following restrictions apply:

- Not supported for AArch64 state.
- You cannot use this option with object files.
- You must use `--output` with this option.

Considerations when using --i32combined

If you convert an ELF image containing multiple load regions to a binary format, `fromelf` creates an output directory named *destination* and generates one binary output file for all load regions in the input image. `fromelf` places the output file in the *destination* directory.

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

Example

To create a single output file, `outfile2.bin`, from an image file `infile2.axf`, with two load regions, and with a start address of `0x1000`, enter:

```
fromelf --i32combined --base=0x1000 --output=outfile2.bin infile2.axf
```

Related information

`--base [[object_file::]load_region_ID=num]` on page 27

`--i32` on page 54

`--output=destination` on page 65

4.35 --ignore_section=option[,option,...]

Specifies the sections to be ignored during a compare. Differences between the input files being compared are ignored if they are in these sections.

Syntax

```
--ignore_section=option[,option,...]
```

Where *option* is one of:

object_name::

All sections in ELF objects with a name matching *object_name*.

object_name::section_name

All sections in ELF objects with a name matching *object_name* and also a section name matching *section_name*.

section_name

All sections with a name matching *section_name*.

You can:

- Use wildcard characters `?` and `*` for symbolic names in *symbol_name* and *object_name* arguments.
- Specify multiple values in one *option* followed by a comma-separated list of arguments.

Restrictions

You must use `--compare` with this option.

Related information

`--compare=option[,option,...]` on page 34

`--ignore_symbol=option[,option,...]` on page 57

`--relax_section=option[,option,...]` on page 68

4.36 --ignore_symbol=option[,option,...]

Specifies the symbols to be ignored during a compare. Differences between the input files being compared are ignored if they are related to these symbols.

Syntax

`--ignore_symbol=option[,option,...]`

Where *option* is one of:

***object_name* :**

All symbols in ELF objects with a name matching *object_name*.

object_name* : *symbol_name

All symbols in ELF objects with a name matching *object_name* and also all symbols with names matching *symbol_name*.

symbol_name

All symbols with names matching *symbol_name*.

You can:

- Use wildcard characters `?` and `*` for symbolic names in *symbol_name* and *object_name* arguments
- Specify multiple values in one *option* followed by a comma-separated list of arguments.

Restrictions

You must use `--compare` with this option.

Related information

`--compare=option[,option,...]` on page 34

`--ignore_section=option[,option,...]` on page 56

`--relax_symbol=option[,option,...]` on page 68

4.37 --in_place

Enables the translation of ELF members in an input file to overwrite the previous content.

Restrictions

You must use `--elf` with this option.

Example

To remove debug information from members of the library file `test.a`, enter:

```
fromelf --elf --in_place --strip=debug test.a
```

Related information

[--elf](#) on page 44

[--strip=option\[,option,...\]](#) on page 73

4.38 --info=topic[,topic,...]

Prints information about specific topics.

Syntax

```
--info=topic[,topic,...]
```

Where *topic* is a comma-separated list from the following topic keywords:

instruction_usage

Categorizes and lists the A32 and T32 instructions defined in the code sections of each input file.



Not supported for AArch64 state.

function_sizes

Lists the names of the global functions defined in one or more input files, together with their sizes in bytes and whether they are A32 or T32 functions.

function_sizes_all

Lists the names of the local and global functions defined in one or more input files, together with their sizes in bytes and whether they are A32 or T32 functions.

sizes

Lists the Code, RO Data, RW Data, ZI Data, and Debug sizes for each input object and library member in the image. Using this option implies `--info=sizes,totals`.

totals

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



Code related sizes also include the size of any execute-only code.

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



Spaces are not permitted between topic keywords in the list. For example, you can enter `--info=sizes,totals` but not `--info=sizes, totals`.

Restrictions

You can use this option only in text mode.

Related information

[--text](#) on page 75

4.39 input_file

Specifies the ELF file or archive containing ELF files to be processed.

Usage

Multiple input files are supported if you:

- Output `--text` format.
- Use the `--compare` option.
- Use `--elf` with `--in_place`.
- Specify an output directory using `--output`.

If *input_file* is a scatter-loaded image that contains more than one load region and the output format is one of `--bin`, `--cad`, `--m32`, `--i32`, or `--vhx`, then `fromelf` creates a separate file for each load region.

If *input_file* is a scatter-loaded image that contains more than one load region and the output format is one of `--cadcombined`, `--m32combined`, or `--i32combined`, then `fromelf` creates a single file containing all load regions.

If *input_file* is an archive, you can process all files, or a subset of files, in that archive. To process a subset of files in the archive, specify a filter after the archive name as follows:

```
archive.a(filter_pattern)
```

where *filter_pattern* specifies a member file. To specify a subset of files use the following wildcard characters:

★

Matches zero or more characters.

?

Matched any single character.



Note

On Unix systems your shell typically requires the parentheses and these characters to be escaped with backslashes. Alternatively, enclose the archive name and filter in single quotes, for example:

```
archive.a\(\?\?str*\)
'archive.a(??str*)'
```

Any files in the archive that are not processed are included in the output archive together with the processed files.

Example

To convert all files in the archive beginning with s, and create a new archive, *my_archive.a*, containing the processed and unprocessed files, enter:

```
fromelf archive.a(s*.o) --output=my_archive.a
```

Related information

[Examples of processing ELF files in an archive](#) on page 20

[--bin](#) on page 28

[--cad](#) on page 32

[--cadcombined](#) on page 33

[--compare=option\[,option,...\]](#) on page 34

[--elf](#) on page 44

[--i32](#) on page 54

[--i32combined](#) on page 55

[--in_place](#) on page 57

[--m32](#) on page 63

[--m32combined](#) on page 64

[--output=destination](#) on page 65

[--text](#) on page 75

[--vhx](#) on page 79

4.40 --interleave=option

Inserts the original source code as comments into the disassembly if debug information is present.

Default

The default is `--interleave=none`.

Syntax

`--interleave=option`

Where *option* can be one of the following:

line_directives

Interleaves `#line` directives containing filenames and line numbers of the disassembled instructions.

line_numbers

Interleaves comments containing filenames and line numbers of the disassembled instructions.

none

Disables interleaving. This is useful if you have a generated makefile where the `fromelf` command has multiple options in addition to `--interleave`. You can then specify `--interleave=none` as the last option to ensure that interleaving is disabled without having to reproduce the complete `fromelf` command.

source

Interleaves comments containing source code. If the source code is no longer available then `fromelf` interleaves in the same way as `line_numbers`.

source_only

Interleaves comments containing source code. If the source code is no longer available then `fromelf` does not interleave that code.

Usage

Use this option with `--emit=code`, `--text -c`, Or `--disassemble`.

Use this option with `--source_directory` if you want to specify additional paths to search for source code.

Related information

[--disassemble](#) on page 42

[--emit=option\[,option,...\]](#) on page 45

[--source_directory=path](#) on page 73

[--text](#) on page 75

4.41 --linkview, --no_linkview

Controls the section-level view from the ELF image.

Usage

`--no_linkview` discards the section-level view and retains only the segment-level view (load time view).

Discarding the section-level view eliminates:

- The section header table.
- The section header string table.
- The string table.
- The symbol table.
- All debug sections.

All that is left in the output is the program header table and the program segments.



This option is deprecated.

Restrictions

The following restrictions apply:

You must use `--elf` with `--linkview` and `--no_linkview`.

Example

To get ELF format output for `image.axf`, enter:

```
fromelf --no_linkview --elf image.axf --output=image_nlk.axf
```

Related information

[--elf](#) on page 44

[--privacy](#) on page 66

[--strip=option\[,option,...\]](#) on page 73

[--privacy linker option](#)

4.42 --localize=option[,option,...]

Converts the selected symbols to local symbols.

Syntax

`--localize=option[,option,...]`

Where *option* is one of:

***object_name* :**

All symbols in ELF objects with a name matching *object_name* are converted to local symbols.

{*object_name* : : *symbol_name*}

All symbols in ELF objects with a name matching *object_name* and also a symbol name matching *symbol_name* are converted to local symbols.

symbol_name

All symbols with a symbol name matching *symbol_name* are converted to local symbols.

You can:

- Use wildcard characters ? and * for symbolic names in *symbol_name* and *object_name* arguments.
- Specify multiple values in one *option* followed by a comma-separated list of arguments.

Restrictions

You must use `--elf` with this option.

Related information

[--elf](#) on page 44

[--hide=option\[,option,...\]](#) on page 53

4.43 --m32

Produces Motorola 32-bit format (32-bit S-records) output. It generates one output file for each load region in the image.

You can specify the base address of the output with the `--base` option.

Restrictions

The following restrictions apply:

- Not supported for AArch64 state.
- You cannot use this option with object files.
- You must use `--output` with this option.

Considerations when using --m32

If you convert an ELF image containing multiple load regions to a binary format, `fromelf` creates an output directory named *destination* and generates one binary output file for each load region in the input image. `fromelf` places the output files in the *destination* directory.



Note

For multiple load regions, the name of the first non-empty execution region in the corresponding load region is used for the filename.

A file is only created when the load region describes code or data that is present in the ELF file. For example a load region containing only execution regions with ZI data in them does not result in an output file.

Example

To convert the ELF file `infile.axf` to a Motorola 32-bit format file, for example `outfile.bin`, enter:

```
fromelf --m32 --output=outfile.bin infile.axf
```

Related information

[--base \[\[object_file::\]load_region_ID=num\]](#) on page 27

[--m32combined](#) on page 64

[--output=destination](#) on page 65

4.44 --m32combined

Produces Motorola 32-bit format (32-bit S-records) output. It generates one output file for an image containing multiple load regions.

You can specify the base address of the output with the `--base` option.

Restrictions

The following restrictions apply:

- Not supported for AArch64 state.
- You cannot use this option with object files.
- You must use `--output` with this option.

Considerations when using --m32combined

If you convert an ELF image containing multiple load regions to a binary format, `fromelf` creates an output directory named *destination* and generates one binary output file for all load regions in the input image. `fromelf` places the output file in the *destination* directory.

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

Example

To create a single Motorola 32-bit format output file, `outfile2.bin`, from an image file `infile2.axf`, with two load regions, and with a start address of `0x1000`, enter:

```
fromelf --m32combined --base=0x1000 --output=outfile2.bin infile2.axf
```

Related information

[--base \[\[object_file::\]load_region_ID=num](#) on page 27

[--m32](#) on page 63

[--output=destination](#) on page 65

4.45 --only=section_name

Filters the list of sections that are displayed in the main section-by-section output from `--text`. It does not affect any additional output after the main section-by-section output.

Syntax

`--only=section_name`

Where *section_name* is the name of the section to be displayed.

You can:

- Use wildcard characters `?` and `*` for a section name.
- Use multiple `--only` options to specify additional sections to display.

Examples

The following examples show how to use `--only`:

- To display only the symbol table, `.symtab`, from the section-by-section output, enter:

```
fromelf --only=.symtab --text -s test.axf
```

- To display all `ERn` sections, enter:

```
fromelf --only=ER? test.axf
```

- To display the `HEAP` section and all symbol and string table sections, enter:

```
fromelf --only=HEAP --only=.*tab --text -s -t test.axf
```

Related information

[--text](#) on page 75

4.46 --output=destination

Specifies the name of the output file, or the name of the output directory if multiple output files are created.

Syntax

`--output=destination`

`--o destination`

Where *destination* can be either a file or a directory. For example:

`--output=foo`

is the name of an output file

`--output=foo/`

is the name of an output directory.

Usage

Usage with `--bin` or `--elf`:

- You can specify a single input file and a single output filename.
- If you specify many input files and use `--elf`, you can use `--in_place` to write the output of processing each file over the top of the input file.
- If you specify many input filenames and specify an output directory, then the output from processing each file is written into the output directory. Each output filename is derived from the corresponding input file. Therefore, specifying an output directory in this way is the only method of converting many ELF files to a binary or hexadecimal format in a single run of `fromelf`.
- If you specify an archive file as the input, then the output file is also an archive. For example, the following command creates an archive file called `output.o`:

```
fromelf --elf --strip=debug archive.a --output=output.o
```

- If you specify a pattern in parentheses to select a subset of objects from an archive, `fromelf` only converts the subset. All the other objects are passed through to the output archive unchanged.

Related information

[--bin](#) on page 28

[--elf](#) on page 44

[--text](#) on page 75

4.47 --privacy

Modifies the output file to protect your code in images and objects that are delivered to third parties.

Usage

The effect of this option is different for images and object files.

For images, this option:

- Changes section names to a default value, for example, changes code section names to `.text`
- Removes the complete symbol table in the same way as `--strip symbols`
- Removes the `.comment` section name, and is marked as `[Anonymous Section]` in the `fromelf --text` output.

For object files, this option:

- Changes section names to a default value, for example, changes code section names to `.text`.
- Keeps mapping symbols and build attributes in the symbol table.
- Removes those local symbols that can be removed without loss of functionality.

Symbols that cannot be removed, such as the targets for relocations, are kept. For these symbols, the names are removed. These are marked as `[Anonymous Symbol]` in the `fromelf --text` output.

Related information

[--strip=option\[,option,...\]](#) on page 73

[--locals, --no_locals linker option](#)

[--privacy linker option](#)

4.48 --qualify

Modifies the effect of the `--fieldoffsets` option so that the name of each output symbol includes an indication of the source file containing the relevant structure.

Usage

This enables the `--fieldoffsets` option to produce functional output even if two source files define different structures with the same name.

If the source file is in a different location from the current location, then the source file path is also included.

Examples

A structure called `foo` is defined in two headers for example, `one.h` and `two.h`.

Using `fromelf` option `--fieldoffsets`, the linker might define the following symbols:

- `foo.a`, `foo.b`, and `foo.c`.
- `foo.x`, `foo.y`, and `foo.z`.

Using `fromelf` options `--qualify --fieldoffsets`, the linker defines the following symbols:

- `oneh_foo.a`, `oneh_foo.b` and `oneh_foo.c`.
- `twoh_foo.x`, `twoh_foo.y` and `twoh_foo.z`.

Related information

[--fieldoffsets](#) on page 49

4.49 --relax_section=option[,option,...]

Changes the severity of a compare report for the specified sections to warnings rather than errors.

Restrictions

You must use `--compare` with this option.

Syntax

```
--relax_section=option[,option,...]
```

Where *option* is one of:

***object_name* :**

All sections in ELF objects with a name matching *object_name*.

{object_name} : section_name

All sections in ELF objects with a name matching *object_name* and also a section name matching *section_name*.

section_name

All sections with a name matching *section_name*.

You can:

- Use wildcard characters `?` and `*` for symbolic names in *section_name* and *object_name* arguments
- Specify multiple values in one *option* followed by a comma-separated list of arguments.

Related information

[--compare=option\[,option,...\]](#) on page 34

[--ignore_section=option\[,option,...\]](#) on page 56

[--relax_symbol=option\[,option,...\]](#) on page 68

4.50 --relax_symbol=option[,option,...]

Changes the severity of a compare report for the specified symbols to warnings rather than errors.

Syntax

```
--relax_symbol=option[,option ,...]
```

Where *option* is one of:

object_name::

All symbols in ELF objects with a name matching *object_name*.

{object_name::symbol_name}

All symbols in ELF objects with a name matching *object_name* and also a symbol name matching *symbol_name*.

symbol_name

All symbols with a name matching *symbol_name*.

You can:

- Use wildcard characters ? and * for symbolic names in *symbol_name* and *object_name* arguments
- Specify multiple values in one *option* followed by a comma-separated list of arguments.

Restrictions

You must use `--compare` with this option.

Related information

[--compare=option\[,option,...\]](#) on page 34

[--ignore_symbol=option\[,option,...\]](#) on page 57

[--relax_section=option\[,option,...\]](#) on page 68

4.51 --rename=option[,option,...]

Renames the specified symbol in an output ELF object.

Syntax

```
--rename=option[,option ,...]
```

Where *option* is one of:

{object_name::old_symbol_name=new_symbol_name}

This replaces all symbols in the ELF object *object_name* that have a symbol name matching *old_symbol_name*.

old_symbol_name=new_symbol_name

This replaces all symbols that have a symbol name matching *old_symbol_name*.

You can:

- Use wildcard characters `?` and `*` for symbolic names in *old_symbol_name*, *new_symbol_name*, and *object_name* arguments.
- Specify multiple values in one *option* followed by a comma-separated list of arguments.

Restrictions

You must use `--elf` and `--output` with this option.

Example

This example renames the `clock` symbol in the `timer.axf` image to `myclock`, and creates a new file called `mytimer.axf`:

```
fromelf --elf --rename=clock=myclock --output=mytimer.axf timer.axf
```

Related information

[--elf](#) on page 44

[--output=destination](#) on page 65

4.52 --select=select_options

When used with `--fieldoffsets` or `--text -a` options, displays only those fields that match a specified pattern list.

Syntax

```
--select=select_options
```

Where *select_options* is a list of patterns to match. Use special characters to select multiple fields:

- Use a comma-separated list to specify multiple fields, for example:
`a*,b*,c*`
- Use the wildcard character `*` to match any name.
- Use the wildcard character `?` to match any single letter.
- Prefix the *select_options* string with `+` to specify the fields to include. This is the default behavior.
- Prefix the *select_options* string with `~` to specify the fields to exclude.

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

Usage

Use this option with either `--fieldoffsets` or `--text -a`.

Example

The output from the `--fieldoffsets` option might include the following data structure:

structure.f1	EQU	0	; int16_t
structure.f2	EQU	0x2	; int16_t
structure.f3	EQU	0x4	; int16_t
structure.f11	EQU	0x6	; int16_t
structure.f21	EQU	0x8	; int16_t
structure.f31	EQU	0xA	; int16_t
structure.f111	EQU	0xC	; int16_t

To output only those fields that start with `f1`, enter:

```
fromelf --select=structure.f1* --fieldoffsets infile.axf
```

This produces the output:

structure.f1	EQU	0	; int16_t
structure.f11	EQU	0x6	; int16_t
structure.f111	EQU	0xC	; int16_t
END			

Related information

[--fieldoffsets](#) on page 49

[--text](#) on page 75

4.53 --show=option[,option,...]

Changes the symbol visibility property of the selected symbols, to mark them with default visibility.

Syntax

```
--show=option[,option,...]
```

Where *option* is one of:

***object_name* :**

All symbols in ELF objects with a name matching *object_name* are marked as having default visibility.

{*object_name* : : *symbol_name*}

All symbols in ELF objects with a name matching *object_name* and also a symbol name matching *symbol_name* are marked as having default visibility.

symbol_name

All symbols with a symbol name matching *symbol_name* are marked as having default visibility.

You can:

- Use wildcard characters ? and * for symbolic names in *symbol_name* and *object_name* arguments
- Specify multiple values in one *option* followed by a comma-separated list of arguments.

Restrictions

You must use `--elf` with this option.

Related information

[--elf](#) on page 44

[--hide=option\[,option,...\]](#) on page 53

4.54 --show_and_globalize=option[,option,...]

Changes the symbol visibility property of the selected symbols, to mark them with default visibility, and converts the selected symbols to global symbols.

Syntax

```
--show_and_globalize=option[,option,...]
```

Where *option* is one of:

***object_name*::**

All symbols in ELF objects with a name matching *object_name*.

{*object_name*::*symbol_name*}

All symbols in ELF objects with a name matching *object_name* and also a symbol name matching *symbol_name*.

symbol_name

All symbols with a symbol name matching *symbol_name*.

You can:

- Use wildcard characters ? and * for symbolic names in *symbol_name* and *object_name* arguments
- Specify multiple values in one *option* followed by a comma-separated list of arguments.

Restrictions

You must use `--elf` with this option.

Related information

[--elf](#) on page 44

4.55 --show_cmdline

Outputs the command line used by the ELF file converter.

Usage

Shows the command line after processing by the ELF file converter, and can be useful to check:

- The command line a build system is using.
- How the ELF file converter is interpreting the supplied command line, for example, the ordering of command-line options.

The commands are shown normalized, and the contents of any via files are expanded.

The output is sent to the standard error stream (`stderr`).

Related information

[--via=file](#) on page 79

4.56 --source_directory=path

Explicitly specifies the directory of the source code.

Syntax

`--source_directory=path`

Usage

By default, the source code is assumed to be located in a directory relative to the ELF input file. You can use this option multiple times to specify a search path involving multiple directories.

You can use this option with `--interleave`.

Related information

[--interleave=option](#) on page 61

4.57 --strip=option[,option,...]

Helps to protect your code in images and objects that are delivered to third parties. You can also use it to help reduce the size of the output image.

Syntax

`--strip=option[,option,...]`

Where *option* is one of:

all

For object modules, this option removes all debug, comments, notes and symbols from the ELF file. For executables, this option works the same as `--no_linkview`.

debug

Removes all debug sections from the ELF file.

comment

Removes the `.comment` section from the ELF file.

filesymbols

The `STT_FILE` symbols are removed from the ELF file.

localsymbols

The effect of this option is different for images and object files.

For images, this option removes all local symbols, including mapping symbols, from the output symbol table.

For object files, this option:

- Keeps mapping symbols and build attributes in the symbol table.
- Removes those local symbols that can be removed without loss of functionality.

Symbols that cannot be removed, such as the targets for relocations, are kept. For these symbols, the names are removed. These are marked as `[Anonymous Symbol]` in the `fromelf --text` output.

notes

Removes the `.notes` section from the ELF file.

pathnames

Removes the path information from all symbols with type `STT_FILE`. For example, an `STT_FILE` symbol with the name `C:\work\myobject.o` is renamed to `myobject.o`.



This option does not strip path names that are in the debug information.

symbols

The effect of this option is different for images and object files.

For images, this option removes the complete symbol table, and all static symbols. If any of these static symbols are used as a static relocation target, then these relocations are also removed. In all cases, `STT_FILE` symbols are removed.

For object files, this option:

- Keeps mapping symbols and build attributes in the symbol table.

- Removes those local symbols that can be removed without loss of functionality.

Symbols that cannot be removed, such as the targets for relocations, are kept. For these symbols, the names are removed. These are marked as `[Anonymous Symbol]` in the `fromelf --text` output.



Stripping the symbols, path names, or file symbols might make the file harder to debug.

Restrictions

You must use `--elf` and `--output` with this option.

Example

To produce an `output.axf` file without debug from the ELF file `infile.axf` originally produced with debug, enter:

```
fromelf --strip=debug,symbols --elf --output=outfile.axf infile.axf
```

Related information

[--elf](#) on page 44

[--linkview](#), [--no_linkview](#) on page 62

[--privacy](#) on page 66

[About mapping symbols](#)

[--locals](#), [--no_locals](#) linker option

[--privacy](#) linker option

4.58 --symbolversions, --no_symbolversions

Turns off the decoding of symbol version tables.

Restrictions

If you use `--elf` with this option, you must also use `--output`.

Related information

[Symbol versioning](#)

[Base Platform ABI for the Arm Architecture](#)

4.59 --text

Prints image information in text format. You can decode an ELF image or ELF object file using this option.

Syntax

```
--text [options]
```

Where *options* specifies what is displayed, and can be one or more of the following:

-a

Prints the global and static data addresses (including addresses for structure and union contents).

This option can only be used on files containing debug information. If no debug information is present, a warning is displayed.

Use the `--select` option to output a subset of fields in a data structure.

If you want to view the data addresses of arrays, expanded both inside and outside structures, use the `--expandarrays` option with this text category.

-c

This option disassembles code, alongside a dump of the original binary data being disassembled and the addresses of the instructions.



Note

Disassembly is generated in `armasm` assembler syntax and not GNU assembler syntax.

Unlike `--disassemble`, the disassembly cannot be input to the assembler.

-d

Prints contents of the data sections.

-e

Decodes exception table information for objects. Use with `-c` when disassembling images.



Note

Not supported for AArch64 state.

-g

Prints debug information.

-r

Prints relocation information.

- s**
Prints the symbol and versioning tables.
- t**
Prints the string tables.
- v**
Prints detailed information on each segment and section header of the image.
- w**
Eliminates line wrapping.
- y**
Prints dynamic segment contents.
- z**
Prints the code and data sizes.

These options are only recognized in text mode.

Usage

If you do not specify a code output format, `--text` is assumed. That is, you can specify one or more options without having to specify `--text`. For example, `fromelf -a` is the same as `fromelf --text -a`.

If you specify a code output format, such as `--bin`, then any `--text` options are ignored.

If *destination* is not specified with the `--output` option, or `--output` is not specified, the information is displayed on `stdout`.

Use the `--only` option to filter the list of sections.

Examples

The following examples show how to use `--text`:

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

```
fromelf --text -c -s --output=outfile.lst infile.axf
```

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

```
fromelf -a --select=* infile.axf
```

- To produce a text file containing all of the structure addresses in `infile.axf` but none of the global or static data variable information, enter:

```
fromelf --text -a --select=*. * --output=structaddress.txt infile.axf
```

- To produce a text file containing addresses of the nested structures only, enter:

```
fromelf --text -a --select=*.*. * --output=structaddress.txt infile.axf
```

- To produce a text file containing all of the global or static data variable information in `infile.axf` but none of the structure addresses, enter:

```
fromelf --text -a --select=*,~*. * --output=structaddress.txt infile.axf
```

- To output only the `.symtab` section information in `infile.axf`, enter:

```
fromelf --only .symtab -s --output=symtab.txt infile.axf
```

Related information

Using `fromelf` to find where a symbol is placed in an executable ELF image on page 24

`--cpu=name` on page 35

`--disassemble` on page 42

`--emit=option[,option,...]` on page 45

`--expandarrays` on page 47

`--info=topic[,topic,...]` on page 58

`--interleave=option` on page 61

`--only=section_name` on page 65

`--output=destination` on page 65

`--select=select_options` on page 70

`-w` on page 80

Linker options for getting information about images

4.60 --version_number

Displays the version of `fromelf` you are using.

Usage

The ELF file converter displays the version number in the format `Mmmuuxx`, where:

- `M` is the major version number, 6.
- `mm` is the minor version number.
- `uu` is the update number.
- `xx` is reserved for Arm internal use. You can ignore this for the purposes of checking whether the current release is a specific version or within a range of versions.

Related information

`--help` on page 53

`--vsn` on page 80

4.61 --vhx

Produces Byte oriented (Verilog Memory Model) hexadecimal format output.

Usage

This format is suitable for loading into the memory models of `Hardware Description Language` (HDL) simulators. You can split output from this option into multiple files with the `--widthxbanks` option.

Restrictions

The following restrictions apply:

- You cannot use this option with object files.
- You must use `--output` with this option.

Considerations when using --vhx

If you convert an ELF image containing multiple load regions to a binary format, `fromelf` creates an output directory named *destination* and generates one binary output file for each load region in the input image. `fromelf` places the output files in the *destination* directory.



For multiple load regions, the name of the first non-empty execution region in the corresponding load region is used for the filename.

A file is only created when the load region describes code or data that is present in the ELF file. For example a load region containing only execution regions with ZI data in them does not result in an output file.

Examples

To convert the ELF file `infile.axf` to a byte oriented hexadecimal format file, for example `outfile.bin`, enter:

```
fromelf --vhx --output=outfile.bin infile.axf
```

To create multiple output files, in the `regions` directory, from an image file `multiload.axf`, with two 8-bit memory banks, enter:

```
fromelf --vhx --8x2 multiload.axf --output=regions
```

Related information

[--output=destination](#) on page 65

[--widthxbanks](#) on page 82

4.62 --via=file

Reads an additional list of input filenames and ELF file converter options from filename.

Syntax

`--via=filename`

Where *filename* is the name of a via file containing options to be included on the command line.

Usage

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

Related information

[Overview of via files](#) on page 84

[Via file syntax rules](#) on page 84

4.63 --vsn

Displays the version information and the license details.



`--vsn` is intended to report the version information for manual inspection. The Component line indicates the release of Arm® Compiler you are using. If you need to access the version in other tools or scripts, for example in build scripts, use the output from `--version_number`.

Example

```
> fromelf --vsn
Product: ARM Compiler N.n
Component: ARM Compiler N.n
Tool: fromelf [tool_id]
license_type
Software supplied by: ARM Limited
```

Related information

[--help](#) on page 53

[--version_number](#) on page 78

4.64 -w

Causes some text output information that usually appears on multiple lines to be displayed on a single line.

Usage

This makes the output easier to parse with text processing utilities such as Perl.

Example

```
> fromelf --text -w -c test.axf
=====
** ELF Header Information
.
.
.
=====
** Section #1 '.text' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]      Size      : 36
   bytes (alignment 4)      Address: 0x00000000      $a
      .text
.
.
.
** Section #7 '.rel.text' (SHT_REL)      Size      : 8 bytes (alignment 4)      Symbol
   table #6 '.symtab'      1 relocations applied to section #1 '.text'
** Section #2 '.ARM.exidx' (SHT_ARM_EXIDX) [SHF_ALLOC + SHF_LINK_ORDER]      Size      :
   8 bytes (alignment 4)      Address: 0x
00000000      Link to section #1 '.text'
** Section #8 '.rel.ARM.exidx' (SHT_REL)      Size      : 8 bytes (alignment 4)      Symbol
   table #6 '.symtab'      1 relocations applied to section #2 '.ARM.exidx'
** Section #3 '.arm_vfe_header' (SHT_PROGBITS)      Size      : 4 bytes (alignment 4)
** Section #4 '.comment' (SHT_PROGBITS)      Size      : 74 bytes
** Section #5 '.debug_frame' (SHT_PROGBITS)      Size      : 140 bytes
** Section #9 '.rel.debug_frame' (SHT_REL)      Size      : 32 bytes (alignment 4)
   Symbol table #6 '.symtab'      4 relocations applied to section #5 '.debug_frame'
** Section #6 '.symtab' (SHT_SYMTAB)      Size      : 176 bytes (alignment 4)      String
   table #11 '.strtab'      Last local symbol no. 5
** Section #10 '.shstrtab' (SHT_STRTAB)      Size      : 110 bytes
** Section #11 '.strtab' (SHT_STRTAB)      Size      : 223 bytes
** Section #12 '.ARM.attributes' (SHT_ARM_ATTRIBUTES)      Size      : 69 bytes
```

Related information

[--text](#) on page 75

4.65 --wide64bit

Causes all addresses to be displayed with a width of 64 bits.

Usage

Without this option fromelf displays addresses as 32 bits where possible, and only displays them as 64 bits when necessary.

This option is ignored if the input file is not an AArch64 state file.

Related information

[input_file](#) on page 59

4.66 --widthxbanks

Outputs multiple files for multiple memory banks.

Syntax

`--widthxbanks`

Where:

banks

specifies the number of memory banks in the target memory system. It determines the number of output files that are generated for each load region.

width

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

Valid configurations are:

```
--8x1
--8x2
--8x4
--16x1
--16x2
--32x1
--32x2
--64x1
```

Usage

`fromelf` uses the last specified configuration if more than one configuration is specified.

If the image has one load region, `fromelf` generates the same number of files as the number of *banks* specified. The filenames are derived from the `--output=destination` argument, using the following naming conventions:

- If there is one memory bank (*banks*=1) the output file is named *destination*.
- If there are multiple memory banks (*banks*>1), `fromelf` generates *banks* number of files named *destinationN* where *N* is in the range 0 to *banks*-1. If you specify a file extension for the output filename, then the number *N* is placed before the file extension. For example:

```
fromelf --cpu=8-A.32 --vbx --8x2 test.axf --output=test.txt
```

This generates two files named `test0.txt` and `test1.txt`.

If the image has multiple load regions, `fromelf` creates a directory named *destination* and generates *banks* files for each load region in that directory. The files for each load region are named

`load_regionN` where `load_region` is the name of the load region, and `N` is in the range 0 to `banks-1`. For example:

```
fromelf --cpu=8-A.32 --vhx --8x2 multiload.axf --output=regions/
```

This might produce the following files in the `regions` directory:

```
EXEC_ROM0
EXEC_ROM1
RAM0
RAM1
```

The memory width specified by `width` controls the amount of memory that is stored in a single line of each output file. The size of each output file is the size of memory to be read divided by the number of files created. For example:

- `fromelf --cpu=8-A.32 --vhx --8x4 test.axf --output=file` produces four files (`file0`, `file1`, `file2`, and `file3`). Each file contains lines of single bytes, for example:

```
00
00
2D
00
2C
8F
...
```

- `fromelf --vhx --16x2 test.axf --output=file` produces two files (`file0` and `file1`). Each file contains lines of two bytes, for example:

```
0000
002D
002C
...
```

Restrictions

You must use `--output` with this option.

Related information

[--bin](#) on page 28

[--output=destination](#) on page 65

[--vhx](#) on page 79

5. Via File Syntax

Describes the syntax of via files accepted by the `armasm`, `armlink`, `fromelf`, and `armar` tools.

5.1 Overview of via files

Via files are plain text files that allow you to specify command-line arguments and options for the `armasm`, `armlink`, `fromelf`, and `armar` tools.

Typically, you use a via file to overcome the command-line length limitations. However, you might want to create multiple via files that:

- Group similar arguments and options together.
- Contain different sets of arguments and options to be used in different scenarios.



In general, you can use a via file to specify any command-line option to a tool, including `--via`. Therefore, you can call multiple nested via files from within a via file.

Via file evaluation

When you invoke the `armasm`, `armlink`, `fromelf`, or `armar`, the tool:

1. Replaces the first specified `--via via_file` argument with the sequence of argument words that are extracted from the via file, including recursively processing any nested `--via` commands in the via file.
2. Processes any subsequent `--via via_file` arguments in the same way, in the order they are presented.

That is, via files are processed in the order that you specify them. Each via file is processed completely, including any nested via files contained in that file, before processing the next via file.

Related information

[Via file syntax rules](#) on page 84

[--via=file](#) on page 79

5.2 Via file syntax rules

Via files must conform to some syntax rules.

- A via file is a text file containing a sequence of words. Each word in the text file is converted into an argument string and passed to the tool.

- Words are separated by whitespace, or the end of a line, except in delimited strings, for example:

```
--vhx --8x2 (two words)
```

```
--vhx--8x2 (one word)
```

- The end of a line is treated as whitespace, for example:

```
--vhx
--8x2
```

This is equivalent to:

```
--vhx --8x2
```

- Strings enclosed in quotation marks ("), or apostrophes (') are treated as a single word. Within a quoted word, an apostrophe is treated as an ordinary character. Within an apostrophe delimited word, a quotation mark is treated as an ordinary character.

Use quotation marks to delimit filenames or path names that contain spaces, for example:

```
--output C:\\My Project\\output.txt (three words)
```

```
--output "C:\\My Project\\output.txt " (two words)
```

Use apostrophes to delimit words that contain quotes, for example:

```
-DNAME='"Arm Compiler"' (one word)
```

- Characters enclosed in parentheses are treated as a single word, for example:

```
--option(x, y, z) (one word)
```

```
--option (x, y, z) (two words)
```

- Within quoted or apostrophe delimited strings, you can use a backslash (\\) character to escape the quote, apostrophe, and backslash characters.
- A word that occurs immediately next to a delimited word is treated as a single word, for example:

```
--output "C:\\Project\\output.txt "
```

This is treated as the single word:

```
--outputC:\\Project\\output.txt
```

- Lines beginning with a semicolon (;) or a hash (#) character as the first nonwhitespace character are comment lines. A semicolon or hash character that appears anywhere else in a line is not treated as the start of a comment, for example:

```
-o objectname.axf      ;this is not a comment
```

A comment ends at the end of a line, or at the end of the file. There are no multi-line comments, and there are no part-line comments.

Related information

[Overview of via files](#) on page 84

[--via=file](#) on page 79