ARM® Compiler v5.04 for µVision

Version 5

fromelf User Guide

ARM DUI0459D

ARM® Compiler v5.04 for µVision

fromelf User Guide

Release Information

Document History

Issue	Date	Confidentiality	Change
A	December 2008	Non-Confidential	Release for RVCT v4.0 for µVision
B	June 2011	Non-Confidential	Release for ARM Compiler v4.1 for µVision
C	July 2012	Non-Confidential	Release for ARM Compiler v5.02 for µVision
D	30 May 2014	Non-Confidential	Release for ARM Compiler v5.04 for µVision

Non-Confidential Proprietary Notice

ARM Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

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.

Web Address

www.arm.com

Contents

ARM® Compiler v5.04 for µVision fromelf User Guide

List of Tables
ARM® Compiler v5.04 for µVision fromelf User Guide

Preface

This preface introduces the *ARM® Compiler v5.04 for µVision fromelf User Guide*.

It contains the following:

About this book

ARM Compiler for µVision fromelf User Guide. This manual provides information on how to use the fromelf utility. Available as a PDF.

Using this book

This book is organized into the following chapters:

Chapter 1 Overview of the fromelf Image Converter

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

Chapter 2 Using fromelf

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

Chapter 3 fromelf Command-line Options

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

Chapter 4 Via File Syntax

Describes the syntax of via files accepted by the `fromelf`.

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.

Typographic conventions

italic
: Introduces special terminology, denotes cross-references, and citations.

bold
: Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.

`monospace`
: Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.

`mono`space
: Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.

`monospace italic`
: Denotes arguments to monospace text where the argument is to be replaced by a specific value.

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

<and>
: Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example:

```
MRC p15, 0 <Rd>, <CRn>, <CRm>, <Opcode_2>
```

SMALL CAPITALS
: Used in body text for a few terms that have specific technical meanings, that are defined in the *ARM glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Feedback

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

If you have comments on content then send an e-mail to *errata@arm.com*. Give:

- The title.
- The number ARM DUI0459D.
- The page number(s) to which your comments refer.
- A concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

Other information

- *ARM Information Center*.
- *ARM Technical Support Knowledge Articles*.
- *Support and Maintenance*.
- *ARM Glossary*.

Chapter 1
Overview of the fromelf Image Converter

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

It contains the following sections:

1.1 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 produced by the compiler, assembler, and linker.
- Process all ELF files in an archive produced by `armar`, and output the processed files into another archive if required.
- Convert ELF images into other formats that can be used by ROM tools or directly loaded into memory. The formats available are:
  - Plain binary.
  - Motorola 32-bit S-record.
  - Intel Hex-32.
  - 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.

Note

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

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

Related references

1.2 fromelf execution modes on page 1-12.
1.4 fromelf command-line syntax on page 1-14.
3 fromelf Command-line Options on page 3-21.

1.2 fromelf execution modes

You can run `fromelf` in various execution modes.

The execution modes are:

- Text mode (`--text`, and others), to output information about an object or image file.
- Format conversion mode (`--bin`, `--m32`, `--i32`, `--vhx`).

Related references

3.2 --bin on page 3-24.
--elf fromelf option.
3.27 --i32 on page 3-55.
3.35 --m32 on page 3-64.
3.45 --text on page 3-74.
3.47 --vhx on page 3-77.

1.3 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.

Examples

To display the help information, enter:

```
fromelf --help
```

Related references

1.4 fromelf command-line syntax on page 1-14.
3.26 --help on page 3-54.

1.4 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 references

3 fromelf Command-line Options on page 3-21.

3.32 input_file on page 3-60.

Chapter 2
Using fromelf

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

It contains the following sections:

2.1 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 references

3.1 --base [[object_file::]load_region_ID=]num on page 3-23.
3.32 input_file on page 3-60.

2.2 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

The following examples show how to process ELF files in an archive, `test.a`, that contains:

```
bmw.o
bmw1.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/
```

Related references

3.32 input_file on page 3-60.
3.38 --output=destination on page 3-67.

2.3 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.

Note

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 references

1.4 fromelf command-line syntax on page 1-14.
3.20 --emit=option[,option,...] on page 3-45.
3.45 --text on page 3-74.

2.4 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.

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, for example:

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

Examples

Do the following:

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

```
long long altstack[10] __attribute__ ((section ("STACK"), zero_init));
int main()
{
    return sizeof(altstack);
}
```

2. Compile the source:

```
-c s.c -o s.o
```

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

```
armlink --keep=s.o(STACK) 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 `STACK` and `altstack` symbols in the `fromelf` output, for example:

```
...
** Section #9
    Name        : .symtab
    Type        : SHT_SYMTAB (0x00000002)
    Flags       : None (0x00000000)
    Addr        : 0x00000000
    File Offset : 2792 (0xae8)
    Size        : 2896 bytes (0xb50)
    Link        : Section 10 (.strtab)
    Info        : Last local symbol no = 115
    Alignment   : 4
    Entry Size  : 16    Symbol table .symtab (180 symbols, 115 local)
      #  Symbol Name                Value      Bind  Sec  Type  Vis  Size
=============================================================================
...
     16  STACK                      0x00008228   Lc    2  Sect  De   0x50
...
    179  altstack                   0x00008228   Gb    2  Data  Hi   0x50
...
```

The `Sec` column shows the section where the stack is placed. In this example, section 2.

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

```
...
==================================
** Section #2
    Name        : ER_ZI
    Type        : SHT_NOBITS (0x00000008)
    Flags       : SHF_ALLOC + SHF_WRITE (0x00000003)
    Addr        : 0x000081c8
    File Offset : 508 (0x1fc)
    Size        : 176 bytes (0xb0)
    Link        : SHN_UNDEF
    Info        : 0
    Alignment   : 8
    Entry Size  : 0
```

```
====================================
...
```

This shows that the symbols are placed in a ZI execution region.

Related references

3.45 --text on page 3-74.

Chapter 3
fromelf Command-line Options

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

It contains the following sections:

3.1 --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.

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 3-1 Examples of using --base

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

Related concepts

2.1 General considerations when using fromelf on page 2-16.

Related references

3.27 --i32 on page 3-55.
3.28 --i32combined on page 3-56.
3.35 --m32 on page 3-64.
3.36 --m32combined on page 3-65.

3.2 --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.

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 a output file.

Examples

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

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

Related references

3.38 --output=destination on page 3-67.
3.51 --widthxbanks on page 3-81.

3.3 --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 `0x2FFFFF00` 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 references

3.4 --bincombined_base=address on page 3-26.
3.5 --bincombined_padding=size,num on page 3-27.
3.38 --output=destination on page 3-67.
3.51 --widthxbanks on page 3-81.

Related information

Input sections, output sections, regions, and Program Segments.

3.4 --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.

Syntax

`--bincombined_base=`*`address`*

Where *`address`* 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.

Default

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

Restrictions

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

Examples

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

Related references

3.3 --bincombined on page 3-25.
3.5 --bincombined_padding=size,num on page 3-27.

Related information

Input sections, output sections, regions, and Program Segments.

3.5 --bincombined_padding=size,num

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

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.

Note

`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.

Default

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

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 references

3.3 --bincombined on page 3-25.
3.4 --bincombined_base=address on page 3-26.

3.6 --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 a output file.

Examples

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,0x18,0x00,0x00,0x00,0x00,0x30,0xB0,0xE3,0x00,0x40,0xB0,0xE3,0x00,0x50,0xB0,0xE3,
    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,0x00,0x00,0x0C,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
};
```

- 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 references

3.7 --cadcombined on page 3-30.
3.38 --output=destination on page 3-67.

Related information

Input sections, output sections, regions, and Program Segments.

3.7 --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.

Examples

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

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

Related references

3.6 --cad on page 3-28.
3.38 --output=destination on page 3-67.

3.8 --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 references

3.29 --ignore_section=option[,option,...] on page 3-57.
3.30 --ignore_symbol=option[,option,...] on page 3-58.
3.40 --relax_section=option[,option,...] on page 3-69.
3.41 --relax_symbol=option[,option,...] on page 3-70.

3.9 --continue_on_error

Reports any errors and then continues.

Usage

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

Related references

3.18 --diag_warning=tag[,tag,...] on page 3-43.

3.10 --cpu=list

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

Syntax

```
--cpu=list
```

Related references

3.11 --cpu=name on page 3-34.

3.11 --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 or architecture interprets it.

Syntax

`--cpu=`*name*

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

- If *name* is the name of a processor, enter it as shown on ARM data sheets, for example, `ARM7TDMI`, `ARM1176JZ-S`, `MPCore`.
- If *name* is the name of an architecture, it must belong to the list of architectures shown in the following table.

Processor and architecture names are not case-sensitive.

Wildcard characters are not accepted.

Table 3-2 Supported ARM architectures

Architecture	Description	Example processors
`4`	ARMv4 without Thumb	SA-1100
`4T`	ARMv4 with Thumb	ARM7TDMI, ARM9TDMI, ARM720T, ARM740T, ARM920T, ARM922T, ARM940T, SC100
`5T`	ARMv5 with Thumb and interworking	-
`5TE`	ARMv5 with Thumb, interworking, DSP multiply, and double-word instructions	ARM9E, ARM946E-S, ARM966E-S
`5TEJ`	ARMv5 with Thumb, interworking, DSP multiply, double-word instructions, and Jazelle® extensions **Note** `fromelf` cannot generate Java bytecodes.	ARM926EJ-S, ARM1026EJ-S, SC200
`6`	ARMv6 with Thumb, interworking, DSP multiply, double-word instructions, unaligned and mixed-endian support, Jazelle, and media extensions	ARM1136J-S, ARM1136JF-S
`6-M`	ARMv6 micro-controller profile with Thumb only, plus processor state instructions	Cortex-M1 without OS extensions, Cortex-M0, SC000, Cortex-M0plus
`6S-M`	ARMv6 micro-controller profile with Thumb only, plus processor state instructions and OS extensions	Cortex-M1 with OS extensions
`6K`	ARMv6 with SMP extensions	MPCore
`6T2`	ARMv6 with Thumb (Thumb-2 technology)	ARM1156T2-S, ARM1156T2F-S
`6Z`	ARMv6 with Security Extensions	ARM1176JZF-S, ARM1176JZ-S
`7`	ARMv7 with Thumb (Thumb-2 technology) only, and without hardware divide	-

Table 3-2 Supported ARM architectures (continued)

Architecture	Description	Example processors
`7-R`	ARMv7 real-time profile with ARM, Thumb (Thumb-2 technology), DSP support, and 32-bit SIMD support	Cortex-R4, Cortex-R4F, Cortex-R7
`7-M`	ARMv7 micro-controller profile with Thumb (Thumb-2 technology) only and hardware divide	Cortex-M3, SC300
`7E-M`	ARMv7-M enhanced with DSP (saturating and 32-bit SIMD) instructions	Cortex-M4

Note

- ARMv7 is not an actual ARM architecture. `--cpu=7` denotes the features that are common to the ARMv7-A, ARMv7-R, and ARMv7-M architectures. By definition, any given feature used with `--cpu=7` exists on the ARMv7-A, ARMv7-R, and ARMv7-M architectures.
- `7-A.security` is not an actual ARM architecture, but rather, refers to `7-A` plus Security Extensions.

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.
- The supported `--cpu` values include all current ARM product names or architecture versions.

  Other ARM architecture-based processors, such as the Marvell Feroceon and the Marvell XScale, are also supported.

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=5TE --disassemble` produces disassembly that can be assembled for the ARM926EJ-S® processor.

FPU

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

  Note

  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.

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.

To obtain a full list of architectures and processors, use the `--cpu=list` option.

Restrictions

You cannot specify both a processor and an architecture on the same command-line.

Examples

To select the disassembly for the Cortex™-A8 processor, use:

```
--cpu=Cortex-A8
```

Related references

3.10 --cpu=list on page 3-33.
3.31 --info=topic[,topic,...] on page 3-59.
3.45 --text on page 3-74.

3.12 --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 references

3.45 --text on page 3-74.

3.13 --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.

Note

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

Restrictions

You can use this option only in text mode.

Examples

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

```
** Section #12 '.ARM.attributes' (SHT_ARM_ATTRIBUTES)
    Size   : 69 bytes
    'aeabi' file build attributes:
    0x000000:   05 41 52 4d 37 54 44 4d 49 00 06 02 08 01 11 01    .ARM7TDMI.......
    0x000010:   12 02 14 02 17 01 18 01 19 01 1a 01 1e 03 20 02    .............. .
    0x000020:   41 52 4d 00                                        ARM.
        Tag_CPU_name = "ARM7TDMI"
        Tag_CPU_arch = ARM v4T (=2)
        Tag_ARM_ISA_use = ARM instructions were permitted to be used (=1)
        Tag_ABI_PCS_GOT_use = Data are imported directly (=1)
        Tag_ABI_PCS_wchar_t = Size of wchar_t is 2 (=2)
        Tag_ABI_FP_denormal = This code was permitted to require that the sign of a flushed-
to-zero number be preserved in the sign of 0 (=2)
        Tag_ABI_FP_number_model = This code was permitted to use only IEEE 754 format FP
numbers (=1)
        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 values occupy the smallest container big enough to hold all
values (=1)
        Tag_ABI_optimization_goals = Optimized for small size, but speed and debugging
illusion preserved (=3)
        Tag_compatibility = 2, "ARM"
    'ARM' file build attributes:
    0x000000:   04 01 12 01                                        ....
```

Related references

3.19 --dump_build_attributes on page 3-44.
3.20 --emit=option[,option,...] on page 3-45.
3.22 --extract_build_attributes on page 3-48.

Related information

Application Binary Interface for the ARM Architecture.

3.14 --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.
- `warning`, to treat all warnings as errors.

Related references

3.15 --diag_remark=tag[,tag,...] on page 3-40.
3.16 --diag_style={arm|ide|gnu} on page 3-41.
3.17 --diag_suppress=tag[,tag,...] on page 3-42.
3.18 --diag_warning=tag[,tag,...] on page 3-43.

3.15 --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.

Related references

3.14 --diag_error=tag[,tag,...] on page 3-39.
3.16 --diag_style={arm|ide|gnu} on page 3-41.
3.17 --diag_suppress=tag[,tag,...] on page 3-42.
3.18 --diag_warning=tag[,tag,...] on page 3-43.

3.16 --diag_style={arm|ide|gnu}

Specifies the display style for diagnostic messages.

Syntax

`--diag_style=`*`string`*

Where *`string`* is one of:

`arm`
: Display messages using the 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.

Default

The default is `--diag_style=arm`.

Related references

3.14 --diag_error=tag[,tag,...] on page 3-39.
3.15 --diag_remark=tag[,tag,...] on page 3-40.
3.17 --diag_suppress=tag[,tag,...] on page 3-42.
3.18 --diag_warning=tag[,tag,...] on page 3-43.

3.17 --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.
- `error`, to suppress all errors that can be downgraded.
- `warning`, to suppress all warnings.

Related references

3.14 --diag_error=tag[,tag,...] on page 3-39.
3.15 --diag_remark=tag[,tag,...] on page 3-40.
3.16 --diag_style={arm|ide|gnu} on page 3-41.
3.18 --diag_warning=tag[,tag,...] on page 3-43.

3.18 --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.
- `error`, to set all errors that can be downgraded to warnings.

Related references

3.14 --diag_error=tag[,tag,...] on page 3-39.
3.15 --diag_remark=tag[,tag,...] on page 3-40.
3.16 --diag_style={arm|ide|gnu} on page 3-41.
3.18 --diag_warning=tag[,tag,...] on page 3-43.

3.19 --dump_build_attributes

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

Restrictions

You can use this option only in text mode.

Examples

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

```
...
** Section #12 '.ARM.attributes' (SHT_ARM_ATTRIBUTES)
    Size   : 69 bytes
    0x000000:   41 33 00 00 00 61 65 61 62 69 00 01 29 00 00 00    A3...aeabi..)...
    0x000010:   05 41 52 4d 37 54 44 4d 49 00 06 02 08 01 11 01    .ARM7TDMI.......
    0x000020:   12 02 14 02 17 01 18 01 19 01 1a 01 1e 03 20 02    .............. .
    0x000030:   41 52 4d 00 11 00 00 00 41 52 4d 00 01 09 00 00    ARM.....ARM.....
    0x000040:   00 04 01 12 01                                     .....
```

Related references

3.13 --decode_build_attributes on page 3-38.
3.20 --emit=option[,option,...] on page 3-45.
3.22 --extract_build_attributes on page 3-48.
3.45 --text on page 3-74.

3.20 --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.

Restrictions

You can use this option only in text mode.

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

`got`
Prints the contents of the *Global Offset Table* (GOT) objects.

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

vfe
: Prints information about unused virtual functions.

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

Related references

3.13 --decode_build_attributes on page 3-38.
3.21 --expandarrays on page 3-47.
3.45 --text on page 3-74.

3.21 --expandarrays

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

Restrictions

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

Related references

3.45 --text on page 3-74.

3.22 --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

You can use this option only in text mode.

Examples

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

```
================================================================================
** Object/Image Build Attributes
    'aeabi' file build attributes:
    0x000000:   05 41 52 4d 37 54 44 4d 49 00 06 02 08 01 11 01    .ARM7TDMI.......
    0x000010:   12 02 14 02 17 01 18 01 19 01 1a 01 1e 03 20 02    .............. .
    0x000020:   41 52 4d 00                                        ARM.
        Tag_CPU_name = "ARM7TDMI"
        Tag_CPU_arch = ARM v4T (=2)
        Tag_ARM_ISA_use = ARM instructions were permitted to be used (=1)
        Tag_ABI_PCS_GOT_use = Data are imported directly (=1)
        Tag_ABI_PCS_wchar_t = Size of wchar_t is 2 (=2)
        Tag_ABI_FP_denormal = This code was permitted to require that the sign of a flushed-
to-zero number be preserved in the sign of 0 (=2)
        Tag_ABI_FP_number_model = This code was permitted to use only IEEE 754 format FP
numbers (=1)
        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 values occupy the smallest container big enough to hold all
values (=1)
        Tag_ABI_optimization_goals = Optimized for small size, but speed and debugging
illusion preserved (=3)
        Tag_compatibility = 2, "ARM"
    'ARM' file build attributes:
    0x000000:   04 01 12 01                                        ....
```

Related references

3.13 --decode_build_attributes on page 3-38.
3.19 --dump_build_attributes on page 3-44.
3.20 --emit=option[,option,...] on page 3-45.
3.45 --text on page 3-74.

3.23 --fieldoffsets

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

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.

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:

- Is not available if the source file does not have debug information.
- Can be used only in text mode.

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:

```
; 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 C:\Program Files\DS-5\bin\..\include
\stdio.h
|__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 references

3.39 --qualify on page 3-68.

3.42 --select=select_options on page 3-71.

3.45 --text on page 3-74.

Related information

EQU.

GET or INCLUDE.

3.24 --fpu=list

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

Deprecated options are not listed.

Related references

3.25 --fpu=name on page 3-52.

3.25 --fpu=name

Specifies the target FPU architecture.

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

Syntax

`--fpu=`*name*

Where *name* is one of:

`none`
: Selects no floating-point option. No floating-point code is to be used.

`vfpv2`
: Selects a hardware floating-point unit conforming to architecture VFPv2.

`vfpv3`
: Selects a hardware vector floating-point unit conforming to architecture VFPv3. VFPv3 is backwards compatible with VFPv2 except that VFPv3 cannot trap floating-point exceptions.

`vfpv3_fp16`
: Selects a hardware vector floating-point unit conforming to architecture VFPv3 that also provides the half-precision extensions.

`vfpv3_d16`
: Selects a hardware vector floating-point unit conforming to VFPv3-D16 architecture.

`vfpv3_d16_fp16`
: Selects a hardware vector floating-point unit conforming to VFPv3-D16 architecture, that also provides the half-precision extensions.

`vfpv4`
: Selects a hardware floating-point unit conforming to the VFPv4 architecture.

`vfpv4_d16`
: Selects a hardware floating-point unit conforming to the VFPv4-D16 architecture.

`fpv4-sp`
: Selects a hardware floating-point unit conforming to the single precision variant of the FPv4 architecture.

`softvfp`
: Selects software floating-point support where floating-point operations are performed by a floating-point library, `fplib`. This is the default if you do not specify a `--fpu` option, or if you select a CPU that does not have an FPU.

`softvfp+vfpv2`
: Selects a hardware floating-point unit conforming to VFPv2, with software floating-point linkage. Select this option if you are interworking Thumb code with ARM code on a system that implements a VFP unit.

`softvfp+vfpv3`
: Selects a hardware vector floating-point unit conforming to VFPv3, with software floating-point linkage. Select this option if you are interworking Thumb code with ARM code on a system that implements a VFPv3 unit.

`softvfp+vfpv3_fp16`
: Selects a hardware vector floating-point unit conforming to VFPv3-fp16, with software floating-point linkage.

`softvfp+vfpv3_d16`
: Selects a hardware vector floating-point unit conforming to VFPv3-D16, with software floating-point linkage.

`softvfp+vfpv3_d16_fp16`
: Selects a hardware vector floating-point unit conforming to VFPv3-D16-fp16, with software floating-point linkage.

`softvfp+vfpv4`
: Selects a hardware floating-point unit conforming to FPv4, with software floating-point linkage.

`softvfp+vfpv4_d16`
: Selects a hardware floating-point unit conforming to VFPv4-D16, with software floating-point linkage.

`softvfp+fpv4-sp`
: Selects a hardware floating-point unit conforming to FPv4-SP, with software floating-point linkage.

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. For example, the options `--disassemble --cpu=ARM1136JF-S --fpu=softvfp` disassembles code that uses the software floating-point library `fplib`, even though the choice of CPU implies the use of architecture VFPv2.

Restrictions

NEON support is disabled for `softvfp`.

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. For example, the option `--cpu ARM1136JF-S` implies the option `--fpu vfpv2`. If a VFP coprocessor is present, VFP instructions are generated.

Related references

3.24 --fpu=list on page 3-51.
3.31 --info=topic[,topic,...] on page 3-59.
3.45 --text on page 3-74.

3.26 --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 references

3.43 --show_cmdline on page 3-72.
3.46 --version_number on page 3-76.
3.49 --vsn on page 3-79.

3.27 --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:

- 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.

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 a output file.

Examples

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 references

3.1 --base [[object_file::]load_region_ID=]num on page 3-23.
3.28 --i32combined on page 3-56.
3.38 --output=destination on page 3-67.

3.28 --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:

- 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.

Examples

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 references

3.1 --base [[object_file::]load_region_ID=]num on page 3-23.
3.27 --i32 on page 3-55.
3.38 --output=destination on page 3-67.

3.29 --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 references

3.8 --compare=option[,option,...] on page 3-31.
3.30 --ignore_symbol=option[,option,...] on page 3-58.
3.40 --relax_section=option[,option,...] on page 3-69.

3.30 --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 a symbols 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 references

3.8 --compare=option[,option,...] on page 3-31.
3.29 --ignore_section=option[,option,...] on page 3-57.
3.41 --relax_symbol=option[,option,...] on page 3-70.

3.31 --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 ARM and Thumb instructions defined in the code sections of each input file.

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 ARM or Thumb 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 ARM or Thumb 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.

Note

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.

Note

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 references

3.45 --text on page 3-74.

3.32 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.
- 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*)'
```

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

Examples

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

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

Related concepts

2.2 Examples of processing ELF files in an archive on page 2-17.

Related references

3.2 --bin on page 3-24.
3.6 --cad on page 3-28.

3.7 --cadcombined on page 3-30.
3.8 --compare=option[,option,...] on page 3-31.
3.27 --i32 on page 3-55.
3.28 --i32combined on page 3-56.
--in_place fromelf option.
3.35 --m32 on page 3-64.
3.36 --m32combined on page 3-65.
3.38 --output=destination on page 3-67.
3.45 --text on page 3-74.
3.47 --vhx on page 3-77.

3.33 --interleave=option

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

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` or `--text -c`.

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

Default

The default is `--interleave=none`.

Related references

3.20 --emit=option[,option,...] on page 3-45.
3.44 --source_directory=path on page 3-73.
3.45 --text on page 3-74.

3.34 --licretry

If you are using floating licenses, this option makes up to 10 attempts to obtain a license when you invoke `fromelf`.

Usage

Use this option if your builds are failing to obtain a license from your license server, and only after you have ruled out any other problems with the network or the license server setup.

It is recommended that you place this option in the `ARMCC5_FROMELFOPT` environment variable. In this way, you do not have to modify your build files.

Related information

Toolchain environment variables.

ARM DS-5 License Management Guide.

3.35 --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:

- 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 a output file.

Examples

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 references

3.1 --base [[object_file::]load_region_ID=]num on page 3-23.
3.36 --m32combined on page 3-65.
3.38 --output=destination on page 3-67.

3.36 --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:

- 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.

Examples

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 references

3.1 --base [[object_file::]load_region_ID=]num on page 3-23.
3.35 --m32 on page 3-64.
3.38 --output=destination on page 3-67.

3.37 --only=section_name

Forces the output to display only the named section.

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`, enter:

  ```
  fromelf --only=.symtab --text -s test.axf
  ```
- To display all `ER`*n* 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 references

3.45 --text on page 3-74.

3.38 --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`:

- You can specify a single input file and a single output filename.
- 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 --bin mylib.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 references

3.2 --bin on page 3-24.
3.45 --text on page 3-74.

3.39 --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 --fieldoffsets`, the linker might define the following symbols:

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

Using `fromelf --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 references

3.23 --fieldoffsets on page 3-49.

3.40 --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 *`symbol_name`* and *`object_name`* arguments
- Specify multiple values in one *`option`* followed by a comma-separated list of arguments.

Related references

3.8 --compare=option[,option,...] on page 3-31.
3.29 --ignore_section=option[,option,...] on page 3-57.
3.41 --relax_symbol=option[,option,...] on page 3-70.

3.41 --relax_symbol=option[,option,…]

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

Restrictions

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

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``::`*`section_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.

Related references

3.8 --compare=option[,option,...] on page 3-31.
3.30 --ignore_symbol=option[,option,...] on page 3-58.
3.40 --relax_section=option[,option,...] on page 3-69.

3.42 --select=select_options

When used with `--fieldoffsets` or `--text -a` options, selects 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`.

Related references

3.23 --fieldoffsets on page 3-49.
3.45 --text on page 3-74.

3.43 --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 references

3.48 --via=file on page 3-78.

3.44 --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 references

3.33 --interleave=option on page 3-62.

3.45 --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 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.

-c

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

——— **Note** ———

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.

-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`.

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 `inputfile.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 `inputfile.axf` but none of the structure addresses, enter:

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

Related tasks

2.4 Using fromelf to find where a symbol is placed in an executable ELF image on page 2-19.

Related references

3.11 --cpu=name on page 3-34.
3.20 --emit=option[,option,...] on page 3-45.
3.21 --expandarrays on page 3-47.
3.31 --info=topic[,topic,...] on page 3-59.
3.33 --interleave=option on page 3-62.
3.37 --only=section_name on page 3-66.
3.38 --output=destination on page 3-67.
3.42 --select=select_options on page 3-71.
3.50 -w on page 3-80.

Related information

Linker options for getting information about images.

3.46 --version_number

Displays the version of `fromelf` you are using.

Usage

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

- `nnn` is the version number.
- `bbbb` is the build number.

Examples

Version 5.01 build 0019 is displayed as `5010019`.

Related references

3.26 --help on page 3-54.
3.49 --vsn on page 3-79.

3.47 --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.

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 a 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 references

3.38 --output=destination on page 3-67.
3.51 --widthxbanks on page 3-81.

3.48 --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 references

4 Via File Syntax on page 4-83.

3.49 --vsn

Displays the version information and the license details.

Examples

Example output:

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

Related references

3.26 --help on page 3-54.
3.46 --version_number on page 3-76.

3.50 -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.

For 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 references

3.45 --text on page 3-74.

3.51 --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 --vhx --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 --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 --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 references

3.2 --bin on page 3-24.
3.38 --output=destination on page 3-67.
3.47 --vhx on page 3-77.

Chapter 4
Via File Syntax

Describes the syntax of via files accepted by the `fromelf`.

It contains the following sections:

4.1 Overview of via files

Via files are plain text files that allow you to specify ELF file converter command-line arguments and options.

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.

Note

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

Via file evaluation

When the ELF file converter is invoked it:

1. Replaces the first specified `--via` *`via_file`* argument with the sequence of argument words 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 you specify them, and each via file is processed completely including processing nested via files before processing the next via file.

Related references

4.2 Via file syntax rules on page 4-85.
3.48 --via=file on page 3-78.

4.2 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:

  `--debugonly --privacy` (two words)

  `--debugonly--privacy` (one word)
- The end of a line is treated as whitespace, for example:

  ```
  --debugonly
  --privacy
  ```

  This is equivalent to:

  `--debugonly --privacy`
- 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 concepts

4.1 Overview of via files on page 4-84.

Related references

3.48 --via=file on page 3-78.