ELD User Guide

version

Qualcomm Technologies, Inc.

March 13, 2026

Contents

ELD User Guide

This document describes usage of ELD

Getting started

About the linker

Linking is the process of collecting and combining various pieces of code and data into a single file that can be loaded (copied) into memory and executed. Linking is performed automatically by programs called linkers, which enable separate compilation.

Source

eld is available in the git repository:

> **Note**
>
> git clone git://github.com/qualcomm/eld

Summary of the linker features

- **Most important features supported by GNU Linker**
    - Development going on to add additional support
- **LTO**
    - Supports ThinLTO
    - Supports Full LTO
    - Supports Both flavors of LTO with linker scripts
- **Supports Version scripts**
    - Supports extensive usage of Linker scripts
- **Better Diagnostics**
    - YAML Map Files
    - Easier to read Text Map files
- Support for Linker Plugins
- –reproduce flag to easily reproduce the linking step.

What the linker can accept as input

The following types of files can be used as input to ld.eld:

- Object files

- Static Libraries
- Shared Libraries
- Symbol definition file
- Extern list
- Binary files

What the linker outputs

- ELF File
- A partially linked ELF object that can be used as input in a subsequent link step.
- **Map file**
    - YAML Map
    - Text Map
- **Tar file**
    - when using –reproduce flag

Supported Linking Modes

A linking model is a group of command-line options and memory maps that control the behavior of the linker. The linking models supported by eld are:

Static Linking

In this mode, all the symbol definitions/functions are copied in the final executable.

Dynamic Linking

In Dynamic linking is accomplished by placing the name of a sharable library in the executable image. Actual linking with the library routines does not occur until the image is run, when both the executable and the library are placed in memory. An advantage of dynamic linking is that multiple programs can share a single copy of the library.

Partial Linking

This model produces a relocatable ELF object suitable for input to the linker in a subsequent link step. The partial object can be used as input to another link step. The linker performs limited processing of input objects to produce a single output object

PIE

This model produces a Position Independent Executable (PIE). This is an executable that does not need to be executed at a specific address but can be executed at any suitably aligned address. All objects and libraries linked into the image must be compiled to be position independent

Shared Libraries

A shared library is an object module that, at run time, can be loaded at an arbitrary memory address and linked with a program in memory. The process is known as dynamic linking and is performed by a program called a dynamic linker. Shared libraries are also referred to as shared objects (.so).

Shared libraries are “shared” in two different ways:

there is exactly one .so file for a particular library. The code and data in this .so file are shared by all of the exe obj files that reference the library. a single copy of the .text section of a shared library in memory can be shared by different running processes.

Supported Targets

Below are the currently supported targets by the ELD

- Hexagon
- RISC-V
- ARM
- AARCH64

Linker Script

Overview

Linker scripts define how input sections (like .text, .data, .bss) are mapped to output sections and where they are placed in memory. These scripts control:

- Memory regions (e.g., RAM, ROM)
- Section alignment
- Ordering of code and data
- Load vs. execution addresses

This is the standard method used by most linkers like GNU ld.

Linker scripts provide detailed specifications of how files are to be linked. They offer greater control over linking than is available using just the linker command options.

NOTE Linker scripts are optional. In most cases, the default behavior of the linker is sufficient.

Linker scripts control the following properties:

- ELF program header

- Program entry point
- Input and output files and searching paths
- Section memory placement and runtime
- Section removal
- Symbol definition

A linker script consists of a sequence of commands stored in a text file.

The script file can be specified on the command line either with -T, or by specifying the file as an input files.

The linker distinguishes between script files and object files and handles each accordingly.

To generate a map file that shows how a linker script controlled linking, use the M option.

Command	Description
PHDRS	Program Headers
SECTIONS	Section mapping and memory placementELF program header definition
MEMORY	Define memory regions used by `>REGION` and `AT>REGION` placement
ENTRY	ELF program headerProgram execution entry point
OUTPUT_FORMAT	Parsed, but no effect on linking
OUTPUT_ARCH	Parsed, but no effect on linking
SEARCH_DIR	Add additional searching directory for libraries
INCLUDE	Include linker script file
OUTPUT	Define output filename
GROUP	Define files that will be searched repeatedly
ASSERT	Linker script assertion
NOCROSSREFS	Check cross references among a group of sections
OVERLAY	GNU ld compatible OVERLAY block (parsing support)

Basic script syntax

Symbols

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

Comments

Comments can appear in linker scripts.

Strings

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

Expression basics

Expressions are similar to C, and support all C arithmetic operators. They are evaluated as type `long` or `unsigned long`.

Location counter basics

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

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

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

Symbol assignment basics

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

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

Assignment statements are similar to C, and support all C assignment operators. Terminate assignment statements with a semicolon.

Script commands

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

PHDRS

The `PHDRS` (Program Headers) command in a linker script is used to define program headers in the output binary, particularly for ELF Executable and Linkable Format (ELF) files. These headers are essential for the runtime loader to understand how to load and map the binary into memory.

When and why PHDRS is used?

- Custom memory mapping

  You use `PHDRS` when you want to explicitly control how sections are grouped into segments in the ELF file. This is especially important for:

  - Embedded systems
  - Custom bootloaders
  - OS kernels
- Fine-grained segment control
  - Assign specific sections to specific segments.
  - Control segment flags (for example, `PT_LOAD`, `PT_NOTE`, `PT_TLS`).
  - Set permissions (`r`, `w`, `x`) for each segment.

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

The `PHDRS` script command sets information in the program headers, also known as the *segment header* of an ELF output file.

- `name` – Specifies the program header in the `SECTIONS` command.
- `type` – Specifies the program header type.
- `PT_LOAD` – Loadable segment.
- `PT_NULL` – Linker does not include section in a segment. No loadable section should be set to `PT_NULL`.
- `PT_DYNAMIC` – Segment where dynamic linking information is stored.
- `PT_INTERP` – Segment where the name of the dynamic linker is stored.
- `PT_NOTE` – Segment where note information is stored.
- `PT_SHLIB` – Reserved program header type.

- `PT_PHDR` – Segment where program headers are stored.
- `FLAGS` – Specifies the `p_flags` field in the program header. The value of flags must be an integer. It is used to set the `p_flags` field of the program header; for instance, `FLAGS(5)` sets `p_flags` to `PF_R | PF_X` and `FLAGS(0x03000000)` sets OS-specific flags.

> **Note**
>
> If the sections in an output file have different flag settings than what is specified in `PHDRS`, the linker combines the two different flags using bitwise OR.

MEMORY

The `MEMORY` command defines named memory regions (for example `FLASH` and `RAM`) that can be used to place output sections without hard-coding absolute addresses.

In ELD, memory regions can be used in three related ways:

1. **VMA placement**: `>REGION` assigns the output section's VMA to the next available address in `REGION` (starting from `ORIGIN` and advancing by the output section size).
2. **LMA placement**: `AT>REGION` assigns the output section's LMA to the next available address in `REGION` (tracked independently from VMA placement).
3. **Automatic region selection**: if an allocatable output section does not specify `>REGION`, ELD can select a region by matching the section's flags against region attributes (for example, place writable sections in a `(w)` region and code in an `(x)` region).

Syntax

```
MEMORY {
  <name> [(<attrs>)] : ORIGIN = <expr> , LENGTH = <expr>
  <name> [(<attrs>)] : org = <expr> , len = <expr>
  ...
}
```

Notes:

- `ORIGIN` can also be spelled as `org` or `o`.
- `LENGTH` can also be spelled as `len` or `l`.
- `ORIGIN` and `LENGTH` must be separate tokens (for example, `ORIGIN = 0x1000` is accepted, but `ORIGIN= 0x1000` is rejected).
- The expressions are normal linker-script expressions; you can use constants and arithmetic, and size suffixes like `K`, `M`, and `G` (for example, `LENGTH = 4K`).
- The `MEMORY` block can contain `INCLUDE`/`INCLUDE_OPTIONAL` directives.

Memory region names

Region names are identifiers (and may also appear quoted in scripts). Region names are referenced in output section descriptions via `>REGION` and `AT>REGION`.

If a section references an unknown region name, ELD errors out (for example, `Cannot find memory region <name>`).

Memory region attributes

`(<attrs>)` is optional and is primarily used for **automatic region selection** (when an allocatable output section does not specify an explicit `>REGION`).

ELD supports the following attribute letters:

- `w`: writable (matches `SHF_WRITE` sections).
- `x`: executable (matches `SHF_EXECINSTR` sections).
- `a`: allocatable (matches `SHF_ALLOC` sections).
- `r`: read-only (matches *non-writable* allocatable sections; ELF sections do not explicitly carry a read flag).
- `i` / `l`: initialized (matches `SHT_PROGBITS` sections; used to avoid matching `SHT_NOBITS` when desired).
- `!`: negation operator, used to say that attributes must **not** be present. For example, `(r!x)` matches non-writable, non-executable sections.

The attribute matching rules apply only to allocatable output sections that do not already have an explicit `>REGION`.

Important

If you explicitly assign a section to a region using `>REGION`, ELD does not reject the assignment even if the section's flags do not match the region's attributes. Attributes are for auto-selection, not enforcement.

Automatic region selection (orphans and missing `>REGION`)

When a linker script uses `MEMORY` and an allocatable output section is not explicitly assigned to a region, ELD attempts to assign it to the first memory region (in `MEMORY` order) whose attributes match the section's flags.

This includes *orphan output sections* (output sections created implicitly because some input sections were not matched by any rule in `SECTIONS`).

If no memory region matches an allocatable output section, ELD errors out (for example, `Error: No memory region assigned to section .data`).

Note

This behavior can differ from GNU ld and LLD for orphans. For example, some linkers may keep orphan sections in the "current" region/cursor even when region attributes suggest a different region. ELD follows the region attribute matching rules for orphans and other unassigned allocatable output sections. See https://github.com/qualcomm/eld/issues/127 for a concrete example and background.

Common recommendation: if the placement of a section matters (for example, `.rodata`, `.eh_frame`, `.tbss`), add an explicit output section rule and assign it to the intended region rather than relying on orphan heuristics.

Interaction with explicit addresses (VMA overrides)

If an output section has an explicit VMA (for example `.text 0x2000 : { … }` or via `–section-start`), the explicit VMA is honored.

If that output section also has `>REGION`:

- If the explicit VMA lies **within** the region's `[ORIGIN, ORIGIN+LENGTH]` range, the section is still accounted against the region for memory usage checks/reporting.
- If the explicit VMA lies **outside** the region range, the section is not charged against that region (the explicit address is treated as an override).

This is intended to support scripts that use regions as a default placement mechanism while still allowing some sections to be placed at fixed addresses.

TLS and `.tbss`

ELD treats TLS `SHT_NOBITS` sections (for example `.tbss`) specially for `MEMORY` region cursors: `.tbss` does not advance the region cursor, so a subsequent non-TLS output section may end up with the same VMA. If your script requires non-overlapping VMAs for TLS and non-TLS sections, place TLS output sections explicitly (for example, in a dedicated region or at a dedicated address range) rather than relying on region cursors.

LMA regions and AT>REGION

Use AT>REGION on an output section description to place the section's **load memory address (LMA)** in a region, independent of its VMA placement:

```
.text : { *(.text*) } >RAM AT>FLASH
```

Rules:

- AT(<address>) and AT>REGION are mutually exclusive; specifying both is an error.
- If a section has a VMA region (>REGION) but no explicit LMA control, ELD defaults the LMA region to the same region as the VMA region.
- If a section has explicit LMA control (AT(<address>)), ELD does not automatically align the LMA. (See Controlling Physical addresses.)

Builtins: ORIGIN() and LENGTH()

ELD supports GNU-compatible ORIGIN(<region>) and LENGTH(<region>) builtins in linker script expressions.

These functions accept a memory region name or a region alias and evaluate to the region's origin and length, respectively. If the region name cannot be resolved, ELD errors out.

REGION_ALIAS

REGION_ALIAS("alias", "region") defines an alias name for an existing memory region. Aliases can be used anywhere a region name is expected (for example, >alias or ORIGIN(alias)).

ELD restrictions and diagnostics:

- Creating two aliases with the same alias name is an error.
- An alias name must not collide with a real memory region name.

Memory usage checking and reporting

When MEMORY is used, ELD tracks memory usage per region during layout:

- If the total size placed in a region exceeds LENGTH, ELD errors out and reports the overflow amount per output section.
- With -Wlinker-script-memory, ELD can warn on zero-length regions and on regions that end up with zero used size after layout.

Additionally:

- The text map format (-MapStyle txt) includes a # MEMORY block that prints the parsed MEMORY regions and annotates each output section line with Memory : [<VMARegion>, <LMARegion>] when available.
- –print-memory-usage prints a summary table of used size and percentage per memory region when a MEMORY directive is present.

PHDRS interaction and program header loading

MEMORY controls **where sections go** (VMA via >REGION and LMA via AT>REGION). PHDRS controls **which segments exist** and which output sections are assigned to those segments (via :phdr on output section descriptions).

Key points:

- If you do not specify PHDRS, ELD creates segments using its default heuristics. A change in memory region (for example, >FLASH to >RAM) can force ELD to start a new PT_LOAD segment.
- If you specify PHDRS, ELD only creates the program headers declared in the script. In this mode, region changes do *not* create extra segments; you must assign each output section to the intended segment(s) using :phdr.

- Runtime loaders use the ELF program headers (and sometimes the file header) to decide what to map or copy. If your loader needs the file header and/or program headers to be part of the first loadable segment, use `FILEHDR` and `PHDRS` keywords on the corresponding `PT_LOAD` in the `PHDRS` command.

Example (first segment includes file+program headers):

```
PHDRS {
  text PT_LOAD FILEHDR PHDRS;
  data PT_LOAD;
}
```

If you include headers in a loadable segment, you typically must also reserve space for them in the image layout (see Using SIZEOF_HEADERS with MEMORY).

Using SIZEOF_HEADERS with MEMORY

`SIZEOF_HEADERS` evaluates to the total size (in bytes) of the ELF file header and program header table that ELD emits for the output.

Nuance: in ELD, using `SIZEOF_HEADERS` *before the first output section is seen* also acts as a signal that the script is intentionally accounting for headers; this impacts whether ELD considers the headers as occupying space at the start of the image when a linker script is present.

Common MEMORY pitfalls and errors

- **Tokenization**: `ORIGIN/LENGTH` must be separate tokens (`ORIGIN = …`); scripts like `ORIGIN= 0x1000` are rejected.
- **Unknown region name**: `>REGION` or `AT>REGION` refers to a region not defined in `MEMORY` (error: `Cannot find memory region …`).
- **No region assigned**: if `MEMORY` is present and an allocatable output section does not have an explicit region and does not match any region attributes (error: `No memory region assigned to section …`).
- **Overflow**: placed size exceeds `LENGTH` (error: `Memory region <name> exceeded limit … overflowed by …`). Remember to account for alignment, page alignment, and headers if `FILEHDR PHDRS` are used.
- **Mixing ‘‘AT(address)‘‘ with ‘‘AT>REGION‘‘**: these are mutually exclusive. If you need a fixed LMA base but still want region-based placement for the rest of the image, prefer shifting the region origin using expressions (see Using SIZEOF_HEADERS with MEMORY) or keep all LMAs explicit.
- **Orphan placement surprises**: orphan output sections and other unassigned allocatable sections are auto-assigned using region attributes; if a specific section must be in a particular region, add an explicit output section rule.

SECTIONS

Syntax :- `SECTIONS { section_statement section_statement … }`

The `SECTIONS` script command specifies how input sections are mapped to output sections, and where output sections are located in memory. The `SECTIONS` command must be specified once, and only once, in a linker script.

Section statements

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

- An `ENTRY` command.
- A symbol assignment statement to set the location counter. The location counter specifies the default address in subsequent section-mapping statements that do not explicitly specify an address.
- An output section description to specify one or more input sections in one or more library files, and map those sections to an output section. The virtual memory address of the output section can be specified using attribute keywords.

ENTRY

Syntax :- `ENTRY(symbol)`

- The `ENTRY` script command specifies the program execution entry point.
- The entry point is the first instruction that is executed after a program is loaded.
- This command is equivalent to the linker command-line option **`-e`**.

OUTPUT_FORMAT

Syntax :- `OUTPUT_FORMAT(string)`

- The `OUTPUT_FORMAT` script command specifies the output file properties.
- For compatibility with the GNU linker, this command is parsed but has no effect on linking.

OUTPUT_ARCH

Syntax :- `OUTPUT_ARCH("aarch64")`

- The `OUTPUT_ARCH` script command specifies the target processor architecture.
- For compatibility with the GNU linker, this command is parsed but has no effect on linking.

SEARCH_DIR

Syntax :- `SEARCH_DIR(path)`

- The `SEARCH_DIR` script command adds the specified path to the list of paths that the linker uses to search for libraries.
- This command is equivalent to the linker command-line option **`-L`**.

INCLUDE

Syntax :- `INCLUDE(file)`

- The `INCLUDE` script command specifies the contents of the text file at the current location in the linker script.
- The specified file is searched for in the current directory and any directory that the linker uses to search for libraries.

Note

Include files can be nested.

OUTPUT

Syntax :- `OUTPUT(file)`

- The `OUTPUT` script command defines the location and file where the linker will write output data.
- Only one output is allowed per linking.

GROUP

Syntax :- `GROUP(file, file, …)`

- The `GROUP` script command includes a list of archive file names.
- The archive names defined in the list are searched repeatedly until all defined references are resolved.

ASSERT

Syntax :- `ASSERT(expression, string)`

- The ASSERT script command adds an assertion to the linker script.

Expressions

Expressions in linker scripts are identical to C expressions

Expression helper functions

Function	Description
`.`	Return the location counter value representing the current virtual address.
`ABSOLUTE(expression)`	Return the absolute value of the expression.
`ADDR(string)`	Return the virtual address of the symbol or section. `.` is supported.
`ALIGN(expression)`	Return the value of the location counter when aligned to the next expression boundary.
`ALIGN(expression1, expression2)`	Return `expression1` rounded up to the next `expression2` boundary.
`ALIGNOF(string)`	Return the alignment of the symbol or section. `NEXT_SECTION` returns the next allocated output section alignment (only supported with `ALIGNOF`/`SIZEOF`).
`ASSERT(expression, string)`	Throw an assertion if the expression evaluates to zero.
`BLOCK(expression)`	Synonym for `ALIGN(expression)`.
`DATA_SEGMENT_ALIGN(maxpagesize, commonpagesize)`	Equivalent to `ALIGN(maxpagesize) + (. & (maxpagesize - 1))` or `ALIGN(maxpagesize) + (. & (maxpagesize - commonpagesize))` (the larger of the two).
`DATA_SEGMENT_END(expression)`	Return the value of the expression.
`DATA_SEGMENT_RELRO_END(expression)`	Return the value of the expression.
`DEFINED(symbol)`	Return 1 if the symbol is defined in the global symbol table.
`LOADADDR(string)`	Synonym for `ADDR`.
`MAX(expression1, expression2)`	Return the maximum of two expressions.
`MIN(expression1, expression2)`	Return the minimum of two expressions.
`SEGMENT_START(string, expression)`	If the string matches a known segment, return its start address; otherwise return the expression.
`SIZEOF(string)`	Return the size of the symbol, section, or segment. `NEXT_SECTION` returns the size of the next allocated output section (only with `ALIGNOF`/`SIZEOF`).
`SIZEOF_HEADERS`	Return the total size (bytes) of the ELF file header plus program headers.
`CONSTANT(MAXPAGESIZE)`	Return the ABI-defined maximum page size.
`CONSTANT(COMMONPAGESIZE)`	Return the ABI-defined common page size.

Symbol assignments

Linker scripts can define symbols that are used by the link to create the output image. Linker script symbols can be referenced by the program source code. One typical use of linker script symbols is to define the size and start / stop address of an output section.

Linker script symbol assignments support most C operators and follow C-like rules. For example, all the assignments must end with a semi-colon, and the C arithmetic operators are supported. The below mathematical operators are supported in linker script assignments:

```
u = v + w;
u = v - w;
u = v * w;
u = v / w;
u = v & w;
u = v | w;
u = v << w;
u = v >> w;
```

Additionally, linker script also supports compound assignment operators:

```
u += v;
u -= v;
u *= v;
u /= v;
u &= v;
u |= v;
u <<= v;
u >>= v;
```

The left-hand side symbol must already be defined when using compound assignment operator.

Symbol assignment types

Linker script symbol assignments are of 4 key types:

- `symbol = expression;`

  Defines a `GLOBAL` symbol.

- `HIDDEN(symbol = expression);`

  Defines a `GLOBAL` symbol with `HIDDEN` visibility.

- `PROVIDE(symbol = expression);`

  Defines the `symbol` only if it is required. If defined, the symbol will have `GLOBAL` symbol binding.

- `PROVIDE_HIDDEN(symbol = expression);`

  Defines the `symbol` only if it is required. If defined, the symbol will be a `GLOBAL` symbol with `HIDDEN` visibility.

Section of linker script symbols

Linker script symbols defined outside an output section directive are called *absolute* symbols. That is, they do not belong to any output section. The section index of such symbols is a sentinel value `SHN_ABS`.

Linker script symbols that are defined within an output section directive have the output section index as their section index.

```
// script.t
u = 0x100;  // ABS symbol
SECTIONS {
  v = 0x300; // ABS symbol
  foo : {
    *(.text.foo)
```

```
    w = 0x500;  // Section of the symbol is foo
  }
}
e = 0x700; // ABS symbol
```

Location counter

`.` symbol is a special linker symbol that always contains the current output location address. Assigning to it changes the current output location address and can be used to create holes in the output image. This special dot symbol is called the location counter. It may also be referred to as the dot counter and dot symbol.

Assignment evaluation order

Understanding symbol assignment evaluation order is the key to understanding linker scripts and a necessary skill to debug linker script related issues. For the most part, the linker script assignments behave how you expect, but things become interesting when forward references are involved.

Basic case: No forward reference

Let's start with a basic case that does not have any forward references.

```
u1 = 0x100;  // A1
SECTIONS {
  u2 = 0x300; // A2
  foo : {
    *(.text.foo)
    u3 = 0x500;  // A3
  }
  u4 = 0x700;  // A4
  bar : {
    u5 = 0x900;  // A5
    *(.text.bar)
  }
  u5 = 0x1100; // A6
}
u6 = 0x1300;  // A7
```

The linker evaluates the assignment during the layout phase. The assignments , when they do not contain any forward reference, are evaluated in the script order. Thus, in this case, the linker script assignment evaluation order is: `[A1, A2, A3, A4, A5, A6, A7]`.

Forward reference

We will now see a more complex example that has forward references. But before we dive into the example, let's understand how linker evaluates an individual assignment that has forward reference.

```
u = v + w;
```

For the above assignment, let's say that `v` is defined before this assignment as per the linker script and `w` gets defined later. In such case, the *final* value of the symbol (`w` in this case) is used to evaluate the assignment. Let's understand this with a more concrete example:

```
v = 0x100; // A1
u = v + w; // A2
w = 0x200; // A3
foo = w;   // A4
w = 0x400; // A5
v = 0x600; // A6
```

When A2 (`u = v + w`) is evaluated, `v` is already defined and is thus evaluated to its current value `0x100`. On the other hand, `w` is not defined when A2 is executed and thus the final value of `w` (i.e., 0x400) is used to evaluate A2. Thus, after all assignments are processed, the final values are:

- `v = 0x600`
- `u = 0x500`
- `foo = 0x200`
- `w = 0x400`

Now let's look at a more complex example containing forward references.

```
u = v; // A1
SECTIONS {
 v = v1; // A2
 foo : {
   *(.text.foo)
   v1 = v2; // A3
 }
 v2 = v3; // A4
}
v3 = SIZEOF(foo); // A5
```

The linker again tries to evaluate the assignment in order, however, the assignments A1, A2, A3 and A4 cannot be completely evaluated because the variables on their right hand side are not evaluated yet. The linker marks the assignments that cannot be completely evaluated as pending assignments. It also records which nodes were unevaluated in the assignment. After the layout is complete, but before the relaxations begin, the linker recursivly evaluates the pending assignments until all assignments are resolved or a circular dependency is encountered. During re-evaluation of an assignment, only the previously unevaluated nodes are reevaluated.

The assignment evaluation sequence for this example is:

1. Evaluate [A1, A2, A3, A4, A5]: PendingAssignments = [A1, A2, A3, A4], CompletedAssignments = [A5]
2. Re-evaluate [A1, A2, A3, A4]: PendingAssignments = [A1, A2, A3], CompletedAssignments = [A5, A4]
3. Re-evalaute [A1, A2, A3]: PendingAssignments = [A1, A2], CompletedAssignments = [A5, A4, A3]
4. Re-evaluate [A1, A2]: PendingAssignments = [A1], CompletedAssignments = [A5, A4, A3, A2]
5. Re-evaluate [A1]: PendingAssignments = [], CompletedAssignments = [A5, A4, A3, A2, A1]

Circular dependency

What happens if the linker script contains circular dependency among variables?

```
u = v + 0x1; // A1
v = w + 0x1; // A2
w = u + 0x1; // A3
```

What would be the values of `u`, `v`, and `w` here?

In such a case, the linker would report a warning and stop evaluating symbol assignments once a circular dependency is detected. The final values of the symbols here will be:

- `u = 0x2`
- `v = 0x2`
- `w = 0x2`

Before analyzing this behavior, it's important to note that a circular dependency represents an erroneous condition and should be considered undefined behavior. Once a layout enters an undefined state, all guarantees regarding its structure and consistency no longer apply.

With this warning in-place, let's reason how linker arrives at these final values. The assignment evaluation sequence for this example is:

1. Evaluate [A1, A2, A3]: PendingAssignments = [A1, A2, A3], CompletedAssignments = []
2. Re-evaluate [A1, A2, A3]: PendingAssignments = [A1, A2, A3], CompletedAssignments = [], Circular dependency detected, stop evaluation.

The linker stops evaluating assignments when it detects a circular dependency. The linker assigns the value `0x1‘` to the symbols in the first assignment evaluation iteration, then in the second evaluation iteration it assigns the value `0x2` to the symbols. The linker then detects circular dependency and does not evaluate assignments further, as the layout may never converge due to circular dependencies.

Forward references in dot-assignments

Forward references in dot-assignments are more complex than those in non-dot assignments because dot-assignments directly influence the layout. If a dot-assignment cannot be evaluated correctly, the layout itself cannot be computed.

The fundamental rule remains unchanged: whenever a forward reference is encountered, the final value of the symbol is used in the expression.

In eld, when a forward reference appears in a dot-assignment, the layout is computed in two passes:

- First pass: The forward reference symbol is temporarily treated as 0 during layout computation.
- Second pass: After the initial layout is complete, eld recomputes the layout using the actual final values of all forward reference symbols.

–defsym

`–defsym sym=expr` is treated akin to a linker script just with one symbol assignment. For example:

```
ld.eld -o 1.out 1.o --defsym u=0x10 --defsym v=0x30 --defsym w=0x50
```

The above link command is equivalent to:

```
# The scripts consist of the following content:
# script1.t: "u=0x10"
# script2.t: "v=0x30"
# script3.t: "w=0x50"
ld.eld -o 1.out 1.o script1.t script2.t script3.t
```

Function	Description
`HIDDEN(symbol = expression)`	Hide the defined symbol so it is not exported.
`FILL(expression)`	Specify a fill pattern for the current output section. The fill element size can be 1, 2, 4, or 8 bytes (chosen by the linker). A `FILL` covers memory locations from the point at which it occurs to the end of the current output section; multiple `FILL` statements can be used within one output section.
`ASSERT(expression, string)`	If the specified expression is zero, the linker errors out with the specified message.
`PROVIDE(symbol = expression)`	Similar to a symbol assignment, but does not error if the symbol is already defined elsewhere.
`PROVIDE_HIDDEN(symbol = expression)`	Like `PROVIDE`, but the defined symbol is hidden (not exported).
`PRINT(“format-string”, expr, …)`	Print a formatted message while parsing the script. See the PRINT Command section for format string syntax and supported conversions.

NOCROSSREFS

- The NOCROSSREFS command takes a list of space-separated output section names as its arguments.
- Any cross references among these output sections will result in link editor failure.
- The list can also contain an orphan section that is not specified in the linker script.
- A linker script can contain multiple NOCROSSREFS commands.
- Each command is treated as an independent set of output sections that are checked for cross references.

OVERLAY

In GNU ld linker scripts, the OVERLAY command is used to define multiple sections that all execute (run) at the same virtual memory address (VMA), but are stored (loaded) at different load memory addresses (LMA). At runtime, only one of those sections is present in memory at a time, and software is responsible for copying the desired section into the overlay region before executing it.

What OVERLAY does in GNU ld

1. Same execution address (VMA) All sections inside an OVERLAY block are linked to start at the same address. From the CPU's point of view, they occupy the same memory region.
2. Different load addresses (LMA) Each overlaid section has a distinct load address in the output image (often in flash or ROM). These LMAs are laid out back-to-back starting at the overlay's AT(…) address.
3. Runtime-managed swapping The linker does not generate code to manage overlays. Your runtime code (overlay manager) must copy the desired section from its LMA into the overlay execution region before calling into it.

Auto-generated symbols in GNU ld

For each section inside an overlay, GNU ld defines:

- __load_start_<section>
- __load_stop_<section>

These symbols let your code know where to copy from.

NOCROSSREFS

GNU ld accepts an optional NOCROSSREFS keyword in the overlay header, e.g.:

```
OVERLAY 0x1000 : NOCROSSREFS AT(0x4000) { ... }
```

This causes GNU ld to error out if one overlaid section references another.

Effect on the location counter (.) in GNU ld

After an OVERLAY block, . is advanced by the size of the largest overlay member. This ensures the overlay region reserves enough execution memory for the largest case.

Syntax

```
OVERLAY [<start>] :
    [NOCROSSREFS] [AT(<lma_start>)]
{
    <overlay-member>...
} [><region>] [AT><lma_region>] [:<phdr>...] [=<fillexp>]

<overlay-member> :=
    <output-section-name> { <input-section-description>... }
```

Important

Overlay member sections are parsed as *name* + *body only*. Individual overlay members must not use the normal output section description prologue/epilogue syntax (for example, no : prologue, no member-level AT(…), no member-level >REGION/AT>REGION, no :PHDR list, and no =<fill>). eld errors out if these constructs are used on overlay members.

eld support status

eld currently supports parsing `OVERLAY` blocks and printing them into the text map file (`-MapStyle txt`) as comments. The GNU ld overlay *semantics* described above (LMA/VMA overlay placement, generated symbols, overlay-member swapping behavior, overlay-specific `NOCROSSREFS` enforcement, and location counter advancement rules) are not implemented yet.

Output Section Description

A `SECTIONS` command can contain one or more output section descriptions.

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

Syntax

<section-name>

Specifies the name of the output section.

<virtual_addr>

Specifies the virtual address of the output section (optional). The address value can be an expression (see Expressions).

<type>

Specifies the section load property (optional).

- NOLOAD: Marks a section as not loadable.
- INFO: Parsed only; has no effect on linking.

<load_addr>

Specifies the load address of the output section (optional). The address value can be specified as an expression (see Expressions).

<section_align>

Specifies the section alignment of the output section (optional). The alignment value can be an expression (see Expressions).

<subsection_align>

Specifies the subsection alignment of the output section (optional). The alignment value can be an expression (see Expressions).

<constraint>

Specifies the access type of the input sections (optional).

- NOLOAD: All input sections are read-only.

<output-section-command>

Specifies an output section command (see Output section commands). An output section description contains one or more output section commands.

<region>

Specifies the VMA placement memory region (optional) using `>REGION`.

If the output section does not have an explicit VMA, ELD uses the next available address in the region (starting from `ORIGIN` and respecting alignment) as the section's VMA.

If the output section has an explicit VMA, the explicit address is honored. The region is used for memory usage accounting only when the explicit VMA is within the region bounds.

<lma-region>

Specifies the LMA placement memory region (optional) using `AT>REGION`.

ELD places the section's LMA at the next available address in the selected region, tracked independently from the VMA region cursor. `AT(<address>)` and `AT>REGION` are mutually exclusive.

<fillexp>

Specifies the fill value of the output section (optional). The value can be an expression. This option is parsed, but it has no effect on linking.

<phdr>

Specifies a program segment for the output section (optional). To assign multiple program segments to an output section, this option can appear more than once in an output section description.

INSERT AFTER <section-name> | INSERT BEFORE <section-name>

Requests placing this output section after/before the named output section. The anchor section must exist, or the linker emits an error. Overlay member sections do not support output section epilogues, so INSERT is not allowed inside OVERLAY member blocks.

Sorting input sections

Linker scripts can control input-section ordering within an output section using sorting directives in input section descriptions. These directives include `SORT`, `SORT_BY_NAME`, `SORT_BY_ALIGNMENT`, and `SORT_BY_INIT_PRIORITY`.

ELD also supports the GNU linker shorthand `SORT(CONSTRUCTORS)` for compatibility with other linkers.

Controlling Physical addresses

In GNU linker scripts, the AT command is used to control the Load Memory Address (LMA) of a section, while the section's placement in memory during execution is defined by its Virtual Memory Address (VMA).

Important

When an AT command is specified as part of the output section, the linker will not automatically align the load memory address of the section.

ALIGN_WITH_INPUT attribute on an output section preserves the VMA-to-LMA offset from the previous output section when both sections use the same VMA region and the same LMA region. If either region changes, the linker does not reuse the prior offset and instead computes the LMA from the current output section's placement rules. This matches GNU behavior when combining explicit LMA control with region-based placement.

Behavior summary:

- VMA placement is governed by the output section's virtual address rules and the selected VMA region.
- LMA placement is governed by the AT/AT> directives and the selected LMA region, and it is tracked independently from the VMA placement.
- ALIGN_WITH_INPUT preserves the prior VMA-to-LMA delta, but only while both regions remain the same.

See also:

- test/Common/standalone/linkerscript/AlignWithInput/NoPhdrs/AlignWithInput.test
- test/Common/standalone/linkerscript/AlignWithInput/TLS/TLS.test
- test/Common/standalone/linkerscript/AlignWithInput/NoLoadATRAM/NoLoadATRAM.test

GNU-compatibility

The eld linker script syntax and semantics are GNU-compliant. This means that any linker script that works with the GNU linker should also work with eld, with the exception of a few GNU linker script features that are not yet supported by eld.

Previously, eld supported two extensions to the GNU linker script syntax. **These extensions are no longer supported.** Any scripts using these extensions must be updated to maintain compatibility with eld. These extensions are:

1. Assignment without space between the symbol and =

Previously supported:

```
symbol=<expr>
```

GNU-compliant syntax (required now):

```
symbol = <expr>
```

GNU requires a space between the symbol and the assignment operator. eld now enforces this requirement. Scripts must be updated accordingly.

2. Output section description without space between the output section name and `:`

Previously supported:

```
SECTIONS {
  FOO: {
    *(.text.foo)
  }
}
```

GNU-compliant syntax (required now):

```
SECTIONS {
  FOO : {
    *(.text.foo)
  }
}
```

GNU requires a space between the output section name and the colon. eld now enforces this requirement for full GNU compatibility.

Why cannot eld support these extensions along with GNU-compatibility?

eld cannot support these extensions along with GNU-compatibility because they directly conflict with the GNU linker script syntax. For example, GNU ld allows `:` in section names and allows = in symbol names. The core issue is that GNU ld uses the same lexing state to parse symbol and section names to keep the parser simple and efficient. Due to this, GNU ld also allows other non-trivial characters in symbol names such as +, -, `:` and so on. For example, for the below linker script snippet, gnu ld creates a symbol of the name a+=:

```
a+= = b # lhs symbol is a+=
```

eld cannot easily add exception to the two cases that were supported by eld extensions while keeping everything else the same to keep the linker script parser efficient. To support these as an exception, the parser needs to lookahead two tokens to resolve ambiguities. Let's understand this with the help of an example:

```
SECTIONS {
  FOO: {
    *(.text.foo)
  }
  u=v;
}
```

When parsing the `SECTIONS` commands, the parser does not know in which LexState to parse the command. If the command is an output section description, `FOO:`, then the parser should parse the token in `LexState::default`, whereas if the command is an assignment, then the parser should parse the token in `LexState::Expr`. `LexState::default` allows some characters in tokens that are not appropriate when parsing an expression. These characters include +, -, = and more.

To correctly determine which `LexState` to use, the parser needs to peek (lookahead) two tokens in `LexState::Expr`. With the two tokens peek, the parser can determine whether the command is an assignment command or not.

This simple change requires a lot of changes in the parser. The parser needs to change from LL(1) (Simple and efficient) to LL(2) (Complex and less efficient).

PRINT Command

Overview

The `PRINT` command lets a linker script emit formatted messages on the linker's standard output stream while the script is being parsed and evaluated. This is useful for debugging script expressions, inspecting symbol values, or annotating map files during development.

Syntax

The `PRINT` command has the following syntax:

```
PRINT("format-string", expression, ...)
```

Where:

- `format-string` is a C `printf`-style format string.
- Each `expression` is a regular linker script expression whose value is substituted into the format string.

The command does not define any symbols and does not affect the layout or contents of the linked image; it only produces textual output.

Format String Semantics

Basics

The format string closely follows C `printf` semantics with a limited set of conversion specifiers:

- %% – prints a literal % character; consumes no argument.
- %d – signed decimal integer.
- `%i` – signed decimal integer (alias for %d).
- %u – unsigned decimal integer.
- %o – unsigned octal integer.
- %x – unsigned hexadecimal integer (lowercase digits).
- %X – unsigned hexadecimal integer (uppercase digits).
- %c – single character; the low 8 bits of the numeric expression.
- %s – symbol name corresponding to a *symbol expression*.

All numeric expressions are evaluated as 64-bit integers. Length modifiers are accepted for compatibility but ignored; the value is always normalized to 64 bits before formatting.

Flags, Width, and Precision

The following flag characters are recognized:

- `-` – left-justify within the field width.
- `+` – always include a sign for signed conversions.
- ‘ ‘ (space) – prefix a space for positive signed conversions.
- `#` – alternate form (for example, add `0x` for hex).
- `0` – pad numeric fields with leading zeros.

Field width and precision have the usual `printf` syntax:

```
%[flags][width][.precision][length]conversion
```

with the following constraints:

- `width` and `precision` must be decimal integer constants.
- `*` is **not** supported for either width or precision; using it produces a diagnostics error.
- Supported length modifiers (`hh`, `h`, `l`, `ll`, `j`, `z`, `t`, `L`) are parsed but ignored – values are always treated as 64-bit.

Escape Sequences

The format string supports a small set of C-style escape sequences, which are unescaped before formatting:

- `\n` – newline
- `\t` – horizontal tab
- `\r` – carriage return
- `\\` – backslash
- `\"` – double quote

Any other escape sequence is left unchanged (the leading backslash is kept as-is).

Argument Matching and Errors

The linker validates the correspondence between conversion specifiers in the format string and the expressions supplied to `PRINT`. The following conditions are diagnosed as `PRINT` errors:

- Not enough arguments:
  - The format string requires more values than the number of expressions supplied (for example, `PRINT(“%d %u”, 1);`).
- Too many arguments:
  - More expressions are supplied than conversion specifiers in the format string (for example, `PRINT(“%d”, 1, 2);`).
- Unterminated or malformed format specifier:
  - A `%` at the end of the string (for example, `“value %”`).
  - A partially specified format that never reaches a conversion character.
- Unsupported conversion:
  - Any conversion character other than `d`, `i`, `u`, `o`, `x`, `X`, `c`, `s`, or `%` (for example, `%f`) is rejected.
- Unsupported width or precision:
  - Using `*` for width (for example, `“%*d”`).

    - Using * for precision (for example, “%. *d”).
- Invalid %s argument:
    - The expression corresponding to a %s conversion must be a *symbol expression* (such as foo). Passing an arbitrary numeric expression (for example, 1 + 2) is rejected.

On any of these conditions the linker emits an error_printcmd diagnostic and treats the PRINT invocation as a fatal error for that link.

Examples

Printing Numeric Expressions

```
/* Signed and unsigned forms */
PRINT("value=%d (0x%x)\\n", 42, 42);

SECTIONS {
  .text : { *(.text*) }
}
```

This prints a line similar to:

```
value=42 (0x2a)
```

Using %c and %s

```
/* Assume 'foo' is a symbol defined by an input object. */
PRINT("symbol %s at '%c' section start\\n", foo, 'T');
```

The %s conversion prints the symbol name (foo) while %c prints the low 8 bits of the numeric expression.

Using Width, Precision, and Flags

```
PRINT("val=%+08d hex=%#06x\n", 42, 42);

SECTIONS {
  .text : { *(.text*) }
}
```

This uses:

- + to always show the sign for the decimal value.
- 0 and a width of 8 to zero-pad the decimal field (for example, +0000042).
- # and a width of 6 for hexadecimal, causing a leading 0x and zero-padding in the remaining field (for example, 0x002a).

Error Examples

```
/* Not enough arguments */
PRINT("x=%d y=%d\\n", 1);

/* Unsupported conversion */
PRINT("value=%f\\n", 1);

/* Invalid %s argument */
PRINT("value=%s\\n", 1 + 2);
```

Each of these invocations produces a PRINT-related diagnostic and prevents the link from succeeding.

Diagnostics

Linker diagnostics refer to the messages and reports generated by a linker to help developers identify and resolve issues during the linking phase of compilation. These diagnostics can include:

- Warnings and errors about malformed or unsupported linker scripts
- Misplaced or overlapping sections
- Unresolved symbols
- Alignment issues
- Plugin-related inconsistencies

What *-Wall* and *-Werror* Mean for ELD

`-Wall`: Enable All Warnings

In the eld linker, `-Wall` enables a broad set of diagnostic warnings during the linking process. These warnings help developers catch:

- Misuse of linker script commands
- Compatibility issues during linking
- Linker script forward references
- Suspicious section overlaps or misalignments
- Deprecated or unsupported syntax
- Potentially undefined behavior in symbol resolution
- Command line usage

This flag is particularly useful in embedded development, where layout precision and deterministic behavior are critical.

`-Werror`: Treat Warnings as Errors

When `-Werror` is used, any warning promoted by `-Wall` (or other `-W` flags) is treated as a **fatal error**, halting the linking process. This ensures:

- No warnings are ignored during CI/CD builds
- Developers are forced to resolve all issues before producing a final binary
- Consistency across builds, especially when linker behavior may vary across toolchain versions

This is especially important in embedded workflows, where eld is used to build critical components like firmware and device drivers.

Why It Matters in ELD

- **Embedded Safety**: In embedded systems, even minor layout issues can lead to runtime faults. `-Wall -Werror` ensures these are caught early.
- **Plugin Infrastructure**: ELD supports linker plugins. These can emit custom diagnostics, and `-Wall` ensures they're surfaced; `-Werror` enforces them.
- **Script Compatibility**: ELD aims for GNU linker script compatibility. These flags help validate script correctness during migration from GNU ld.

Example Usage

```
eld -T script.ld -o firmware.elf main.o -Wall -Werror
```

This command will:

- Use `script.ld` for layout
- Link `main.o` into `firmware.elf`
- Emit all warnings (`-Wall`)
- Fail the build if any warning is encountered (`-Werror`)

What are the various categories that are grouped in

ELD Linker Warning Flags

This document provides a categorized list of warning flags supported by the ELD linker.

General Warning Flags

Flag	Description
`-Wall`	Enables all standard warnings supported by ELD. Useful for catching script and layout issues early.
`-Warchive-file`	Warns about issues related to archive (.a) when they contain duplicate members.
`-Wattribute-mix`	Flags inconsistent or conflicting section attributes across input files.
`-Wbad-dot-assignments`	Warns when the `.` (location counter) is used in a way that may cause layout issues or undefined behavior.
`-Wcommand-line`	Enables warnings for malformed or conflicting command-line options.
`-Werror`	Treats all warnings as errors, halting the link process.
`-Wlinker-script`	Enables warnings specific to linker script issues, such as malformed directives, deprecated syntax, or unsupported constructs.
`-Wlinker-script-memory`	Focuses on memory region definitions in linker scripts, such as overlaps or undefined regions.
`-Wwhole-archive`	Warns when `–whole-archive` is used inappropriately or causes symbol bloat.
`-Wzero-sized-sections`	Flags sections that are defined but have zero size, which may indicate a script or input error.
`-Wosabi`	Generates a warning when an input file's OS/ABI value differs from those encountered in previously processed files.

Suppression Flags

Flag	Description
`-Wno-archive-file`	Suppresses archive file-related warnings.
`-Wno-attribute-mix`	Suppresses attribute mismatch warnings.
`-Wno-error`	Allows warnings to be emitted without halting the link process.
`-Wno-linker-script`	Disables linker script-related warnings.
`-Wno-osabi`	Disables OS/ABI warnings.

ELD (debugging guide)

This document describes the high-level flow of how ELD executes a link, with call-site pointers and practical tips for debugging failures.

Big picture

At a high level, a link invocation looks like this:

1. **eld main** expands response files and selects a driver flavor/target.
2. **Driver** parses options and builds an ordered list of input actions.
3. **Linker prepare** initializes target/emulation, inputs, and plugins, then reads and normalizes input files (and may run LTO).
4. **Linker link** resolves symbols/relocations, lays out output sections, then emits the final ELF and optional map files.
5. **Optional diagnostics**: reproduce tarball/mapping file, plugin activity log, timing stats, summary, etc.

Entry point and driver selection

The executable entry point is `tools/eld/eld.cpp`:

- Expands `@response` files via `llvm::cl::ExpandResponseFiles(…)`.
- Creates a `Driver` and calls `Driver::setDriverFlavorAndInferredArchFromLinkCommand(…)`.
- Creates a GNU-ld-compatible driver (`GnuLdDriver`) and calls `GnuLdDriver::link(…)`.

Driver flavor selection is implemented in `lib/LinkerWrapper/Driver.cpp`:

- First tries to infer a target from the program name (e.g. `arm-link`, `aarch64-link`, `hexagon-link`).
- Otherwise inspects early arguments like `-m <emulation>` or `-march` to select a target-specific driver.

Environment hooks that affect arguments:

- `ELDFLAGS`: appended to the link command by the driver (useful for always-on debug flags).

Argument parsing and preprocessing

The top-level flow of option parsing and dispatch is in `lib/LinkerWrapper/GnuLdDriver.cpp` (`GnuLdDriver::link(…)`):

1. `parseOptions(…)`
2. `processLLVMOptions(…)` (parses `-mllvm …` arguments)
3. `processTargetOptions(…)` (handles `-mtriple`, `-march`, `-mabi`, `-m <emulation>`, etc.)
4. `processOptions(…)` (general linker options)
5. `checkOptions(…)` and `overrideOptions(…)`
6. `createInputActions(…)` to build the ordered action list
7. `doLink(…)` to run the actual link pipeline

If you suspect argument/option issues, start with:

- `–verbose` (or `–verbose=<level>`)
- `–trace=command-line`
- `–trace=files` or `-t` (prints processed files)
- `–error-style=GNU` or `–error-style=LLVM` (if output formatting matters)

Input actions (what gets fed to the linker)

After parsing options, the driver builds a sequence of actions that are later “activated” to create inputs:

- `-T <script>` / `–default-script <script>` -> `ScriptAction`
- `-R <file>` -> `JustSymbolsAction`
- `–defsym <sym>=<expr>` -> `DefSymAction`
- `-l <namespec>` -> `NamespecAction` (library search)
- Plain inputs -> `InputFileAction`
- State toggles like `–whole-archive`, `–as-needed`, `–start-group`/`–end-group`, `–start-lib`/`–end-lib` -> corresponding actions that affect how subsequent inputs are treated

This happens in `GnuLdDriver::createInputActions(…)` in `lib/LinkerWrapper/GnuLdDriver.cpp`.

Debug tip: if you see failures about mismatched groups/libs, the error is detected here (before any ELF parsing starts).

Link pipeline overview (doLink -> Linker)

Once actions are created, `GnuLdDriver::doLink(…)` does the setup:

- Looks up the LLVM target and the ELD target based on the chosen triple.
- Creates a `Module` (`lib/Core/Module.cpp`) and optional `LayoutInfo` for map printing.
- Selects map printers based on `–MapStyle=…` (or defaults) and prepares layout printers.
- Constructs an `eld::Linker` and runs: * `linker.prepare(actions, target)` * `linker.link()` (unless running “LTO-only” modes) * `linker.printLayout()` (map file emission)
- Runs plugin teardown hooks, unloads plugins, emits stats, and finalizes the diagnostic engine summary.

Prepare phase (Linker::prepare)

`Linker::prepare(…)` (`lib/Core/Linker.cpp`) is responsible for:

1. **Target/emulation + backend**
    - Initializes emulator and backend for the selected target.

2. **Initialize inputs**
    - Builds the input tree, creates internal linker-generated inputs, activates the action list, and reads linker scripts.
3. **Universal plugins**
    - Reads plugin configuration, loads universal plugins from the script, stores them, and runs plugin init hooks.
4. **Read/normalize inputs**
    - Reads all input files, sections, symbols, and (optionally) runs LTO-related preprocessing.

Common debug levers for this phase:

- `–trace=linker-script` or `–trace-linker-script` (script parsing)
- `–trace=threads` (parallel input/section reading behavior)
- `–trace=LTO` or `–trace-lto` (LTO stage boundaries)
- `–plugin-config=<config-file>` and `–no-default-plugins` (plugin triage)

Normalize phase (Linker::normalize)

`Linker::normalize()` (`lib/Core/Linker.cpp`) performs:

- Optional command-line header/summary printing when `–trace=command-line` is enabled (via `LinkerConfig::printOptions(…)` in `lib/Config/LinkerConfig.cpp`).
- Reading all input files via `ObjLinker->normalize()`: * Parses ELF objects/archives/shared libraries/bitcode inputs. * Populates symbol tables and initial symbol resolution.
- Loads non-universal plugins.
- Computes code position (static/dynamic/PIE) and validates incompatible options (e.g. patch options with non-static output).
- Parses external scripts:
    - Version scripts
    - Dynamic list (when building dynamic artifacts)
- Adds linker-script-defined symbols.
- LTO steps (when needed):
    - Creates an LTO object from bitcode inputs.
    - Re-runs normalization post-LTO after replacing bitcode with generated objects.

Debug tip: LTO-related failures often reproduce reliably with `–reproduce` because the tarball will include bitcode inputs and any generated LTO objects recorded by the linker.

Resolve + layout + emit (Linker::link)

`Linker::link()` in `lib/Core/Linker.cpp` is the main “work” phase:

1. **Standard sections**
    - Initializes default sections (and per-file synthetic dynamic sections when producing dynamic outputs).
2. **Resolve (symbol/reloc processing)**
    - Reads relocations.
    - Allocates common symbols.
    - Assigns output sections using default + linker script rules.
    - Runs plugin hooks around rule matching/layout.
    - Processes target-specific input handling.

- Optionally performs garbage collection / stripping.
- Scans relocations, finalizes scan results, and builds output/dynamic symbol tables.
- Merges sections and creates section symbols.

3. **Layout**
   - Initializes stubs/trampolines, prelayout, merge-strings optimization.
   - Establishes final layout and postlayout output section table.
   - Finalizes symbol values, runs output-section iterator plugins, applies relocations, and finalizes output state.
4. **Emit**
   - Computes output file size and creates an `llvm::FileOutputBuffer`.
   - Writes section contents, performs post-processing, emits Build ID, commits the output, and optionally verifies the output size on disk.

If a failure happens late (layout/emit), map files are usually the fastest way to pinpoint the problematic section/segment/relocation.

Internal inputs and “internal sections”

ELD creates a number of linker-generated inputs up front so later stages can treat them uniformly as normal inputs/sections/symbols.

Creation happens in `Module::createInternalInputs()` (`lib/Core/Module.cpp`). Each internal input corresponds to a named `Input`/`InputFile` (for example `Attributes`, `CommonSymbols`, `DynamicSections`, `Trampoline`, and others) and is used to host sections/fragments that are not sourced from a user object file.

Two other “internal” concepts are easy to confuse:

- **Linker-internal input sections**: sections owned by an internal input file (`Input::Internal`). These typically have `LDFileFormat::Internal` kind and may carry relocations to be applied later.
- **Output-format sections**: sections that come from the backend/output format (not from a user input file), for example dynamic tables/headers. These are treated as output sections and can be matched/discarded via linker-script rules; see `ObjectLinker::markDiscardFileFormatSections()` in `lib/Object/ObjectLinker.cpp`.

Debug tips:

- If you suspect an unexpected section exists (or is missing), prefer a text map: `-M –MapStyle=Text –Map=<file>`.
- If a section is unexpectedly discarded, use `–trace-section <name>` and check whether it matched a `/DISCARD/` rule.

Section merging (input sections -> output sections)

The “merge sections” name in ELD means: take the *input* section graph (from object files + internal inputs), and populate the *output* section layout according to default rules + linker-script rules + plugins.

There are three distinct sub-steps to keep straight:

1. **Rule matching / output section assignment**
   - `ObjectLinker::assignOutputSections(…)` (`lib/Object/ObjectLinker.cpp`) uses an `ObjectBuilder` to match input sections against linker-script rules (including wildcards, sorting policies, `EXCLUDE_FILE`, etc).
2. **Input section merging**
   - `ObjectLinker::mergeSections()` calls `mergeInputSections(…)` which iterates all input sections and merges them into output sections, with special handling for some section kinds (`.eh_frame`, `.sframe`, target overrides, linkonce/reloc sections, etc).

3. **Finalize output sections**

    - `createOutputSection(…)` builds each output section's fragment list, computes flags/alignment, assigns fragment offsets, and inserts the output sections into the module's output section table.

Special-case section handling during merging

`ObjectLinker::mergeInputSections(…)` (`lib/Object/ObjectLinker.cpp`) handles some input section kinds specially:

- `LDFileFormat::Relocation` and `LDFileFormat::LinkOnce`: if the "link" section is discarded/ignored, the relocation/linkonce section is ignored too.
- `LDFileFormat::Target`: backends may override merging via `GNULDBackend::DoesOverrideMerge(…)` and `GNULDBackend::mergeSection(…)`.
- `LDFileFormat::EhFrame`:
    - Splits and re-chunks `.eh_frame` into CIE/FDE fragments.
    - If enabled, registers content for `.eh_frame_hdr` and creates filler/hdr fragments in the backend.
- `LDFileFormat::SFrame`:
    - Parses the section and may create an `SFrame` header fragment when configured.

Everything else typically flows through `ObjectBuilder::mergeSection(…)` and ends up contributing fragments to an output section.

Output section construction and offsets

Once merging decides *which* fragments belong to a particular output section, `ObjectLinker::createOutputSection(…)` and `ObjectLinker::assignOffset(…)` lay them out:

- Output section `ALIGN` / input section `SUBALIGN` (from linker script) is enforced when present.
- "Dirty" rules (modified by plugins) trigger a recomputation of input section flags/type/align based on the fragments that ended up in the rule.
- Fragment offsets are assigned linearly; per-fragment padding/alignment is applied by `Fragment::paddingSize()` (`lib/Fragment/Fragment.cpp`).

Debug tips:

- If you see "offset not assigned" diagnostics, the fragment/section likely never got placed into an output section (or got discarded). The diagnostic plumbing is in `Fragment::getOffset(…)` (`lib/Fragment/Fragment.cpp`).
- If rule sorting changes layout unexpectedly, check whether the linker script wildcard includes a sort policy, or whether `–sort-section=…` is enabled.

String merging (MergeString fragments)

String merging is a dedicated optimization pass that runs during layout, before final output layout is established:

- `ObjectLinker::doMergeStrings()` calls:
    - `mergeIdenticalStrings()`: merges `MergeStringFragment` content (can be threaded across output sections; global non-alloc merge is done single-threaded).
    - `fixMergeStringRelocations()`: updates relocations that refer into merged strings via `Relocator::doMergeStrings(…)`.

The output offset calculation for merged strings is special-cased in `FragmentRef::getOutputOffset(…)` (`lib/Fragment/FragmentRef.cpp`), because multiple input strings may map to a shared output string (including suffix merging).

Debug tips:

- Use `–trace=merge-strings` / `–trace-merge-strings=<option>` to see why strings were merged and how offsets were computed.

Relocations: read -> scan -> apply -> (optional) emit reloc sections

There are multiple relocation passes, and confusing them is a common source of “where did this relocation come from?” debugging pain.

Read relocations

`ObjectLinker::readRelocations()` reads relocations from input objects (`lib/Object/ObjectLinker.cpp`):

- Skips non-object inputs, and skips inputs marked “just symbols”.
- For patch-base inputs, runs patch-base parsing via the executable-object parser.

Scan relocations (reservation / planning)

`ObjectLinker::scanRelocations(…)` (`lib/Object/ObjectLinker.cpp`) is the “planning” pass. It typically:

- Invokes `Relocator::scanRelocation(…)` per relocation, which is where the backend decides whether it needs to reserve GOT/PLT entries, create or reserve dynamic relocations, create stubs/trampolines, etc. (target-specific logic is in `lib/Target/*/*Relocator.cpp`).
- Collects copy-relocation candidates per input, then creates copy relocations once per symbol (see `createCopyRelocation(…)` / `addCopyReloc(…)` in `lib/Object/ObjectLinker.cpp`).
- Merges per-file dynamic relocation vectors into a single “reloc input” (`getDynamicSectionHeadersInputFile()`) so later code can treat them consistently.
- Runs `ObjectLinker::finalizeScanRelocations()` which calls `GNULDBackend::finalizeScanRelocations()` for backend-specific finalization.

In relocatable/partial links, ELD uses `partialScanRelocation(…)` instead.

Create output relocation sections (`–emit-relocs`)

If `–emit-relocs` is enabled, ELD creates output relocation sections during prelayout:

- `ObjectLinker::prelayout()` calls `createRelocationSections()`.
- `createRelocationSections()` counts relocations per output section and creates the corresponding output relocation sections (`.rel.<sec>` / `.rela.<sec>` style, based on target) sized to hold all entries.

Apply relocations (writes relocation results)

Relocation application happens in `ObjectLinker::relocation(…)`:

- Applies internal/linker-created relocations.
- Applies input relocations, skipping relocations that are known to be relaxed or that target discarded/ignored sections/symbols.
- Applies branch-island (relaxation) relocations after input relocations are applied.
- If `–emit-relocs` is enabled, emits external-form relocation records into the output relocation sections (via `EmitOneReloc`).

Finally, `syncRelocations(…)` writes relocation results into the output buffer, including extra ordering/barriers to avoid races when multi-threaded.

Debug tips:

- `–trace=reloc=<pattern>` pinpoints a single relocation kind.
- `–trace=symbol=<name>` helps tie relocations back to symbol resolution.
- If you see overflows/unencodable immediates, diagnostics originate from `Relocation::issueSignedOverflow(…)` / `issueUnencodableImmediate(…)` in `lib/Readers/Relocation.cpp`.

Dynamic relocations (what creates `.rel[a].dyn` / `.rel[a].plt`)

Dynamic relocation entries are typically created/reserved during the relocation scan phase inside the target relocator and backend:

- Target relocators decide whether a given relocation needs: * a static relocation only, * a dynamic relocation (REL/RELA), * a PLT/GOT entry (and an associated relocation), * a copy relocation (for executable data symbol imports).
- Backends provide the actual sections for dynamic relocations (for example `.rela.dyn` / `.rela.plt`) and may sort/finalize them. A common set of helper logic lives in `lib/Target/GNULDBackend.cpp`.

You can generally think of the relocation scan as “reserving and populating dynamic relocation vectors”, and layout/emission as “placing and writing those sections”.

Garbage collection (`–gc-sections`)

Garbage collection in ELD is graph reachability over sections, built from relocations and a chosen root set (“entry sections”).

Where it runs

The default GC pass is triggered during the resolve phase:

- `ObjectLinker::dataStrippingOpt()` checks `IRBuilder::shouldRunGarbageCollection()` and calls `ObjectLinker::runGarbageCollection("GC")`.
- The implementation is `GarbageCollection` in `lib/GarbageCollection/GarbageCollection.cpp`.

How the graph is built

`GarbageCollection::setUpReachedSectionsAndSymbols()`:

- Traverses input relocations and records, per “apply section”, the set of reachable target sections and reachable symbols.
- Handles special cases:
  - Script-defined symbols: walks the assignment expression’s symbol references.
  - Magic `__start_*/__stop_*` symbols: forces sections with matching names into the reachable set.
  - Bitcode: defers some reachability until it can map referenced symbols back to bitcode “input sections”.
- Allows the backend to add extra reachability via `GNULDBackend::setUpReachedSectionsForGC(…)`.

How entry sections are chosen

`GarbageCollection::getEntrySections()` considers multiple root sources:

- The configured entry symbol (if it resolves to a fragment).
- Sections matched by `KEEP(…)` in the linker script.
- When producing dynamic outputs, exported/visible symbols (subject to version script scoping) contribute entry sections.
- Sections marked with `SHF_GNU_RETAIN` are treated as entry-like.

Mark-and-sweep

`GarbageCollection::findReferencedSectionsAndSymbols(…)` performs a BFS from entry sections, following the reachability map built earlier, producing a live set. `stripSections(…)` then marks sections not in the live set as ignored (and can optionally print what got collected).

Debug tips:

- `–print-gc-sections` shows what got collected.

- `–trace=garbage-collection` and `–trace=live-edges` are useful when a section is unexpectedly dead/alive.
- If GC keeps/drops a zero-sized section unexpectedly, check whether it is the target of a relocation or contains a symbol (see the FAQ discussion of zero-sized sections).

Fragment model (Fragment / FragmentRef)

ELD uses a fragment model internally where **fragments are the minimum linking unit**, not sections.

Fragments

`Fragment` (`include/eld/Fragment/Fragment.h`) represents a typed chunk of content that will appear in the output. Examples include:

- raw data regions (region fragments),
- stubs / trampolines / branch island content,
- GOT / PLT entries,
- mergeable strings,
- `.eh_frame`-related pieces (CIE/FDE fragments),
- build-id fragments, timing fragments, and others.

Each fragment belongs to an owning (input) section and has:

- an alignment requirement,
- an assigned (unaligned) offset, and derived padding size,
- an `emit(…)` implementation that writes bytes during output generation.

Offsets, padding, and “why is this input marked used?”

During output section construction, ELD assigns fragment offsets linearly. The final effective offset includes per-fragment padding computed by `Fragment::paddingSize()` (`lib/Fragment/Fragment.cpp`). When a fragment offset is assigned, `Fragment::setOffset(…)` also marks the owning input as “used” when it contributes allocatable content (this feeds into GC and diagnostics).

FragmentRef (symbol/relocation addressing)

`FragmentRef` (`include/eld/Fragment/FragmentRef.h`) is a pointer to:

- a fragment, plus
- an additional byte offset within that fragment.

This is the core indirection used by:

- output symbols (symbols carry a `FragmentRef` to their definition),
- relocations (relocation “place” and/or “target” is a `FragmentRef`).

Output offset computation is not always “fragment offset + ref offset”:

- `FragmentRef::getOutputOffset(…)` special-cases `.eh_frame` to map offsets through the split/piece layout.
- It also special-cases merged strings so references land on the deduplicated output string (including suffix merging).

Debug tips:

- If a relocation points somewhere surprising, inspect: * the relocation’s `targetRef` (place), and * the symbol’s `fragRef` (definition), and remember both can have special-cased output offset behavior.

Map files (layout printers)

Map emission is handled after the link attempt in `Linker::printLayout()` and also from the crash signal handler.

Key options:

- `-M` / `–print-map`: enable map generation
- `–Map=<filename>`: choose map output file
- `–MapStyle=<YAML|Text|Binary>`: choose format(s)
- `–MapDetail=<option>`: more detail in maps
- `–color-map`: colorize map output
- `–trampoline-map <filename>`: trampoline information (YAML)

Reproducing failures (tarball + mapping file)

ELD can capture a self-contained reproducer for link issues:

- `–reproduce <tarfilename>`: always produce a tarball
- `–reproduce-compressed <tarfilename>`: compressed tarball
- `–reproduce-on-fail <tarfilename>`: only on failure
- `ELD_REPRODUCE_CREATE_TAR`: environment variable that forces reproducer creation (uses a temporary tar file if no filename is provided)

Additional reproduce helpers:

- `–mapping-file <INI-file>`: reproduce link using a mapping file
- `–dump-mapping-file <outputfilename>`: dump mapping info
- `–dump-response-file <outputfilename>`: dump rewritten response file

The reproduce tarball logic is wired through:

- `GnuLdDriver::handleReproduce(…)` and `writeReproduceTar(…)` in `lib/LinkerWrapper/GnuLdDriver.cpp`
- `Module::createOutputTarWriter()` creation decision via `LinkerConfig::shouldCreateReproduceTar()` (`lib/Config/LinkerConfig.cpp`)

Crash/signal behavior (what gets written on a crash)

ELD installs a default signal handler in `GnuLdDriver::doLink(…)`:

- Flushes a text map file (if configured).
- Detects likely plugin crashes and reports them.
- Writes a temporary `.sh` script that appends `–reproduce build.tar` to the command line and instructs the user to rerun.

This is implemented in `GnuLdDriver::defaultSignalHandler(…)` in `lib/LinkerWrapper/GnuLdDriver.cpp`.

Where failures typically come from (symptoms -> pipeline stage)

This section is meant as a quick index: if you see a symptom, these are the stages/files to inspect first. Many of these topics are also discussed in more detail in `docs/userguide/documentation/linker_faq.rst`.

Driver/target selection failures

Symptoms:

- “unsupported emulation” / “cannot find target”
- wrong backend chosen when using `-m` / `-march`

Start here:

- `Driver::getDriverFlavorFromLinkCommand(…)` in `lib/LinkerWrapper/Driver.cpp`
- `GnuLdDriver::processTargetOptions(…)` and target lookups in `GnuLdDriver::doLink(…)` (`lib/LinkerWrapper/GnuLdDriver.cpp`)

Input specification / archive/group issues

Symptoms:

- “mismatched group” / “mismatched lib”
- unexpected missing objects from an archive

Start here:

- `GnuLdDriver::createInputActions(…)` for `–start-group`/`–end-group` and `–start-lib`/`–end-lib` balancing and ordering
- Use `-t` / `–trace=files` to confirm what ELD actually processed

Linker script rule-matching errors

Symptoms:

- “no linker script rule for “.bss”” / “.data.bar” style errors
- sections landing in unexpected output sections

Start here:

- `ObjectLinker::assignOutputSections(…)` and friends in `lib/Object/ObjectLinker.cpp`
- Emit a map file and confirm whether the input section matched any rule; the FAQ has a guide for diagnosing these errors and for finding used/unused rules.
- If this is a *script parsing/syntax* problem (rather than rule matching), enable `–trace=linker-script` / `–trace-linker-script` and use `–reproduce[-on-fail]` to capture the exact script(s) and rewritten command line that ELD used.

Practical tips:

- Reduce the script: comment out most rules and add back until the behavior flips. For rule-matching bugs, keep only the relevant `SECTIONS` rules and a minimal `MEMORY` map.
- Prefer map files while iterating: they show which input section went to which output section and why that changed across experiments.

Linker script parsing and evaluation errors

Symptoms:

- parse errors (unexpected token, unexpected `)`, etc)
- script expression issues (undefined symbol in an expression, unexpected value)
- placement issues that are script-driven (for example: region overflow, PHDR mismatch, or a section being forced into an incompatible segment)

Start here:

- `–trace=linker-script` / `–trace-linker-script` to see script parsing, includes, and key evaluation decisions.
- A text map file to confirm what the script actually did: memory regions, output section addresses, and segment layout are usually visible there.

- `–reproduce[-on-fail]` so the exact script(s) used by the link are captured alongside the rewritten response file (this is critical when scripts are generated by the build system).

Undefined references and symbol resolution surprises

Symptoms:

- “undefined reference” failures
- symbol unexpectedly resolved from a different archive/object

Start here:

- Resolve phase in `Linker::resolve()` (`lib/Core/Linker.cpp`) and the diagnostic engine output
- Use `–trace=symbol=<name>` (or `–trace=all-symbols`) to see the resolution path

When the failure is a *runtime* crash (not a link error), symbol resolution can still be the root cause:

- Wrong interposition/visibility (a symbol resolves but to an unexpected definition at runtime).
- Lazy binding via PLT (a crash happens on the first call into a function that is resolved late by the dynamic loader).

Quick inspections:

```
# Symbol tables (static and dynamic), with type/binding/visibility:
llvm-readelf -s --demangle --extra-sym-info ./app | less
llvm-readelf -s --demangle --extra-sym-info ./libfoo.so | less

# Dynamic symbols only (what the runtime loader sees):
llvm-readelf --dyn-syms --demangle --extra-sym-info ./libfoo.so | less

# Undefined dynamic symbols (what must be provided by dependencies):
llvm-readelf --dyn-syms --demangle --extra-sym-info ./libfoo.so | awk '$7==\"UND\" {print}'

# Imported/Exported symbols (alternate views):
nm -D --defined-only ./libfoo.so | less
nm -D --undefined-only ./libfoo.so | less
```

Garbage collection removed something needed

Symptoms:

- function/data present in inputs but missing from output
- a section disappears only with `–gc-sections`

Start here:

- `GarbageCollection` implementation in `lib/GarbageCollection/GarbageCollection.cpp`
- `–print-gc-sections` plus `–trace=garbage-collection` / `–trace=live-edges`
- Ensure linker-script `KEEP(…)` is used for sections that must never be GC’d

Relocation overflows / unencodable immediates / target-specific relocation bugs

Symptoms:

- overflow/unencodable relocation diagnostics
- crashes during relocation scan/apply
- output runs but has wrong addresses at runtime

Start here:

- Scan phase: `ObjectLinker::scanRelocations(…)` and target relocators (`lib/Target/*/*Relocator.cpp`)
- Apply phase: `ObjectLinker::relocation(…)` and sync/writeback (`lib/Object/ObjectLinker.cpp`)

- Diagnostics: `lib/Readers/Relocation.cpp` (location printing, overflow, etc)

Trampolines / stubs / relaxation issues

Symptoms:

- failures mentioning trampolines, far calls, or branch islands
- layout changes causing new trampolines or changing trampoline reuse

Start here:

- `ObjectLinker::initStubs()` and target stub factories/backends
- Map/trampoline map options (`–trampoline-map`) plus FAQ sections on trampoline naming and reuse controls

LTO failures

Symptoms:

- failures only with `-flto` / ThinLTO / Full LTO
- “LTO merge error” / codegen diagnostics

Start here:

- `ObjectLinker::createLTOObject()` and LTO diagnostics handler in `lib/Object/ObjectLinker.cpp`
- `–trace=LTO` / `–trace-lto` and `–reproduce[-on-fail]` to capture inputs and generated objects
- `–save-temps` (or `–save-temps=<dir>`) to preserve intermediate LTO artifacts for inspection (files use the prefix `<output>.llvm-lto.*`)
- If you need stable, non-temporary LTO-generated objects: `–lto-obj-path=<prefix>` (also keeps the objects from being deleted after LTO)

Plugin-caused failures

Symptoms:

- crashes only when a plugin is configured
- non-deterministic behavior across runs with the same inputs

Start here:

- `–plugin-activity-file=<file>` to capture plugin activity
- `–no-default-plugins` to isolate
- Crash handler output from `GnuLdDriver::defaultSignalHandler(…)` can explicitly call out a plugin as the likely crash source

Output emission failures

Symptoms:

- “unwritable output” / commit errors
- output size verification failures

Start here:

- `Linker::emit()` in `lib/Core/Linker.cpp` (`llvm::FileOutputBuffer` creation/commit)

Practical debugging checklist

When a link fails and you need actionable data quickly, try (in order):

1. Add `–reproduce-on-fail repro.tar` (or `–reproduce repro.tar`).
2. Add `–verbose –trace=command-line –trace=files`.

3. Enable map output: `-M –Map=layout.map –MapStyle=Text` (or YAML).
4. If plugins are involved: `–plugin-activity-file=plugins.json` and try `–no-default-plugins` to isolate.
5. If time-sensitive or flaky: `–print-timing-stats` and consider `–emit-timing-stats=<file>` to capture timing consistently.

Debugging runtime crashes in ELD-built images

This section is for cases where the link *succeeds* but the produced ELF image (executable / shared library / firmware image) fails at runtime (crash, abort, unexpected exception, bad unwind/backtrace, etc.).

Preserve the right artifacts

For runtime debugging, the most common blocker is having only a stripped image with no symbols or line tables.

Keep (or be able to re-create) at least:

- The exact linked ELF that ran (same build-id if you use build-ids).
- An unstripped ELF (or a separate `.debug` file) that matches the runtime image.
- The ELD map file (`–Map=… –MapStyle=Text` or YAML) for fast address-to-section/symbol correlation.
- The crash report: PC/LR/SP, full backtrace if available, and a core dump when possible.

If your production image must be stripped, keep debug info out-of-band using `llvm-objcopy` (or GNU `objcopy`):

```
llvm-objcopy --only-keep-debug app app.debug
llvm-objcopy --strip-debug --strip-unneeded app app.stripped
llvm-objcopy --add-gnu-debuglink=app.debug app.stripped
```

Use linker map files for layout correlation

When debugging runtime failures that are layout-sensitive (for example: wrong PLT/GOT access, unexpected text/rodata placement, thunk/trampoline differences, RELRO placement), generate and keep a linker map file for the exact link:

```
eld ... -M --Map=layout.map --MapStyle=Text
```

The map file is often the fastest way to answer: “which output section/segment contains this address?” and “why did this archive member/section get pulled in?”.

Symbolize a crash address (PC) quickly

If you have an address from a crash report (for example `PC=0x…`) you can usually get file:line without opening a debugger:

```
# Pick one:
llvm-addr2line -f -C -e ./app 0xADDR
addr2line      -f -C -e ./app 0xADDR
```

For PIE executables and shared libraries under ASLR, `0xADDR` is typically a *runtime virtual address*. Convert it to an ELF-relative address first:

1. Find the module load base (`/proc/<pid>/maps` for a running process, or `image list -o -f` in lldb for a core).
2. Compute `REL = ADDR - BASE`.
3. Run `addr2line` on `REL` using the corresponding module file.

If you are using sanitizers, make sure symbolization is enabled and points at a working symbolizer:

- `LLVM_SYMBOLIZER_PATH=/path/to/llvm-symbolizer`
- `ASAN_OPTIONS=symbolize=1:abort_on_error=1` (plus your project defaults)
- `UBSAN_OPTIONS=print_stacktrace=1`

Debug with lldb (core dumps and live debugging)

For a core dump:

```
ulimit -c unlimited
./app   # reproduce crash
lldb -c core ./app
```

On systems using `systemd-coredump`, `coredumpctl` is often the easiest:

```
coredumpctl list ./app
coredumpctl dump <PID> --output=core
lldb -c core ./app
```

In lldb, start with:

- `bt` / `thread backtrace all`
- `register read` and `disassemble -m -p –start-address $pc`
- `image list -o -f` (verify loaded modules + load addresses)

If you cannot run the image locally (cross/embedded), use `lldb-server` on the target and attach from the host toolchain debugger.

Run musl builds under qemu (quick cross-runtime triage)

If you need a fast, reproducible runtime environment (especially for cross-target issues), a practical workflow is: build a small musl-based binary and run it under qemu (user-mode or system-mode).

User-mode qemu (recommended for simple tests)

For Linux user-mode emulation, run the target executable directly:

```
# Examples (pick your target qemu):
qemu-aarch64 ./app
qemu-arm     ./app
qemu-riscv64 ./app
qemu-riscv32 ./app
qemu-x86_64  ./app
```

If the binary is dynamically linked, point qemu at a sysroot that contains the target loader + shared libraries (musl or glibc as appropriate):

```
qemu-aarch64 -L /path/to/<triplet>/sysroot ./app
```

Common qemu-user tips:

- `-strace` prints syscalls (useful when a crash is actually an `ENOENT` / loader issue).
- `-E VAR=…` sets environment variables for the emulated program (for example `-E LD_LIBRARY_PATH=…`).
- `-d in_asm,cpu,exec -D qemu.log` logs executed instructions; it is noisy but can pinpoint the last instruction before a crash.
- `-g <port>` enables a gdbstub; you can attach with lldb using gdb-remote:

  ```
  qemu-aarch64 -g 1234 ./app
  lldb ./app
  (lldb) gdb-remote 1234
  ```

System-mode qemu (when you need a full OS image)

Use system-mode (`qemu-system-*`) when user-mode is insufficient (for example: kernel/driver interactions, missing syscalls, or you need a full rootfs).

In system-mode, typical debugging flags include:

- `-s -S` (open gdbstub and stop at reset)

- `-d in_asm,cpu,exec -D qemu.log` (instruction logging; very verbose)

Inspect exception handling and unwinding

Runtime failures that look like “crash while unwinding”, “terminate called after throwing”, or incorrect backtraces typically reduce to missing/mismatched unwind or exception tables.

At link time, ELD may merge/synthesize unwind-related sections (for example `.eh_frame` and `.eh_frame_hdr`) and can also process SFrame (`.sframe` with `–sframe-hdr`). For ARM EHABI you may also see `.ARM.exidx` / `.ARM.extab`.

Quick checks:

```
llvm-readelf -S ./app | grep -E \"eh_frame|eh_frame_hdr|gcc_except_table|ARM\\.exidx|ARM\\.extab|sframe\"
llvm-readelf --unwind ./app   # unwind info (includes .eh_frame when present)
```

Common causes of missing/insufficient unwind info:

- Built without unwind tables (toolchain flags such as `-fno-asynchronous-unwind-tables`, `-fno-unwind-tables`).
- Over-aggressive stripping (for example link-time `–strip-debug`) combined with not keeping a matching `.debug` file.
- Inconsistent binaries/libraries at runtime (debugging with one ELF but running a different one; mismatched build-ids).
- For C++ exceptions: missing runtime pieces (for example `__gxx_personality_v0` not resolved, or the wrong unwinder/libgcc_s on the target).

If the problem is “backtrace is garbage” rather than exceptions specifically, also consider building with frame pointers (for example `-fno-omit-frame-pointer`) and validating that your unwinder matches the format your toolchain emits (`.eh_frame` vs SFrame vs target-specific unwind tables).

Inspect loader and shared-library problems

Crashes very early in process startup (before `main`) are often due to dynamic loader configuration rather than “code bugs”.

```
llvm-readelf -l ./app    # interpreter (PT_INTERP) and program headers
llvm-readelf -d ./app    # NEEDED, RPATH/RUNPATH
ldd ./app           # what the system thinks will load (when available)
```

For glibc-based systems, `LD_DEBUG` can explain loader decisions:

```
LD_DEBUG=libs,bindings ./app
```

Target ABI / relocations / GOT-PLT debugging notes

When debugging runtime crashes that involve dynamic linking, relocation application, or address materialization sequences, it helps to look at:

- Relocations: `llvm-readelf -r ./app` and `llvm-readelf –dyn-relocations ./app`.
- Dynamic table: `llvm-readelf -d ./app` (`NEEDED`, `RPATH`/`RUNPATH`, flags).
- Program headers: `llvm-readelf -l ./app` (`PT_LOAD` flags/alignment, `PT_GNU_RELRO`).
- GOT/PLT-related sections and their contents:

  ```
  llvm-readelf -S ./app | grep -E \"\\.plt|\\.got|\\.rela(\\.plt|\\.dyn)|\\.rel(\\.plt|\\.dyn)\"
  llvm-readelf -x .got ./app        2>/dev/null | less
  llvm-readelf -x .got.plt ./app    2>/dev/null | less
  llvm-readelf -x .plt ./app        2>/dev/null | less
  ```

  Correlate entries with disassembly + relocations:

  ```
  llvm-objdump -dr --no-show-raw-insn ./app | less
  llvm-readelf -r ./app | less
  ```

External ABI / ELF references (authoritative relocation tables)

When you need a definitive answer for relocation semantics, PLT/GOT conventions, TLS models, or calling convention/stack rules, prefer the architecture psABI/ABI documents (rather than reverse-engineering from a tool implementation).

Useful starting points:

- Generic ELF / System V ABI:
  - Linux Foundation reference index: ELF and ABI Standards
  - System V gABI Edition 4.1: gabi41.pdf
  - ELF Object File Format (modern publication of the ELF chapters): Xinuos ELF spec (HTML)
- x86_64 System V psABI:
  - psABI PDF: x86_64-SysV-psABI.pdf
  - Sources/repo: x86-64 psABI (GitLab)
- Arm AArch32 / AArch64:
  - Official Arm ABI repository (AAELF32/AAELF64, PCS, EHABI, etc): ARM-software/abi-aa
  - AAELF (Arm ELF, AArch32): IHI0044E_aaelf.pdf
  - AAELF64 (AArch64 ELF): IHI0056G_2020Q2_aaelf64.pdf
- RISC-V psABI:
  - Spec site: riscv-elf-psabi-doc
  - Sources/releases: riscv-elf-psabi-doc (GitHub)
- Qualcomm Hexagon:
  - ABI user guide (includes Hexagon-specific ELF/relocations): Qualcomm Hexagon ABI User Guide (PDF)

Architecture-specific relocation names are target-defined; a quick way to see what you are dealing with is the relocation inventory command in this guide. The following are *common* relocation families you may see during runtime triage:

- **x86_64 ABI**: look for `R_X86_64_*` (for example: `*_RELATIVE`, `*_JUMP_SLOT`, `*_GLOB_DAT`, `*_PC32`, `*_PLT32`, `*_GOTPCREL*`).
- **AArch64 ABI**: look for `R_AARCH64_*` (for example: `*_CALL26`, `*_JUMP26`, `*_ADR_PREL_PG_HI21`, `*_ADD_ABS_LO12_NC`, `*_RELATIVE`, `*_JUMP_SLOT`, `*_GLOB_DAT`).
- **ARM ABI (arm/thumb)**: look for `R_ARM_*` (for example: `*_CALL`, `*_JUMP24`, `*_THM_CALL`, `*_THM_JUMP24`) and, on EHABI platforms, unwind tables like `.ARM.exidx`/`.ARM.extab`.
- **RISC-V 32/64**: look for `R_RISCV_*` (for example: `*_PCREL_HI20`, `*_PCREL_LO12_*`, `*_CALL*`, `*_JAL`, plus dynamic relocations like `*_RELATIVE`, `*_JUMP_SLOT`, `*_GLOB_DAT`).
- **Hexagon ABI**: look for `R_HEX_*` relocations; use `llvm-readelf -r` and disassembly to connect the relocation type to the instruction sequence.

ARM-specific: verifying veneers/thunks

On ARM/Thumb, the linker may need to create veneers/thunks (branch islands) when branches cannot reach their targets or when interworking is required.

When a crash looks like a bad branch target or a call landing in the wrong mode:

- Enable and inspect trampoline/thunk diagnostics and maps: `–trampoline-map` (plus the usual text map file).
- Disassemble around the call site and look for a veneer sequence and its relocation(s): `llvm-objdump -dr –start-address=… –stop-address=…`.
- Confirm the callee symbol type and interworking expectations in the symbol table (`llvm-readelf -s –extra-sym-info`).

Minimize runtime failures with A/B experiments

When a runtime crash is hard to reason about, treat it like a minimization problem: change one knob at a time until the failure becomes deterministic, then pinpoint which library/object/relocation pattern is responsible.

Dynamic-linking knobs that often turn “mystery crashes” into actionable data:

- Force eager binding (converts lazy-PLT crashes into startup failures):

  ```
  LD_BIND_NOW=1 ./app
  ```

  Or make it a link-time property of the binary: `-Wl,-z,now`.
- Strengthen shared-library resolution rules (catch unresolved imports sooner): `-Wl,-z,defs` (or `-Wl,–no-undefined` depending on toolchain).
- Toggle dependency pruning and interposition-related behavior: `-Wl,–as-needed` / `-Wl,–no-as-needed` and (when applicable) `-Wl,-Bsymbolic` / `-Wl,-Bsymbolic-functions`.
- Toggle RELRO (affects which GOT/relocation targets become read-only at runtime): `-Wl,-z,relro` and `-Wl,-z,norelro`.

Use these knobs together with map files and relocation inspection (see below) to connect “crash address” -> “instruction” -> “relocation” -> “symbol” -> “DSO that provides it”.

Identify which shared library caused a crash (swap-and-isolate)

If the crash only reproduces with a particular shared library present, isolate it by swapping one dependency at a time:

1. Confirm what actually loads:

   ```
   ldd ./app
   llvm-readelf -d ./app | less
   ```
2. Swap the suspected DSO:
   - Use `LD_LIBRARY_PATH` to point at an alternate build directory.
   - Use `LD_PRELOAD` to force a specific `.so` (for interposition or to override a weak/indirect dependency).
   - Rename/move one dependency to force a load failure; if the crash disappears when the DSO is absent (or replaced), that narrows the search quickly.
3. Ensure you are swapping *compatible* binaries:
   - Same target triple/ABI, same libc/loader expectations.
   - Prefer swapping libraries built by the same toolchain revision first (compiler + assembler + linker + runtimes).
   - Verify build-ids match what you intend to test:

     ```
     llvm-readelf -n ./app | less
     llvm-readelf -n ./libfoo.so | less
     ```

If you suspect a linker-specific issue in a shared library, rebuild just that library with an alternate linker (GNU ld / gold / lld, depending on target support) and re-run the A/B test. If the crash tracks the linker choice with the same sources/flags, it is strong evidence of a link-time layout/relocation problem rather than a pure runtime logic bug.

Switching toolchains/compilers to bisect regressions

If the failure appeared after a toolchain update, a fast way to narrow root cause is to bisect *one dimension at a time*:

- Swap compiler only (keep assembler/linker constant) to see if codegen changes are responsible.
- Swap linker only (keep compiler constant) to see if layout/relocation handling is responsible.
- Swap runtime libraries (libc, libstdc++/libc++, libgcc_s/libunwind) to catch ABI or unwinder differences.

Use build-ids, map files, and the relocation inventory command to keep these experiments grounded in “what changed” rather than “what you think changed”.

Write small regression tests (symbols/relocs/dynamic flags)

When you can reproduce a runtime failure, try to turn it into a small *link-time observable* property so it can be tested without running on a target.

In this repo, most linker tests are lit tests (`*.test`) that:

- compile tiny C/asm inputs,
- run `%link` (eld) with specific flags, and
- verify ELF properties using `%readelf` (`llvm-readelf`), `llvm-readobj`, or `%objdump` + `FileCheck`.

Examples of properties that correlate strongly with runtime behavior:

- Symbol kind/binding/visibility (from `llvm-readelf -s –extra-sym-info` / `nm`).
- Relocation types and placement (from `llvm-readelf -r` / `llvm-readobj -r`).
- Dynamic linking flags and segments (`llvm-readelf -d` for `BIND_NOW` / `FLAGS_1`, and `llvm-readelf -l` for `PT_GNU_RELRO` and `p_align`).

If you need a starting point, see `test/Templates/ExampleOfMyLitTest.test` and existing tests under `test/lld/ELF` that check relocations and flags like `-z now` via `%readelf`/`%objdump`.

Illustrative lit pattern (dynamic flags + RELRO):

```
# RUN: %link %linkopts -shared -z now -z relro %t.o -o %t.so
# RUN: %readelf -d %t.so | FileCheck %s --check-prefix=DYN
# RUN: %readelf -l %t.so | FileCheck %s --check-prefix=PHDR
# DYN: BIND_NOW
# PHDR: GNU_RELRO
```

Other useful inspection tools

- Address/section correlation: `llvm-nm -n`, `nm -n`, `llvm-objdump -d`, `objdump -d`, and the ELD map file.
- ELF metadata: `llvm-readelf -h -l -S -s -n` (build-id in `llvm-readelf -n`).
- Relocation inventory across a build tree (quickly see which relocation types are present in objects/archives):

  ```
  find . -name \"*.o\" -o -name \"*.a\" | xargs llvm-readelf -r | awk '{print $3}' | sort -u | grep R_
  ```

  More robust (handles spaces in paths and avoids `find` precedence traps):

  ```
  find . \\( -name \"*.o\" -o -name \"*.a\" \\) -print0 \\
    | xargs -0 llvm-readelf -r \\
    | awk '{print $3}' \\
    | sort -u \\
    | grep R_
  ```

- Inspect relocations around a crash site (when investigating wrong codegen, bad PLT/GOT usage, or runtime loader fixups):

  ```
  # Disassemble with relocations shown (pick one):
  llvm-objdump -dr --no-show-raw-insn ./app | less
  objdump     -dr --no-show-raw-insn ./app | less

  # Narrow to a region around an address (adjust for PIE/ASLR first):
  llvm-objdump -dr --start-address=0xSTART --stop-address=0xSTOP ./app
  ```

- Segment properties / page alignment / RELRO (loader-relevant):

  ```
  llvm-readelf -l ./app    # program headers: p_flags, p_align, PT_LOAD, PT_GNU_RELRO, PT_INTERP
  llvm-readelf -d ./app    # dynamic tags: DT_BIND_NOW, (FLAGS / FLAGS_1), RPATH/RUNPATH
  ```

- System-call tracing: `strace -f` (and `ltrace` when applicable).
- Memory corruption: ASan/UBSan/TSan builds; `valgrind` when supported.

Shared Library Versioning

Overview

Shared library versioning is a mechanism used in operating systems to manage different versions of dynamic libraries (.so files on Unix-like systems). It ensures that applications link to the correct version of a library while maintaining compatibility.

Key Concepts

SONAME

- The SONAME (Shared Object Name) is an identifier embedded in a shared library that represents its ABI (Application Binary Interface) version.
- Applications link against the SONAME, not the actual filename, allowing upgrades without breaking compatibility.

Semantic Versioning

- Libraries often follow semantic versioning: *MAJOR.MINOR.PATCH*.
- **MAJOR**: Changes that break backward compatibility.
- **MINOR**: Backward-compatible feature additions.
- **PATCH**: Backward-compatible bug fixes.

ABI vs API

- **API (Application Programming Interface)**: The set of functions and symbols exposed by the library.
- **ABI (Application Binary Interface)**: The compiled interface, including calling conventions, data types, and memory layout.
- ABI stability is crucial for shared libraries because applications depend on binary compatibility.

Backward Compatibility

- Libraries should maintain backward compatibility within the same major version.
- Breaking ABI changes require incrementing the major version and updating the SONAME.

Best Practices

- Use SONAME to manage ABI versions.
- Avoid breaking ABI unless necessary.
- **Provide symbolic links:**
  - *libexample.so* → *libexample.so.1* → *libexample.so.1.2.3*
- Document changes clearly.

Nuances

- Different distributions may package libraries differently.
- Some systems use *ldconfig* to manage symbolic links.
- Applications may fail if the expected SONAME is missing, even if the actual library file exists.

Examples

SONAME Usage Example

When building a shared library:

```
clang -shared -Wl,-soname,libexample.so.1 -o libexample.so.1.2.3 example.o
```

Resulting files:

```
libexample.so.1.2.3   # Actual library file
libexample.so.1       # SONAME symlink (ABI version)
libexample.so         # Generic symlink for development
```

Applications link against *libexample.so.1*, not the full filename.

Semantic Versioning Example

- libmath.so.1.0.0 → Initial release
- libmath.so.1.1.0 → Adds new functions (backward-compatible)
- libmath.so.2.0.0 → Changes function signatures (breaks ABI)

When moving from 1.x.x to 2.x.x, update SONAME to *libmath.so.2*.

ABI vs API Example

- **API Change (safe)**: Adding a new function *add_matrix()* does not break existing binaries.
- **ABI Break (unsafe)**: Changing *struct Matrix { int rows; int cols; }* to include a new field changes memory layout, breaking existing binaries.

Symbolic Link Structure

After installation:

```
/usr/lib/
    libexample.so -> libexample.so.1
    libexample.so.1 -> libexample.so.1.2.3
    libexample.so.1.2.3
```

ASCII Diagram:

```
libexample.so → libexample.so.1 → libexample.so.1.2.3
```

Nuances

- Different Linux distros may package libraries differently.
- If SONAME is missing, apps fail with:

```
error while loading shared libraries: libexample.so.1: cannot open shared object file
```

GNU ELF Symbol Versioning

Note

Support for symbol versioning is planned for a future release. Stay tuned for updates

This document continues to show a minimal end-to-end demonstration of GNU ELF symbol versioning, then documents:

1. What the `.symver` / `__attribute__((symver))` mechanisms do.
2. What a GNU ld *version script* does.
3. Which ELF sections the linker/loader create/use for symbol versioning.

The example exports two versions of the same symbol and preserves backward compatibility across library releases.

A. Minimal Working Example (C + GNU ld)

Goal

Export two versions of an API symbol (`foo`), keeping ABI compatibility for old applications while making a new implementation the default for new applications.

Example files

`demo_v1.c` — first release of the library:

```
// v1 of the library: exports foo()
#include <stdio.h>

void foo(void) { puts("foo v1"); }
```

`demo_v1.map` — version script for first release:

```
DEMO_1 {
  global: foo;
  local:  *;
};
```

This assigns `foo` to version node `DEMO_1` and hides everything else.

`demo_v2.c` — second release (keep old `foo` and add a new default `foo`):

```
#include <stdio.h>

/* Approach A: classic assembler directive (works with GCC & Clang) */
__asm__(".symver foo_v1,foo@DEMO_1");    // non-default (old) foo
__asm__(".symver foo_v2,foo@@DEMO_2");   // default (new) foo

void foo_v1(void) { puts("foo v1"); }
void foo_v2(void) { puts("foo v2 (default)"); }

/* New API symbol in v2 */
void bar(void) { puts("bar v2"); }
```

demo_v2.map — version script for second release:

```
DEMO_1 {
  global: foo;
  local:  *;
};

DEMO_2 {
  global: foo; bar;
} DEMO_1;  /* DEMO_2 inherits from DEMO_1 */
```

`main_old.c` — an “old” app built against v1:

```
void foo(void);
int main(void) { foo(); }
```

`main_new.c` — a “new” app built against v2:

```
void foo(void);
void bar(void);
int main(void) { foo(); bar(); }
```

`Makefile`:

```
CC?=clang
CFLAGS=-fPIC -O2 -Wall -Wextra

all: stage1 stage2 inspect

stage1: libdemo.so.1.0 p_old

libdemo.so.1.0: demo_v1.c demo_v1.map
    $(CC) $(CFLAGS) -shared -Wl,-soname,libdemo.so.1 \
        -Wl,--version-script=demo_v1.map -o $@ demo_v1.c
    ln -sf $@ libdemo.so

p_old: main_old.c libdemo.so
    $(CC) -Wl,-rpath,'$$ORIGIN' -L. -o $@ main_old.c -ldemo

stage2: libdemo.so.2.0 p_new

libdemo.so.2.0: demo_v2.c demo_v2.map
    $(CC) $(CFLAGS) -shared -Wl,-soname,libdemo.so.1 \
        -Wl,--version-script=demo_v2.map -o $@ demo_v2.c
    ln -sf $@ libdemo.so

p_new: main_new.c libdemo.so
    $(CC) -Wl,-rpath,'$$ORIGIN' -L. -o $@ main_new.c -ldemo

inspect:
    @echo "==> Exported (with versions) in libdemo.so"
    @nm -D --with-symbol-versions libdemo.so | egrep ' foo| bar' || true
    @echo; echo "==> Version sections in libdemo.so"
    @readelf -W --version-info libdemo.so | sed -n '1,120p'

clean:
    rm -f p_old p_new libdemo.so libdemo.so.* *.o
```

Build & Run

```
# Stage 1: build first library + "old" app
make stage1
LD_LIBRARY_PATH=. ./p_old
# -> prints: foo v1

# Stage 2: replace lib with v2 (keeps SONAME), rebuild "new" app
make stage2
LD_LIBRARY_PATH=. ./p_old
# -> still prints: foo v1  (old app bound to DEMO_1)
LD_LIBRARY_PATH=. ./p_new
# -> prints: "foo v2 (default)" and "bar v2"
```

Inspecting the Result

See exported symbols and their versions:

```
nm -D --with-symbol-versions libdemo.so | egrep ' foo| bar'
# ... foo@DEMO_1
# ... foo@@DEMO_2
# ... bar@@DEMO_2
```

@@ marks the *default* version; @ marks a non-default (older) version.

See version metadata sections:

```
readelf -W --version-info libdemo.so
```

You should see:

- Version definitions (from `.gnu.version_d`).
- Per-symbol version indices in `.gnu.version`.
- (In executables) Version needs in `.gnu.version_r`.

Optional runtime trace from the dynamic linker:

```
LD_DEBUG=versions LD_LIBRARY_PATH=. ./p_new 2>&1 | grep -E 'checking for version|needed'
```

B. What `.symver` / `__attribute__((symver))` Do

- `.symver` (assembler directive) - Binds a specific implementation symbol to a public name *and* a version node. - Syntax: `.symver <impl>, <public>@<NODE>` or `.symver <impl>, <public>@@<NODE>`. - @@ marks the default implementation selected by new linkers. - The version node must exist in the version script when you build the DSO.
- `__attribute__((symver("…")))` - Front-end attribute that makes compiler emit the corresponding `.symver`. - Handy with LTO because it attaches at the language level.

Exactly one default definition should exist for a symbol (the one with @@).

C. What a Version Script Does

A GNU ld *version script* (use with `-Wl,–version-script=…`):

1. Declares **version nodes** and binds **symbols** to those nodes.
2. Controls visibility (`global:` exported; `local:` hidden).
3. Supports **inheritance** between nodes to evolve the ABI without bumping the SONAME.
4. For C++, you can match demangled names with an `extern "C++" { … }` block.

If you want to tag all currently **exported** symbols with one version without changing visibility, this works:

```
V1 { *; };
```

This does not force hidden symbols to become exported; it only tags whatever is already exported.

D. ELF Sections Used by Symbol Versioning

When you link with versioned symbols, the following GNU extension sections (referenced by dynamic tags) are produced/used:

- `.gnu.version` (*SHT_GNU_versym*, referenced by `DT_VERSYM`) - A table parallel to `.dynsym` that stores a 16-bit version index per symbol.
- `.gnu.version_d` (*SHT_GNU_verdef*, referenced by `DT_VERDEF`/`DT_VERDEFNUM`) - Version **definitions** provided by this DSO (node names, indices, hashes).
- `.gnu.version_r` (*SHT_GNU_verneed*, referenced by `DT_VERNEED`/`DT_VERNEEDNUM`) - Version **requirements** (what this module needs from its dependencies).

`readelf -W –version-info` summarizes these sections so you can verify bindings and defaults.

E. Practical Notes & Common Pitfalls

- C++ name mangling: - If you version C++ functions with `.symver`, use the **mangled** name. - In version scripts you can use demangled names inside `extern “C++” { … }`.
- Default vs. non-default: - Default shows as @@; older variants show as @. - Newly linked apps bind to the default; previously linked apps keep using their recorded version.
- Diagnostics: - `nm -D –with-symbol-versions`: quick view of `foo@@V2` / `foo@V1`. - `readelf -W –version-info`: detailed view of `.gnu.version*` sections. - `LD_DEBUG=versions`: watch the dynamic linker’s version checks at runtime.

F. One-Page Cheat Sheet

- `.symver` / `__attribute__((symver))`: - Attach a version to a symbol definition; exactly one default (@@) per symbol. - Ensure the version node exists in the link-time version script.
- Version script: - Define version nodes, bind symbols, control visibility, and express inheritance. - Can version all currently exported symbols via `V1 { *; };`.
- ELF sections: - `.gnu.version` — per-dynamic-symbol version indices (`DT_VERSYM`). - `.gnu.version_d` — version definitions (`DT_VERDEF*`). - `.gnu.version_r` — version requirements (`DT_VERNEED*`).

Appendix: Quick Commands

```
# Build first version and old app
make stage1
LD_LIBRARY_PATH=. ./p_old

# Upgrade library, build new app
make stage2
LD_LIBRARY_PATH=. ./p_old
LD_LIBRARY_PATH=. ./p_new

# Inspect exports and versions
nm -D --with-symbol-versions libdemo.so | egrep ' foo| bar'
readelf -W --version-info libdemo.so

# Trace loader checks
```

Image Structure and Generation

Processor-specific flags

Processor-specific flags are stored in the `e_flags` field of the ELF header. `eld` will set these flags according to the emulation specified or inferred from its inputs. The following are special cases for the `e_flags` field:

1. A lone empty object file:

   ```
   touch empty.o
   ld.eld empty.o -m<emulation>
   llvm-readelf -h a.out
   ```

   The output file will have flags set according to the emulation specified.

2. A symbol definition file:

   ```
   ld.eld -m<emulation> symbols.def
   ```

   The output file will have flags set according to the emulation specified.

3. A lone binary file:

   ```
   ld.eld -m<emulation> --format=binary foo.bin
   llvm-readelf -h a.out
   ```

   Flags are always 0 regardless of the emulation specified.

Linker Map Files

A map file provides detailed information about the link and helps analyze output images, debug build issues, and better understand linker decisions. It contains detailed information for input files, linker scripts, memory regions, image memory layout, and so much more. Effectively using map files can significantly reduce the time spent analyzing and debugging builds.

Use **`-Map`** to generate map files for your builds.

Map File Styles

Map files are available in three distinct styles:

1. Text
2. YAML
3. Binary

Use option **`-MapStyle`** to specify the map file style. Text map file is the default map style.

Navigating Text Map File

Map file contents

Section	Description
Tools Version, and System and Link features	Information about tools version, and system and link features.
Link Stats	Quick overview of key link information
Archive Member Information	Specifies which archive members were included and why.
Allocating Common Symbols	Lists the common symbols which got allocated
Linker Script and Memory Map	Specifies input object files
Build statistics	
Linker scripts used	Specifies the linker scripts used by the link.
Linker plugin information	Details plugin information.
Output Section and Layout	Delineates output image layout

Let's explore each of these sections in detail!

Tools Version, and System and Link Features

As the title suggests, this consists information about tools version, and system and link features. An example of this section is shown below:

```
# Linker from QuIC LLVM Hexagon Clang version Version 19.0-alpha2 Engineering Release: hexagon-clang-190-229
# Linker based on LLVM version: 19.0
# Linker: d8a255719548f774e7e73cd97959ca04fd780b54
# LLVM: 9ab18b8e7547380db4fb83b2ada1d7704876b334
# Notable linker command/script options:
# CPU Architecture Version:
# Target triple environment for the link: unknown
# Maximum GP size: 8
# Link type: Static
# ABI Page Size: 0x1000
# CommandLine : hexagon-link -o main.elf main.o libs/libfoo.a libs/libbar.a -Map main.map.txt
```

Link Stats

'Link Stats' provides a quick overview of key link information. **Link stats are not displayed if their value is 0.**

An example of 'Link Stats' is shown below:

```
# LinkStats Begin
# ObjectFiles : 2
# LinkerScripts : 2
```

```
# ThreadCount : 64
# NumInputSections : 37
# ZeroSizedSections : 28
# Total Symbols: 6 { Global: 3, Local: 2, Weak: 0, Absolute: 3, Hidden: 1, Protected: 0 }
# Discarded Symbols: 1 { Global: 1, Local: 0, Weak: 0, Absolute: 0, Hidden: 1, Protected: 0 }
# NumLinkerScriptRules : 2
# NumOutputSections : 7
# NumOrphans : 5
# NoRuleMatches : 3
# NumSegments : 1
# OutputFileSize : 4920 bytes
# LinkStats End
```

There might be more or less link stats than what is shown here depending on your tools version. Remember that only non-zero stats are displayed.

Archive Member Information

'Archive Member Information' tells which archive members were included in the link and due to which symbol. Archive members are only included in the link if they satisfy an undefined symbol reference. The format of each entry of 'Archive Member Information' is:

```
<ArchiveFile> (<ArchiveMember>)
     <CompilationUnitReferringArchiveMember> (<ReferencedSymbol>)
```

The below example shows that archive members 'libs/libfoo.a(Foo.o)' and 'libs/libbar.a(Bar.o)' are included in the link because 'main.o' referenced symbols 'foo' and 'bar' from the archive members.

```
Archive member included because of file (symbol)
libs/libfoo.a(Foo.o)
                main.o (foo)
libs/libbar.a(Bar.o)
                main.o (bar)
```

Allocating Common Symbols

- 'Allocating Common Symbols' list the common symbols that were allocated by the linker. Each entry in the common symbols section contains the following items:
    - The name of the symbol
    - The size of the memory area allocated for the symbol
    - The full pathname of the archive file accessed by the linker
    - The name of the archived object file that defines the symbol (in parentheses)

In the following example, the __eh_nodes symbol has size of **0x4** and is defined in the the object file: **xregister.o**

```
1 Common symbol       size    file
2
3 __eh_nodes  0x4     $PATH/../target/hexagon/lib/v65/G0/libc.a(xregister.o)
```

Linker Script and Memory Map Section

- The linker script section of a link map lists the complete linker script that is specified for the link.

> ### Note
>
> Linker scripts are optional. If a script is not specified on the linker command line, the link map does not include a linker script.

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

```
1 Linker Script and memory map
2 LOAD $PATH/../target/hexagon/lib/v65/G0/crt0_standalone.o[hexagonv65]
3 LOAD target/main.o[hexagonv65]
4 START GROUP
5 LOAD $PATH/../target/hexagon/lib/v65/G0/libstandalone.a(_exit.o)[hexagonv65]
6 ...
7 LOAD $PATH/../target/hexagon/lib/v65/G0/libgcc.a(vmemcpy_h.o)[hexagonv65]
8 END GROUP
9 LOAD $PATH/../target/hexagon/lib/v65/G0/fini.o[hexagonv65]
```

Linker Scripts Used

'Linker Scripts Used' specifies the linker scripts used by the link.

```
1 Linker scripts used (including INCLUDE command)
2 target/linker.script
```

Todo

Add 'Linker Plugin Information' section docs!

Todo

Add 'Build statistics' section docs!

Output Section and Layout

'Output Section and Layout' describes the image layout. It tells where output sections are placed and what they consist of. It presents information in an elaborate format loosely based on the linker script `SECTIONS` command. This format allows users to easily visualize and understand how `SECTIONS` subcommands, linker script expressions and assignment to the location counter affect the output layout.

'Output Section and Layout' contains detailed information about output sections, input section descriptions (commonly called rules), input sections, symbols, padding and linker script expressions. It is the most complex part of the map file. Let's explore its structure to better understand its content.

Simplified 'Output Section and Layout' Structure

We will first see its simplified structure. Padding and linker script expressions are missing from this simplified structure. As we move forward in the documentation, these missing components will be gradually covered to make the structure complete.

<OutputSectionAndLayout>

```
# Output Section and Layout
<OutputSection_1>
<OutputSection_2>
...
<OutputSection_N>
```

Each <OutputSection> describes one output section. Let's see what it consists of.

<OutputSection>

<OutputSection> is represented as:

```
<OutputSectionName> <size> <VMA> # Offset: <offset>, LMA: <LMA>, Alignment: <alignment>, Flags: <SectionFlags>, Type: <SectionType>, Segments: <segments>
# INSERT AFTER|BEFORE <anchor-section> # <script:line>
<Rule_1>
<Rule_2>
...
<Rule_N>
```

The INSERT line is present only when the output section was positioned using `INSERT AFTER` or `INSERT BEFORE` in a linker script. The trailing script context shows the originating linker script location.

<OutputSection> tells output section properties and contains a list of `Rule` entries. Placement of rules conveys the placement of the rule contents in the output image layout. The contents of *Rule_1* are placed before the contents of *Rule_2* and so on. The contents of a rule is the merged content of all the sections that were matched to the rule. Now, let's see what does <Rule> consist of.

<Rule>

<Rule> (input section description) is represented as:

```
<InputSectionDesc> #Rule <RuleNumber>, <InputFile>
<InputSection_1>
<InputSection_2>
...
<InputSection_N>
```

`RuleNumber` makes it easier to refer to a particular input section description. <Rule> contains a list of <InputSection> that got matched to this rule.

<InputSection>

<InputSection> is typically represented as:

```
<InputSectionName> <VMA> <size> <InputFile> #<SectionType>,<SectionFlags>,<SectionAlignment>
<Symbol_1>
<Symbol_2>
...
<Symbol_N>
```

However, garbage-collected input sections are prefixed with '#', are annotated with `<GC>` and do not have `<VMA>` component. The <InputSection> for garbage-collected input sections is represented as:

```
# <InputSectionName> <GC> <VMA> <InputFile> #<SectionType>,<SectionFlags>,<SectionAlignment>
<Symbol_1>
<Symbol_2>
...
<Symbol_N>
```

<Symbol>

<Symbol> is typically represented as:

```
<VMA> <SymbolName>
```

However, garbage-collected symbols are prefixed with '#', and do not have `VMA` component. The <Symbol> for garbage-collected symbol is represented as:

```
# <SymbolName>
```

Example

We have covered the simplified <OutputSectionAndLayout> structure. Now, let's see a practical example.

Foo.c

```
int foo() { return 1; }
```

Bar.c

```
int bar() { return 3; }
```

script.t

```
SECTIONS {
  FOO : { *(*foo*) }
  BAR : { *(*bar*) }
}
```

Generate the map-file:

```
hexagon-clang -o Foo.o Foo.c -c -ffunction-sections
hexagon-clang -o Bar.o Bar.c -c -ffunction-sections
hexagon-link -o FooBar.elf Foo.o Bar.o -T script.t -Map FooBar.map.txt
```

Output Section and Layout excerpt from the map-file

```
# Output Section and Layout
FOO     0x0     0xc # Offset: 0x1000, LMA: 0x0, Alignment: 0x10, Flags: SHF_ALLOC|SHF_EXECINSTR, Type: SHT_PROGBITS
*(*foo*) #Rule 1, script.t [1:0]
.text.foo       0x0     0xc     Foo.o   #SHT_PROGBITS,SHF_ALLOC|SHF_EXECINSTR,16
        0x0             foo
*(FOO) #Rule 2, Internal-LinkerScript (Implicit rule inserted by Linker) [0:0]

BAR     0x10    0xc # Offset: 0x1010, LMA: 0x10, Alignment: 0x10, Flags: SHF_ALLOC|SHF_EXECINSTR, Type: SHT_PROGBITS
*(*bar*) #Rule 3, script.t [1:0]
.text.bar       0x10    0xc     Bar.o   #SHT_PROGBITS,SHF_ALLOC|SHF_EXECINSTR,16
        0x10            bar
*(BAR) #Rule 4, Internal-LinkerScript (Implicit rule inserted by Linker) [0:0]
```

We can see precisely how output sections are laid in the final layout and what they are composed of. For instance, the FOO output section contains the `.text.foo` input section because the `.text.foo` matches the rule `*(*foo*)`. Under the input section information, we can also see where each symbol of the section is placed in the output layout.

Other <OutputSectionAndLayout> substructures

<LinkerScriptExpression>

<LinkerScriptExpression> entries are interspersed in <OutputSectionAndLayout>, <OutputSection> and <Rule> and indicates their placement in the linker script. <LinkerScriptExpression> contains the linker script expression annotated with the values.

For example, if the linker script contains:

```
SECTIONS {
  FOO : {
    v = . + 0x5;
    *(*foo*)
  }
}
```

Then the map-file will contain:

```
FOO     0x0     0x100c # Offset: 0x1000, LMA: 0x0, Alignment: 0x10, Flags: SHF_ALLOC|SHF_EXECINSTR, Type: SHT_PROGBITS, Segments : [ FOO ]
        v(0x5) = .(0x0) + 0x5; # v = . + 0x5; script.t
*(*foo*) #Rule 1, script.t [1:0]
.text.foo       0x0     0xc     1.eld.o         #SHT_PROGBITS,SHF_ALLOC|SHF_EXECINSTR,16
        0x0             foo
```

Note the `v(0x5) = .(0x0) + 0x5;` line under FOO output section.

<Padding>

<Padding> describes a padding in the output image layout. Paddings are either requested explicitly by the user or are added automatically by the linker to satisfy alignment constraints. Like <LinkerScriptExpression>, <padding> is also interspersed in <OutputSectionAndLayout>, <OutputSection> and <Rule>. The position of <padding> entry in the map-file conveys the position of padding in the memory layout.

Let's see an example of explicitly requested padding.

If the linker script contains:

```
SECTIONS {
 FOO : {
```

```
    . = . + 0x10;
    *(*foo*)
  }
 }
```

Then the map-file will contain:

```
# Output Section and Layout

FOO     0x0     0x1c # Offset: 0x1000, LMA: 0x0, Alignment: 0x10, Flags: SHF_ALLOC|SHF_EXECINSTR, Type: SHT_PROGBITS
PADDING 0x0     0x10    0x0
        .(0x10) = .(0x0) + 0x10; # . = . + 0x10; script.t
*(*foo*) #Rule 1, script.t [1:0]
.text.foo       0x10    0xc     Foo.o   #SHT_PROGBITS,SHF_ALLOC|SHF_EXECINSTR,16
        0x10            foo
*(FOO) #Rule 2, Internal-LinkerScript (Implicit rule inserted by Linker) [0:0]
```

Please note the `PADDING` details under the `FOO` output section.

Link map in YAML format

- If you use the MapStyle option to specify YAML-style output for the map file, the linker generates a YAML file instead of the text file that shows the memory map for the program.
- To derive statistics from the YAML map file produced by the linker, use the **YAMLMapParser.py** tool and include any of the following options:
    - **–info=architecture** – List the architecture
    - **sizes** – List the code and data sizes for the objects in the image
    - **summarysizes** – List the code and data sizes of the image
    - **totals** – List the total size of all objects in the image
    - **unused** – List the sections eliminated from the image
    - **unusedsymbols** – List the symbols eliminated from the image
    - **–map** – Display the memory map of the images
    - **–xref** – List the cross references between input sections
    - **list** – Redirect the output to a file

The following table describes the sections of a YAML file

Section	Description
Header	Top level information of the program that was built
Version	Information Tools version information
Archive Records	Archive files pulled in by the linker
Inputs	Inputs that were used
InputInfo	Inputs that were used and not used
Linker script used	Linker script file that was used
BuildType	Type of build
OutputFile	Name of the output file
EntryAddress	Entry address for the image built
CommandLine	information on how the linker was called
CommandLineDefaults	Various defaults applied in the linker
OutputSections	All output sections

DiscardedComdats	COMDAT C++ section groups that discarded
DiscardedSections	Sections discarded by garbage collection
DiscardedCommonSymbols	Common symbols that were discarded
LoadRegions	Segments that the program loader will load
CrossReferences	Cross-reference table for the program

Image layout

At its core, the linker is responsible for transforming compiled object files into a final image that is ready to run on the device. This process includes resolving symbols, relocating sections, and most importantly, laying out the image in memory.

Image layout refers to the linker's assignment of addresses to code, data and metadata determining their arrangement in memory when the program runs, and to the placement of contents within the final output image.

In this document, we will take a deep dive into the linker's image layout process, examining the factors that influence the layout and the reasoning behind the specific choices the linker makes. Understanding why the image layout looks the way it does is challenging, but it's invaluable for diagnosing and fixing subtle, hard-to-decode layout issues.

But why is the image layout important?

The image layout is critical for correctness, performance and security. Among other things, it ensures alignment compliance, enables cache-friendly placement of code and data, and enhances security by enforcing memory protection policies such as preventing any executable page from having write permissions. In embedded systems, image layout is especially critical due to tight memory constraints, real-time performance needs, and hardware-specific placement requirements.

Before we dive deep into defining and understanding how the linker does image layout, it would be useful to understand the distinction between some of the terminology used in the linker.

Linker terminology

Input Sections

A section holds content that can be code, data and metadata (Example: `.text` , `.data`, `.bss`, `.comment`). The linker may reorder, merge or discard whole sections but it treats each section as an indivisible unit. The section header table describes each section's name, type, virtual address, file offset, alignment and more. **The section header table and the sections are used by the linker during linking.** These come from object files (.o) and libraries (.a) compiled from source code.

Examples:

- .text — code
- .data — initialized data
- .bss — uninitialized data
- .rodata — read-only data

Output sections

These are the merged and organized sections in the final binary, created by the linker.

The linker collects input sections of the same type and merges them into output sections.

You can rely on the default rules and built-in heuristics provided by the linker.

You can also control this mapping by using a linker script.

Segments

A segment is composed of one or more output sections and describes how the program should be loaded into memory at runtime. In ELF, The program header table describes each segment's type, virtual and physical address, file size, memory size, permissions, and alignment requirements. **A loader only looks at the program header table to load an executable into memory.** It does not care about the section header table.

Key Points:

- Segments are containers for output sections
- They define memory permissions (read, write, execute)
- Used by the runtime loader

Segment alignment

Empty segments

- Loadable segments all have segment align set to the page size set at link time
- Non loadable segments have the segment alignment set to the minimum word size of the output ELF file

Non Empty segments

- Loadable non empty segments have the segment alignment set to the page size

- Non loadable segments are set to the maximum section alignment of the containing sections in the segment

Now let's get started with understanding the image layout created by eld.

Image layout

There are two main approaches of defining the image layout:

- Using default behavior
- Using custom linker scripts

Using default behavior

When a linker performs layout without an explicit linker script, it relies on default rules and built-in heuristics provided by the linker implementation.

The linker has a default memory layout that defines:

- The order of sections (e.g., .text, .data, .bss)
- Default starting addresses (e.g., code starts at 0x08000000 for embedded systems or 0x400000 for ELF binaries on Linux)
- Alignment requirements for each section
- Default page alignment
- Whether program headers are loaded or not loaded

linker scripts

> **Note**
>
> When using eld with a custom linker script, all default assumptions and behaviors built into the linker are overridden. The script provides complete control over the memory layout and section placement, effectively replacing the linker's internal defaults.

The linker script primary job is to define the image layout requirements. It achieves this with the `SECTIONS`, `PHDRS`, and the `MEMORY` commands.

- `SECTIONS` command specifies the output section properties and the mapping of input sections to the output sections.
- `PHDRS` specifies which program headers (also known as segments) should be created by the linker. If `PHDRS` is not specified, then the linker creates sensible PHDRs based on some default rules.
- `MEMORY` command specifies the available memory regions. The output sections can then be assigned to particular memory regions. It provides a convenient way of arranging the output sections into memory. See MEMORY for syntax, region attribute matching rules, and LMA/VMA placement details.

Linker Script describes these commands in more detail.

Here we will only highlight, or in some cases reiterate, some of the important and/or subtle layout-related features and behavior.

Output section contents

An output section is composed of input sections. If no `SECTIONS` command is provided, the linker applies sensible default rules to match the input sections to the output sections.

When a `SECTIONS` command is present, the linker script input section description, commonly referred to as linker script rule or just rule, control the input-output section mapping. Input sections that match no rule are called orphans sections. Each orphan section is placed into an output section with the same name; if no such output section exists, then the linker creates a new one.

Image base address (starting address)

The default image base address depend upon the target, the linking type (whether the link is creating an executable or a shared library), and whether the link contains a linker script. The address 0 is used as the starting address if a linker script is present or if the link is creating a shared library.

The default image base value can be overridden by using **`-image-base`** command-line option.

Output section virtual memory address (VMA)

Multiple mechanisms can assign or influence the addresses of output sections. A linker script can specify addresses explicitly; command-line options can set section start addresses; and script directives such as `MEMORY` allows to conveniently arrange sections without hard-coding absolute addresses. Let's examine all these methods, detail their behavior, and clarify their precedence.

Brief review of the methods

1) No address/memory-region specified

If no explicit address or MEMORY region is specified for an output section, it is placed at the current value of the location counter (`.`'). The location counter is initialized to the image base, and is automatically incremented whenever a content is added. For example, if the value of the location counter is 0x1000 before assigning address to `.foo` output section (size 0x200), then `foo` is placed at `0x1000` and the location counter advances to 0x1200.

The location counter can be explicitly incremented with a linker script assignment:

```
. = . + ALIGN(0x8);
```

If an output section has an explicit address (or a memory region), then the location counter is set to the address (or the valid address in the memory region) before placing the output section.

There is an exception to this, orphan sections whose name ends with `@<address>` are placed exactly at the specified address.

2) Explicit linker script VMA address

Linker script can specify an explicit address for an output section:

```
section [address] :
{
  output-section-command
  ...
}
```

Let's see this in action with the help of an example:

```
// 1.c
int foo() { return 1; }

int bar() { return 3; }

// script.t
SECTIONS {
  .foo (0x1000) : { *(.text.foo) }
  .bar (0x2000) { *(.text.bar) }
}
```

The linker assigns the exact addresses as specified in the linker script. `.foo` is assigned the VMA `0x1000` and `bar` is assigned the VMA `0x2000`.

The linker assigns the exact addresses even if the addresses are not *exactly* aligned. For the below example, `.foo` will get VMA assigned to `0x1001` and `.bar` will get VMA assigned to `0x2002` even though they do not satisfy the alignment requirements. In this case, the output section contains alignment padding at the beginning such that the alignment requirements for the actual content (input sections) is satisfied.

```
// 1.c
int foo() { return 1; }

int bar() { return 3; }

// script.t
SECTIONS {
  .foo (0x1001) : { *(.text.foo) }
  .bar : AT(0x2002) { *(.text.bar) }
}
```

However, if `ALIGN` is also specified, then the explicit VMA is aligned to the alignment specified in `ALIGN`.

Todo

Add an example showing explicit VMA + `ALIGN(…)` interaction.

3) Command-line options for setting section addresses

There are various linker command-line options for setting output section VMA: `-Tbss`, `-Tdata`, `-Ttext` and `–section-start`.

When both the linker script and the command line specify an output-section address, the command-line option takes precedence and overrides the script's explicit address.

4) Memory region

Assigning an output section to a MEMORY region places it at the region's next available address, subject to alignment and size constraints.

Let's understand this with the help of an example:

```
    // 1.c
    int a;
    int u = 11;
    int v = 13;
    int foo() { return 1; }
    int bar() { return 3; }
// script.t
MEMORY {
  RAM : ORIGIN = 0x1000, LENGTH = 0x1000
}

SECTIONS {
  .data : { *(.data*) } >RAM
  .foo : { *(.text.foo) } >RAM
  .bss : {
    *(.bss*)
  } >RAM
  .bar : { *(.text.bar) } >RAM
}
```

For the example above, the virtual addresses of the output section are shown below, along with the assumed size and alignment.

- `.data` (size: 0x8; alignment: 0x8): 0x1000
- `.foo` (size: 0x8; alignment: 0x8): 0x1008
- `.bss` (size: 0x4; alignment: 0x4): 0x1010

- `.bar` (size: 0x8; alignment: 0x8): 0x1014

Precedence of virtual address assignment methods

The precedence is:

1. Command-line options for setting section addresses
2. Explicit linker script VMA address
3. Memory region
4. No address/memory-region specified

Segment creation when PHDRS is not specified

eld uses a set of heuristics to decide when to start a new segment. Before detailing those heuristics, we need to note how eld walks the layout:

eld processes output sections in the order specified by the linker script; if no script is provided, then it uses the order in which the linker has created default output sections.

We use the below terms to describe when eld creates a segment:

- Previous output section to refer to the output section that immediately precedes the current one in the traversal order.
- Previous allocatable output section to refer to the nearest preceding allocatable output section.
- Previous LOAD segment to refer to the segment of the nearest preceding allocatable output section.

With this traversal model and the terms in mind, we can now describe the segment-creation heuristics.

A new load segment is created when:

- There is no previous LOAD segment.
- Explicit output section address has been set using linker command-line options such as: `-Ttext` or `–section-start`.
- The segment flags required for the current output section is incompatible with the previous LOAD segment. By default, the segment flags `R` and `RE` are compatible, whereas `RW` is incompatible with `R` and `RW`.

  `–rosegment` and **`–omagic`** influence which sections can be part of the same segment.
- The memory region of the current output section is different than the memory region of the previous allocatable output section.
- The previous output section type is NOBITS and the current output section type is PROGBITS.
- The previous output section virtual memory address is greater than the current output section virtual memory address.
- The VMA difference between the previous allocatable output section is greater than the segment alignment.

Other segment types:

- A `PT_TLS` segment is created when an input file contains `.tdata`/`.tbss`.
- A `PT_DYNAMIC` segment is created when a shared library or a dynamic executable is getting built.
- A `PT_GNU_EH_FRAME` segment is created when the output contains `eh_frame_hdr` section.

Segment creation when PHDRS is specified

When the `PHDRS` is specified in the linker script, then the linker only creates the section specified in the `PHDRS` command. No additional segments are created.

Linker script assignment evaluation

Linker script assignment evaluation can influence the image layout as the location counter value can be modified using a linker script assignment and the location counter controls where the next content would be placed.

eld does not support lazy expression evaluation and forward references in expressions.

In eld, the linker script assignments outside the `SECTIONS` command are evaluated before the linker script assignments inside the `SECTIONS` command. The linker script assignments order is:

- First, all the linker script assignments outside the `SECTIONS` command are evaluated in the specified order.
- Then all the linker script assignments inside the `SECTIONS` command are evaluated in the specified order.

The linker script assignments specified using `–defsym` are considered as outside-`SECTIONS` linker script assignments.

Linker options that affect layout

`–rosegment`

By default, readonly non-executable (`R`) sections such as `.rodata` sections and executable sections (`RX`) such as `.text` can be part of the same segment. When **`–rosegment`** is specified, a different segment is created for readonly non-executable segments.

`–omagic`

If **`–omagic`** is specified, then readonly non-executable (`R`), executable (`RX`), and read-write (`RW`) sections can be part of the same segment. Moreover, the segment alignment is set to the maximum section alignment instead of the page alignment.

`–align-segments`

This option cannot be used with linker scripts. When used, the addresses of segments (both virtual and physical addresses) are aligned to the page boundaries.

`–enable-bss-mixing` and `–disable-bss-conversion`

These options control how the linker treats BSS sections relative to non-BSS sections within the same segment.

`–disable-bss-conversion` controls whether the linker converts BSS to non-BSS when BSS/non-BSS sections are mixed. When this option is passed, BSS remains NOBITS when BSS and Non-BSS are mixed in a segment. This option must be combined with **`–enable-bss-mixing`** if the BSS section will be placed before a non-BSS section because otherwise BSS before non-BSS is an error.

This combination of options is useful for reducing file size of a program. For example, if a program has a layout requirement that a BSS section (let's say `.bss`) must be placed before a non-BSS section (let's say `.data`) in the same segment, and the addresses of `.bss` and `.data` are `0x1000` and `0x2000` respectively, then with the default behavior `.bss` will be converted to PROGBITS and 0s would need to be filled explicitly in the file for this section. This can significantly increase the file size. On the contrary, when `–disable-bss-conversion –enable-bss-mixing`, `.bss` will remain as NOBITS and thus no file size will be consumed by this section.

Please note that with such a layout you may need a custom loader because most standard loaders would not accept a layout where a BSS section is followed by a non-BSS section in the same segment.

> **Note**
>
> These options are currently supported only for ARM, AArch64, and RISCV targets.

Let's understand these options in more detail with the help of an example:

Below is a minimal example showing how these options affect layout when placing both `.bss` and `.data` in the same segment.

bssmix.c:

```
int bssvar;            // goes to .bss (NOBITS)
int data = 1;          // goes to .data (PROGBITS)
int main() { return data + bssvar; }
```

Linker script (script.t):

```
PHDRS {
  A PT_LOAD;
  B PT_LOAD;
}

SECTIONS {
  .text (0x1000) : { *(.text*) } :A
  .data (0x3000) : { *(.data*) } :B
  .bss  (0x2000) : { *(.bss*)  } :B
}
```

Build and link:

```
$ clang -o bssmix.o --target=aarch64-unknown-elf -c bssmix.c -ffunction-sections -fdata-sections

# .bss section is promoted to PROGBITS because .bss is placed before .data in the segment B.
$ ld.eld -o bssmix.out bssmix.o -T script.t

# Error: Mixing BSS and non-BSS sections in segment. non-BSS '.data' is after
# BSS '.bss' in linker script
$ ld.eld -o bssmix.disable_conversion.out bssmix.o -T script.t --disable-bss-conversion

# .bss section is placed before .data in the segment B and BSSness of .bss is preserved,
# that is, it is not promoted to PROGBITS.
$ ld.eld -o bssmix.disable_conversion.out bssmix.o -T script.t --disable-bss-conversion \
    --enable-bss-mixing
```

Advanced image layout control using linker plugins

More finer-grained control over the image layout can be achieved using linker plugins.

- Custom Layout Logic: Plugins can override default section placement logic to optimize for cache locality or hardware-specific constraints.
- Dynamic Behavior: Unlike static scripts, plugins can make layout decisions based on runtime metadata or build-time heuristics.
- Programmatic Layout: Developers can write plugins (e.g., LayoutOptimizer) that programmatically define how sections are arranged, enabling more flexible and performance-tuned binaries

This approach is especially useful in embedded systems where:

- Memory is constrained and fragmented.
- Performance depends on precise placement of code/data.
- Debugging requires reproducible and traceable layouts.

Linker Plugins

Linker plugin framework is a user-friendly solution to add custom link behavior. This allows non-linker developers to tweak and hack linker for the specialized use-cases. It is similar to how clang plugins let you tweak frontend compliation and LLVM passes let you transform the LLVM IR. [[The core idea is to make the tool extendable and hackable by providing the framework]].

But why would you ever need to customize link? Short answer: Embedded development can get really fun at times. Long and more useful answer will be discussed as we move on.

This guide contains all that you need to know about linker plugins, and how to effectively write one yourself.

Overview

- Implemented as a C++ class.
- Facilitates user-defined behavior to crucial parts of the link process.
- Provides finer control over the output image layout than a linker script.
- Allows inspecting symbols, chunks, relocation, section mapping, input and output sections.
- Allows adding and modifying section mapping rules, relocations, chunk attributes, and symbols.
- Plugins can communicate with the linker as well as with each other.
- Easy traceability of plugins using `–trace=plugin` option.
- `PluginADT` provides wrappers for many common linker data types.

Linker Plugins

What and why of linker plugins

Linker plugins make it possible to run user-defined code during the link process. Implemented as a C++ class, they are a programmatical way to add new functionalities and modify default linker behavior.

But why modify default linker behavior? We will now discuss the long answer we promised at the beginning of the linker plugins documentation.

Plugins provide more control and flexibility over the output image layout as compared to the linker scripts. For complicated rules, linker scripts are often very cumbersome to write, sometimes desired rule matching behavior might even be impossible from just linker script. For example, a plugin can move around chunks from one output section to another. Linker plugins also allow the user to alter program layout dynamically which is not possible using linker script.

Plugins add custom behavior to the linker by defining behavior for the hooks that the linker runs in various stages of the link process. These hooks allows plugins to modify the default linker behavior, provide a fine-grained control over the output image layout, add new functionalities to the linker.

This guide will describe all that you need to know about linker plugins, and how to effectively write one yourself. This guide assumes that the reader has a basic understanding of linkers, linker relocations and linker scripts.

Elements of the link process

Before we move further, let's first quickly go over some important elements of the link process. A solid understanding of these elements will help to better understand the link process, and linker's capabilities and limitations. This in turn will help to effectively design and write plugins.

Symbols

Formally, a symbol is a name associated with a value. This value can be an offset within a section, a virtual memory address, or simply an absolute value. Symbols are an integral component of an object file. Among other things, they are used to refer to global variables and functions in a program.

> **Note**
>
> Global variables were specifically mentioned because local variables are created and managed on the stack at run-time and linker is blissfully unaware of them. Beware: local symbols are not the same as local variables!

`eld::plugin::Symbol` represents a linker symbol.

Symbol resolution

For each group of non-local symbols with the same name, symbol resolution selects one of these symbols using well-defined rules. The selected symbol is part of the output image symbol table and is used to resolve symbol references to that name.

Input Section

Each relocatable object module, *M*, has input sections. Input sections contains most of the object file information for the linking view: instructions, data, symbol table, relocation information and so on.

`eld::plugin::Section` represents an input section.

Output Section

Output section contains one or more input sections. The output sections are in turn mapped to a segment. A segment contains one or more output sections, and is used for finally loading the program. Typically, output sections with same *Alloc*, *Exec*, and *Write* permissions are mapped to the same segment.

`eld::plugin::OutputSection` represents an output section.

Chunk

In ELD's terminology, a section is made up of many sub-parts called as chunk or fragment. Plugins can move chunks from one output section to another. Linker scripts do not have this level of control over the output image layout.

`eld::plugin::Chunk` is a handler for a chunk. It can be used to inspect the properties, symbols and content of a chunk.

Relocation

Relocation is the process of connecting symbolic references with symbolic definitions. All symbolic references of the same symbol should be connected to the same definition. Object files contain relocation entries that describes how the relocations should be processed.

`eld::plugin::Use` represent relocations.

Section Mapping

Each input section is either discarded or is mapped to some output section. Linker has some default mapping rules that defines which input section is to be mapped to which output section. Traditionally, linkers allowed users to add their own custom mapping rules using linker script `SECTIONS` command.

Now, we have covered the basic elements of the link process. Let's dive into the linker plugin functionality.

Plugin Types

ELD has different plugin types. Plugin types differ in which hooks they provide. Hooks are pre-defined spots in the link pipeline where a plugin can tap into the linker to customize the link. Plugin taps into the linker by defining the callback function for the hooks. The linker calls these callback functions at the hook sites. For brevity, we will refer callback function for the hook as hook callback function.

There two distinct kinds of plugins: LinkerPlugin plugin and layout plugins. LinkerPlugin plugin is more general and provides hook that covers the entire link process. On the other hand, the layout plugins have specialized hooks for iterating over certain link components (input sections, output sections, …) instead of having general hooks. Different layout plugin types iterate over different link components. For example, `SectionMatcher` plugin type provide hooks to process input sections and `OutputSectionIterator` plugin type provides hooks to process output sections.

There are a total of 6 plugin types. One LinkerPlugin plugin type and 5 layout plugin types. For each plugin type, there is a corresponding plugin class. Plugin authors create a new plugin by inheriting from the corresponding plugin class and implementing the hook callback functions.

The 6 plugin interfaces along with their corresponding C++ class and header file are listed below.

Plugin interface	Plugin interface class	Header file
LinkerPlugin	LinkerPlugin	ELD/PluginAPI/LinkerPlugin.h
SectionMatcher	SectionMatcherPlugin	ELD/PluginAPI/SectionMatcherPlugin.h
SectionIterator	SectionIteratorPlugin	ELD/PluginAPI/SectionIteratorPlugin.h
ControlFileSize	ControlFileSizePlugin	ELD/PluginAPI/ControlFileSizePlugin.h
ControlMemorySize	ControlMemorySizePlugin	ELD/PluginAPI/ControlMemorySizePlugin.h
OutputSectionIterator	OutputSectionIteratorPlugin.h	ELD/PluginAPI/OutputSectionIteratorPlugin.h

Linker Wrapper

Running a linker plugin

There are two methods to run linker plugins:

- Adding a plugin invocation command in a linker script. *(Not supported for :code:'LinkerPlugin' plugins)*
- Specifying plugin configuration file using *–plugin-config* option. *(Recommended way)*

We call these methods as *plugin load specification* because they specify how the linker should load a plugin.

Adding a plugin invocation command in a linker script.

```
<PluginTypeKeyword>("LibraryName", "PluginName" [, "PluginOption"])
```

For output section plugin types, `ControlMemorySizePlugin` and `ControlFileSizePlugin`, users should add the plugin invocation command to the output section description of the output section on which these plugins should run. For example:

```
SECTIONS {
  output_section <PluginTypeKeyword>("LibraryName", "PluginName" [, "PluginOption"]) : {
    *(.text)
  }
}
```

Note

Only one output section plugin can be attached to an output section.

- **PluginTypeKeyword**

  Each plugin type has a corresponding linker script plugin keyword.

Plugin interface type	Linker script keyword
SectionMatcher	PLUGIN_SECTION_MATCHER
SectionIterator	PLUGIN_ITER_SECTIONS
ControlFileSize	PLUGIN_CONTROL_FILESZ
ControlMemorySize	PLUGIN_CONTROL_MEMSZ
OutputSectionIterator	PLUGIN_OUTPUT_SECTION_ITER

- **LibraryName**

  - Name of the dynamic library that contains the plugin for the linker to load.
  - Finds the library in the same search paths as if the library was passed as an input to the linker.
  - Uses the name of the library without the lib prefix on Linux and without the .so/.dll suffix on Linux/Windows, respectively

- **PluginName**

  Name of the plugin. The linker queries the dynamic library to provide an implementation of the plugin with the name *PluginName* for the specified interface type.

- **PluginOption**

  It can be used to pass an option to the plugin.

Specifying plugin configuration file using `–plugin-config` option

`–plugin-config` option takes a plugin configuration yaml file. Plugin configuration file should contain a `GlobalPlugins` list. Elements of the `GlobalPlugins` list defines which plugins should be loaded. Each value of the list is an object containing all the information required to load and initialize a specific plugin.

Plugin configuration file format should be as follows::

```
---
GlobalPlugins:
  - Type: PluginInterfaceType
    Name: PluginName
    Library: LibraryName
    Options: PluginOption
```

```
OutputSectionPlugins:
  - OutputSection : OutputSectionName
    Type : PluginInterfaceType
    Name : PluginName
    Library : LibraryName
    Options : PluginOption
```

`GlobalPlugins` list can specify any number of elements. `Options` member is optional.

`Library` name should be specified without the lib prefix on Linux and without the .so/.dll suffix on Linux/Windows

`ControlMemorySizePlugin` and `ControlFileSizePlugin` are output section plugins. Therefore, in the plugin configuration file, they need to be added to `OutputSectionPlugins` list.

Note

Only one output section plugin can be attached to an output section.

User Plugin Workflow

The following steps describe how to develop a plugin:

1. Determine the appropriate plugin interfaces for your plugin.

   A plugin type can be one of `LinkerPlugin`, `SectionIterator`, `SectionMatcher`, `OutputSectionIterator`, `ControlMemorySize`, and `ControlFileSize`.

   A plugin should ideally follow the unix-philosophy of doing one thing well. You should generally try to avoid inheriting from multiple plugin interfaces if possible. It generally leads to more complicated code, and a plugin that does more than one thing. But that being said, if you do think functionalities of multiple plugin interfaces will improve your plugin's design and readability, then you should go ahead with inheriting from multiple plugins.

   You can also create and chain multiple plugins for your task, similar in ideology to how unix utilities operate, where action of one plugin depends on the action and result of the previous one.

1. Create a C++ source file that will contain your plugin(s).

   Plugins source file needs to include the headers of the plugin interface(s) that the plugin(s) inherits from. Plugin interfaces header files are contained in, `${HEXAGON_TOOLCHAIN}/Tools/include/ELD/PluginAPI`.

3. Create a C++ class that inherits from the appropriate plugin interface(s) for each plugin.

4. Override virtual functions.

   Virtual functions are the means the linker works with the plugins. Each plugin interface provides hooks at various place in the linker codebase, the hooks functionalities are implemented by overriding virtual functions. There are also other virtual functions that do not implement functionalities for the hooks, but are required to work with the plugin infrastructure. Example of these virtual functions are: `Plugin::GetName`, `Plugin::GetLastError`, `Plugin::GetLastErrorAsString` etc.

5. Associate the plugins with unique names.

6. Build a shared library for the source file.

   In a unix environment, shared library for the plugins can be created as:

   ```
   # Compile the plugins source file
   clang++ -c -I${HEXAGON_TOOLCHAIN_ROOT}/Tools/include ${SOURCE_BASENAME}.cpp -fPIC -stdlib=libc++

   # Link the plugin library with linker wrapper library, LW.
   clang++ -shared ./${SOURCE_BASENAME}.o -L${HEXAGON_TOOLCHAIN_ROOT}/Tools/lib -lLW -stdlib=libc++ -o lib${SOURCE_BASENAME}.so
   ```

7. Define `RegisterAll` function in C linkage.

   `RegisterAll` function goal is to register the plugins contained in the file. Registering a plugin here simply means to initialize a plugin object.

   Linker calls this function for each plugin library. This function should creates a plugin object for each plugin that the library contains.

8. Define a `getPlugin(const char *pluginName)` function in C linkage.

   `getPlugin(const char* plugin)` function goal is to return the requested plugin. Each plugin in the plugin source file should be uniquely identifiable by a name.

   Linker calls this function at the time of loading plugins to get a plugin object of plugin named `pluginName`.

9. Make the plugin report its API version

   Linker verifies linker plugin compatibility. Linker will report an error and refuse to load a plugin if it determines that it is not compatible.

   Linker plugin compatibility is determined using Linker API version. Plugin API version consists of major and minor version numbers. A plugin is compatible with the linker if plugin's major API version is equal to the linker major API version and plugin's minor API version is equal or lower to the linker's minor API version. The major API version is incremented when a existing functionality in the API is changed in an incompatible way. Such changes are expected to be rare. The minor API version is incremented when new functionality is added to the Plugin API. Note that Plugin API versions are distinct from linker release version numbers. The current linker Plugin API version is reported by the `–version` command line option.

   For the versioning mechanism to work, each plugin must report its API version by exporting the `getPluginAPIVersion(unsigned int *Major, unsigned int *Minor)` function (with C linkage). For convenience, this function is defined in the header file `PluginVersion.h` and including this file in one of the plugin's C++ source files will automatically make the plugin report its correct API version.

   Starting with version 19.0, each plugin must report its API version. Linker will refuse to load a plugin that does not report its version.

10. (Optionally) Define a `getPluginConfig(const char *pluginName)` in C linkage to return the configuration object of the requested plugin.

11. (Optionally) Define a `Cleanup` function in C linkage.

    This function is called at the time unloading plugins. This function generally deletes any allocated memory that was required for the lifetime of the plugin library.

How the linker runs plugin

Linker performs the following operations to load, run and unload plugins.

1. Finds all the plugins, from linker script and plugins configuration file, that needs to be attached to the link process.
2. Loads all the specified plugin libraries.
   1. To find plugin libraries, `LD_LIBRARY_PATH` environment variable is used on unix environment.
   2. Standard method for searching dynamic libraries is used in Windows.
3. Calls `RegisterAll` function from each plugin library. This function should ideally create the plugin objects for each plugin defined in the library.
4. Calls `getPlugin(const char *pluginName)` function for each plugin.

   This function returns an appropriate plugin object for *pluginName* plugin. This object is used to run the plugin.
5. Calls `getPluginConfig(const char *pluginName)` function, if available, for each plugin. This function returns an appropriate plugin configuration object for *pluginName* plugin.
6. Inspects the plugin to verify that the plugin load specification and plugin type have same plugin interface type.
7. Initializes each plugin at their *Init* hook point by calling `Plugin::Init(std::string Option)` virtual function. *Option* argument is optionally specified by the plugin user at the plugin load specification.
8. Calls `Cleanup` function, if available, for each plugin library. This function should ideally delete any allocated memory that was required for the lifetime of the plugin library.
9. Unload the plugins and the plugin libraries.

Tracing plugins

Plugin workflow can be traced using the option `–trace=plugin`.

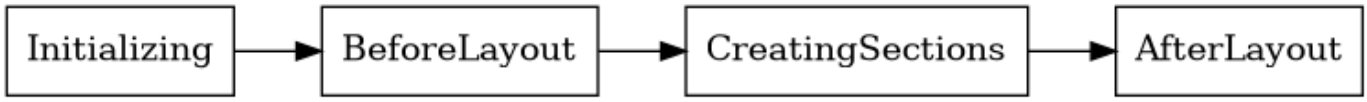
This option informs the linker to emit detailed diagnostics for all the plugins.

A sample trace output is shown below:

```
...
...
Note: Registration function found RegisterAll in Library libDiagOpt.so
Note: Plugin handler getPlugin found in Library libDiagOpt.so
Note: Cleanup function found Cleanup in Library libDiagOpt.so
Note: Plugin Config function getPluginConfig found in library libDiagOpt.so
Note: Registering all plugin handlers for plugin types
Note: Found plugin handler for plugin type DIAGRELOCATION in Library libDiagOpt.so
Note: Initializing Plugin libDiagOpt.so, requested by Plugin DIAGRELOCATION having Name DIAGOPT
...
...
Note: Plugin DIAGRELOCATION Destroyed

<------------------------> DiagOpt::Destroy() AfterLayout<------------------------>
Note: Unloaded Library libDiagOpt.so.16
```

Link States



The link process has different run states, also known as link states. The actions that a plugin can perform at any given time depend on the current link state. Many actions are only meaningful for specific link states. Understanding what happens in each link state will help determine which actions can be performed in each state. Therefore, a thorough understanding of linker and link states is crucial for writing a linker plugin. Different link states are described below:

Initializing

The linker begins with the *Initializing* link state. In this state, the linker reads input files, initializes standard sections and configurations, and read relocations. Not much of interest happens during this state.

Let's move on to the more happening states.

BeforeLayout

After the *Initializing* state, the linker enters the *BeforeLayout* link state. Here, it performs section rule matching, garbage collection, updates input sections with attributes (such as KEEP) present in the linker script, and updates the linker cache-file, among other things.

CreatingSections

After the *BeforeLayout* state, the linker enters the *CreatingSections* link state. Here, it transfers the contents (chunks) of input sections into output sections and finalizes the layout and symbols.

Note that the section rule-matching is already complete before the *CreatingSections* state, therefore, plugins cannot alter the section rule-matching in *CreatingSections* state or any subsequent state.

AfterLayout

After the *BeforeLayout* state, the linker enters *Afterlayout* link state. Here, it finalizes the relocations and writes the output object file.

In this link state, a plugin can compute output image layout checksum using `eld::plugin::LinkerWrapper::getImageLayoutChecksum`. A plugin cannot perform any action that tries to modify the image layout now.

Deep Dive into the Plugin Types

There are two distinct kinds of linker plugins: LinkerPlugin and layout plugins. LayoutPlugins focus on simplifying the modification of specific layout components, such as input and output sections. In contrast LinkerPlugins are more general, uniform and offer more functionalitier than the layout plugins.

PluginBase class

This is the base plugin class for all the plugin interface classes. It provides common functionalities to all the plugin interfaces. It is a pure virtual class and thus cannot be instantiated / directly used.

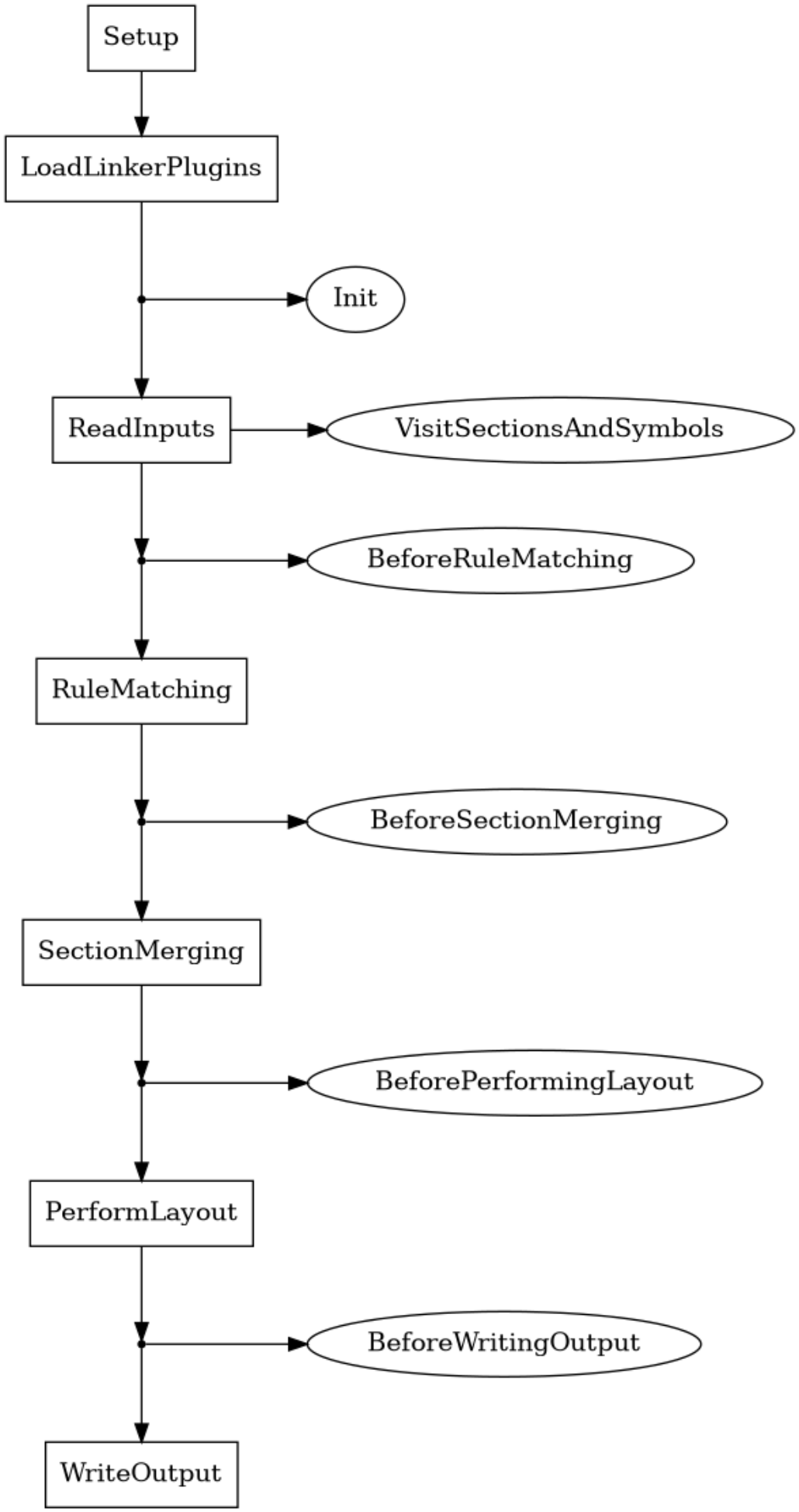
Please note, that even though the virtual functions – `Plugin::Init`, `Plugin::Run` and `Plugin::Destroy` are available in all the plugin interfaces, they map to different hooks in the linker for different plugin interfaces. For example, `Run` hook of `SectionIteratorPlugin‘` is not equivalent to the `Run` hook of `OutputSectionIteratorPlugin` even though in both cases, the hooks functionality is defined by overriding the `Run` virtual function. For the base `Plugin`, these functions are not mapped to any hook in the linker.

LinkerPlugin Type

`LinkerPlugin` is the most versatile and the recommended plugin type for creating new plugins. It has hooks that covers the entire linker flow and offers more functionality than the layout plugins. `LinkerPlugin` has hooks of the two forms:

- `ActBefore<LinkState>`
- `Visit<LinkComponent>`

The `ActBefore<LinkState>` hooks are called *just* before the linker enters the link state `LinkState`. The `Visit<LinkComponent>` hooks are called *immediately* after the linker creates a link component (input section, symbol, ...). For example, `VisitSymbol(eld::plugin::Symbol S)` is called immediately after the symbol is created.

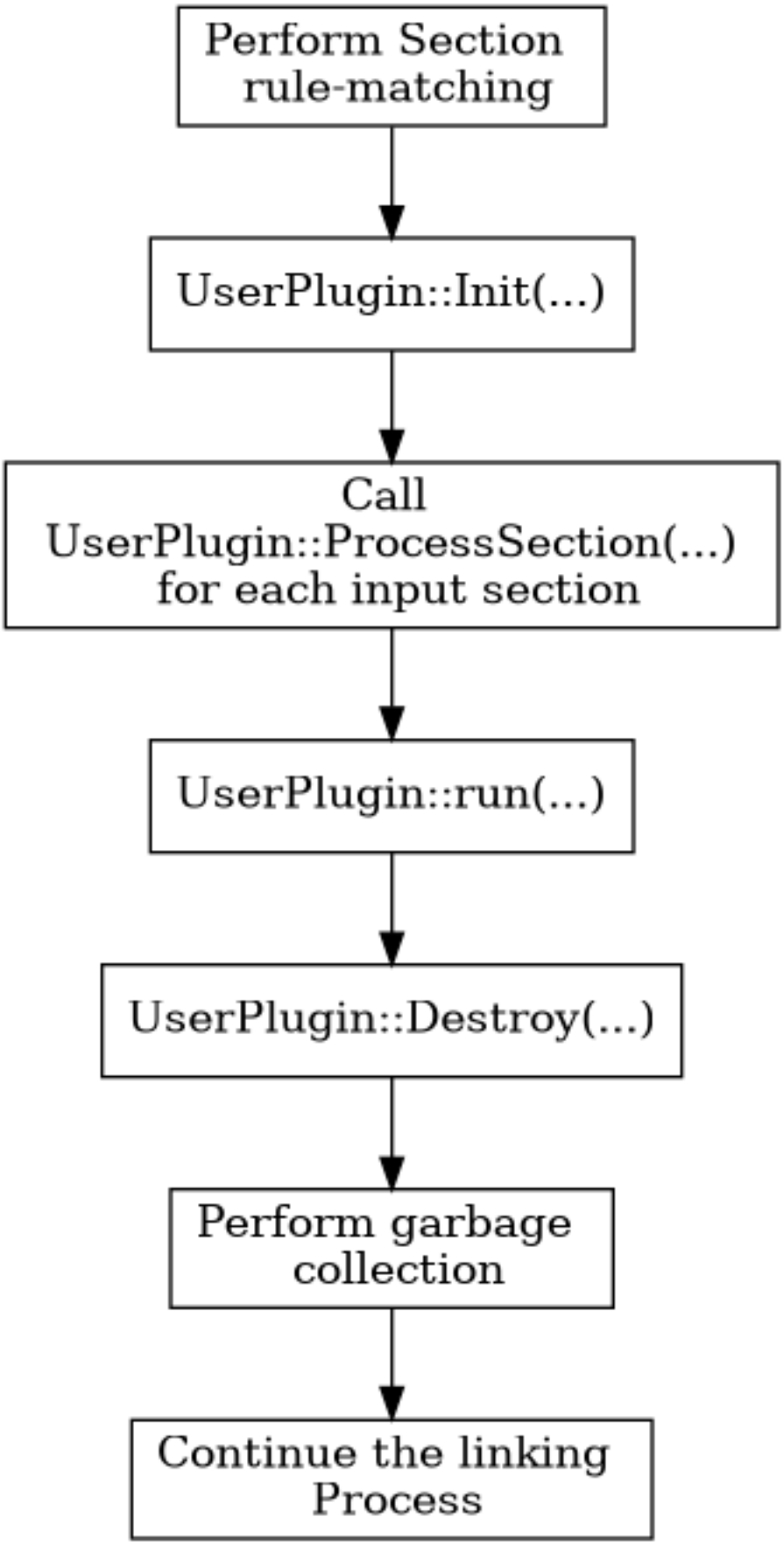

Setup
LoadLinkerPlugins
Init
ReadInputs
VisitSectionsAndSymbols
BeforeRuleMatching
RuleMatching
BeforeSectionMerging
SectionMerging
BeforePerformingLayout
PerformLayout
BeforeWritingOutput
WriteOutput

Section Matcher Interface

`SectionMatcherPlugin` interface iterates input sections. It provides four hooks: `Init`, `ProcessSection`, `Run` and `Destroy`. To define the behavior of these hooks, the functions `Plugin::Init`, `SectionMatcher::ProcessSection` and `Plugin::Destroy` must be overridden.

`ProcessSection(Section S)` callback hook function is called for each input section.

`SectionMatcherPlugin` is called into action after the linker reads input files and performs section rule-matching, but before garbage collection. As a consequence, garbage-collection information cannot be accessed during `SectionMatcherPlugin` run. Link state throughout the plugin run is `eld::plugin::LinkerWrapper::BeforeLayout`.



`UserPlugin::Init` is called before `UserPlugin::ProcessSection`, and `UserPlugin::Run` and `UserPlugin::Destroy` are called subsequently after processing input sections, as described in the figure above.

Linker script keyword for `SectionMatcherPlugin` interface is `PLUGIN_SECTION_MATCHER`. Therefore, to run a `SectionMatcherPlugin` plugin using a linker script, add the following line to the linker script:

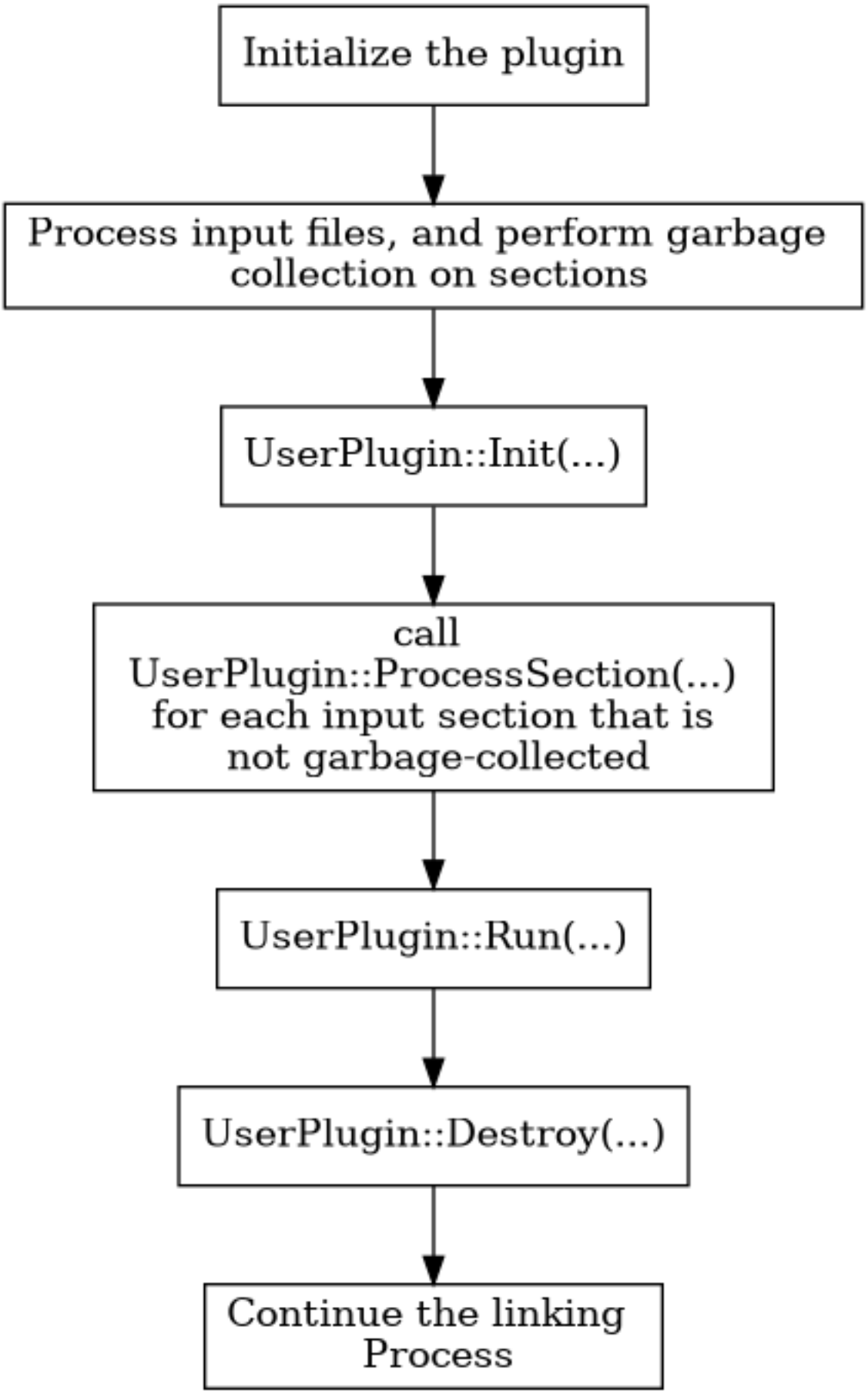
```
PLUGIN_SECTION_MATCHER("LibraryName", "PluginName" [, "PluginOption"])
```

Section Iterator Interface

`SectionIteratorPlugin` interface iterates over input sections. It provides four hooks: `Init`, `ProcessSection`, `Run` and `Destroy`. To define the behavior of these hooks, the functions `Plugin::Init`, `Plugin::ProcessSection`, `SectionIterator::ProcessSection` and `Plugin::Destroy` must be overridden.

`ProcessSection(Section S)` callback hook function is not called for garbage collected input sections. However, it is called for discarded input sections.

`SectionIteratorPlugin` is called into action after the linker performs section rule-matching and garbage collection. As a consequence, section rule matching and garbage collection information is accessible during `SectionIteratorPlugin` run. Link state throughout the plugin run is `eld::plugin::LinkerWrapper::BeforeLayout`.



`UserPlugin::Init` is called before `ProcessSection`, and `UserPlugin::Run` and `UserPlugin::Destroy` are called after `ProcessSection`, as described in the figure above.

Linker script keyword for `SectionIteratorPlugin` interface is `PLUGIN_ITER_SECTIONS`. Therefore, to run a `SectionIteratorPlugin` plugin using a linker script, add the following line to the linker script:

```
PLUGIN_ITER_SECTIONS("LibraryName", "PluginName" [, "PluginOption"])
```

Output Section Iterator Interface

`outputSectionIteratorPlugin` interface iterates over all the output sections. It provides four hooks: `Init`, `ProcessOutputSection`, `Run` and `Destroy`. To define the behavior of these hooks, the functions `Plugin::Init`, `Plugin::ProcessSection`, `OutputSectionIteratorPlugin::ProcessOutputSection` and `Plugin::Destroy` must be overridden. `OutputSectionIteratorPlugin` is called at different link states:

- BeforeLayout
- CreatingSections
- AfterLayout

All the plugin hooks are called for each of the link states. That means, each of `Init`, `Run` and `Destory` are called 3 times over the complete plugin run. And `ProcessOutputSection` is called three times for each output section over the complete plugin run.

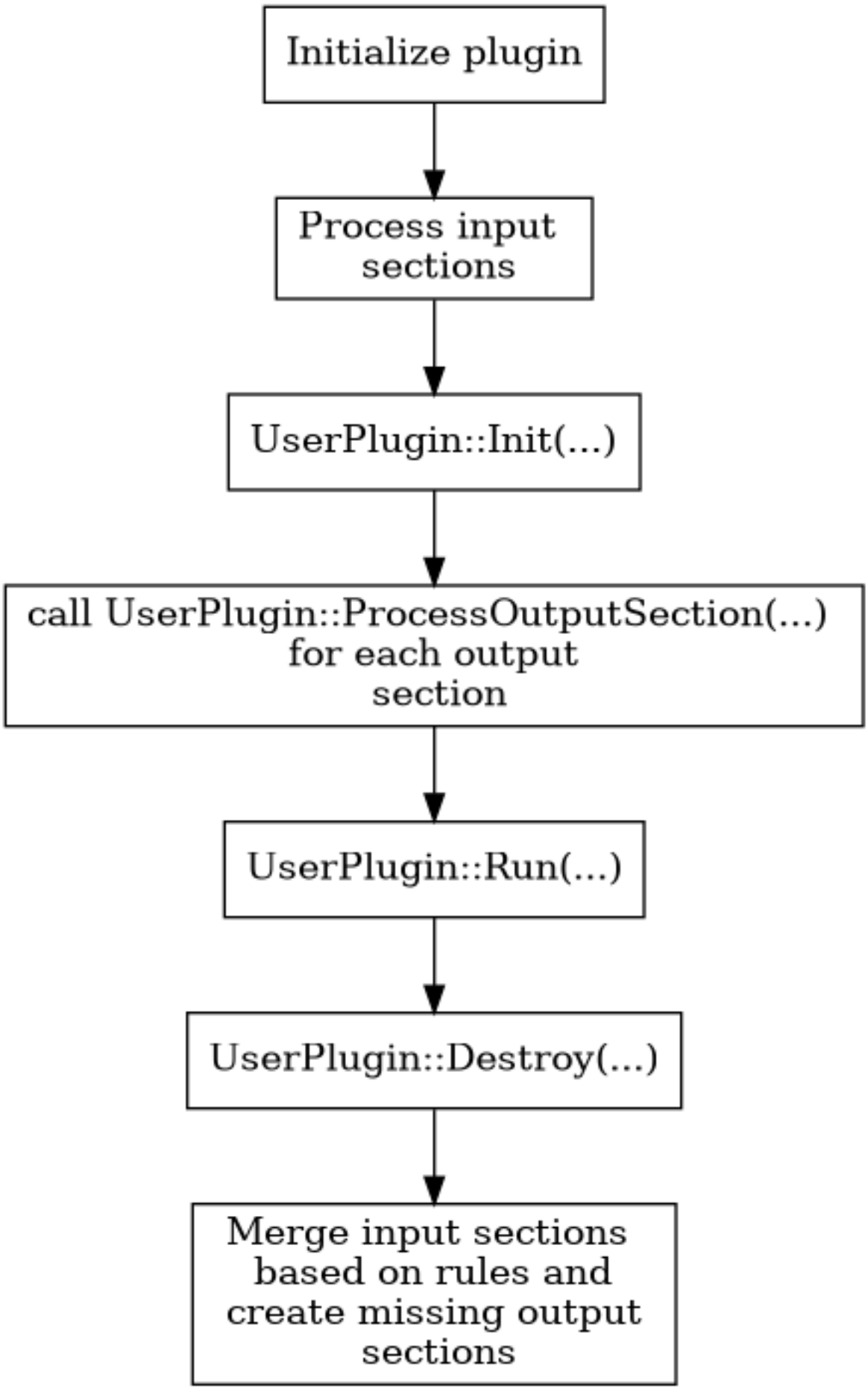
The link state can be queried via `LinkerWrapper::getState` member function.

Linker script keyword for `OutputSectionIteratorPlugin` interface is `PLUGIN_OUTPUT_SECTION_ITER`. Therefore, to run a `SectionIteratorPlugin` using a linker script, add the following line to the linker script:

```
PLUGIN_OUTPUT_SECTION_ITER("LibraryName", "PluginName" [, "PluginOption"])
```

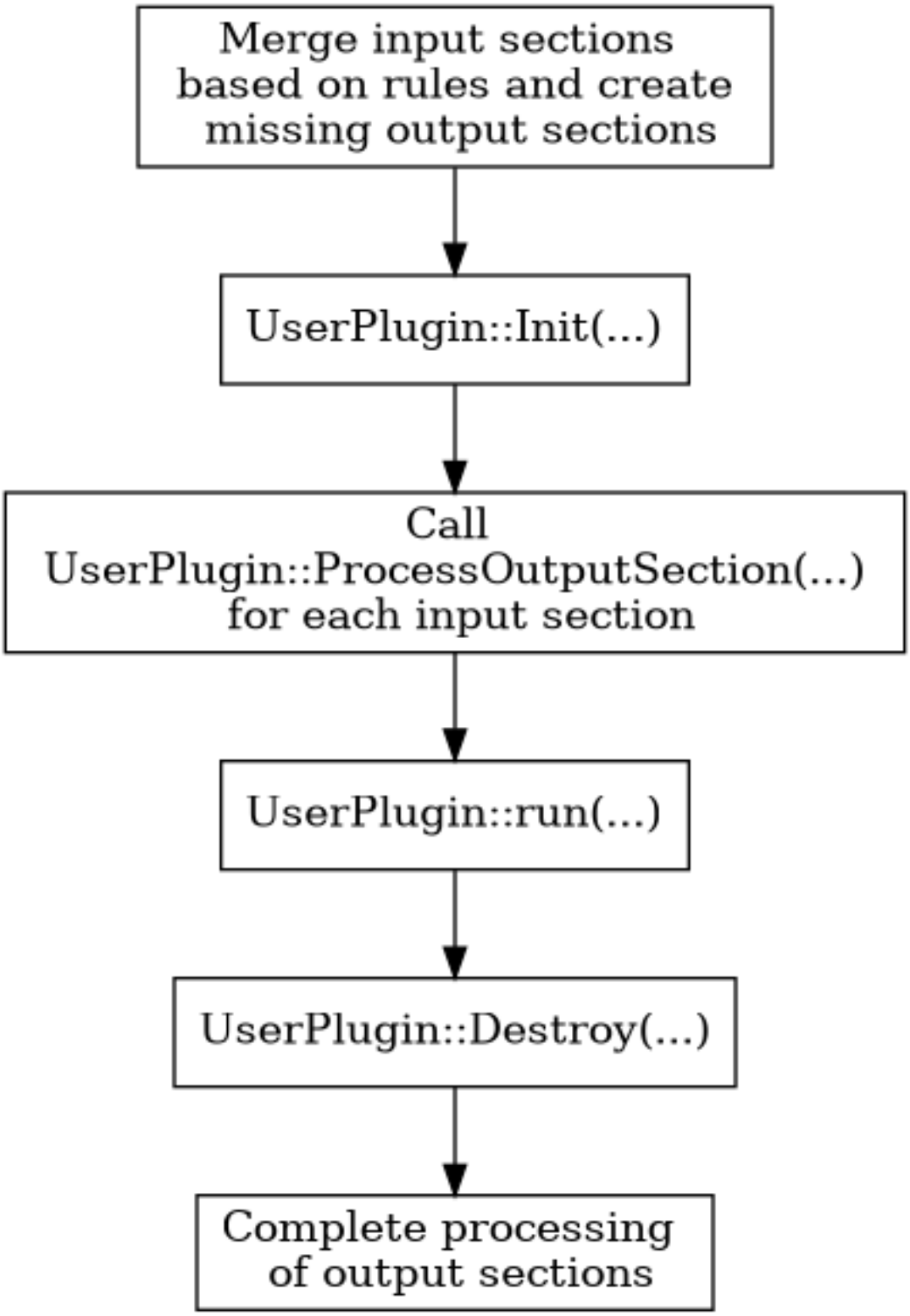
Different phases of `OutputSectionIteratorPlugin` are described below:

BeforeLayout – Phase 1



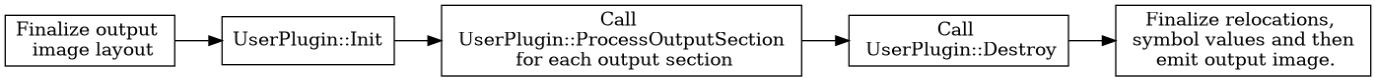
This phase is called into action after the linker has performed section rule-matching and before the linker has performed section merging.

CreatingSections – Phase 2



This phase is called into action after the linker has performed section merging.

AfterLayout – Phase 3

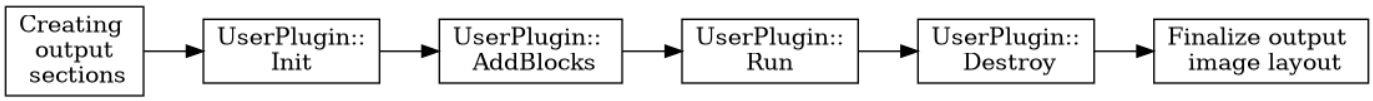


This phase is called into action after the linker has finalized the image layout and before the linker has written the output object file.

ControlMemorySizePlugin

This interface allows the plugin to add input sections to the output image. These newly added input sections are generally, but not necessary, based on some already existing output section.

`ControlMemorySizePlugin` is an output section plugin. Plugins of this type has to explicitly mark each output section that it needs to run on. Since this is an output section plugin, `init` and `destroy` functions are also called for each output section the plugin runs on. This plugin is called in the *LinkerWrapper::CreatingSections* step of the linker.



`init`, `AddBlocks`, `Run`, `GetBlocks` and `Destroy` functions are called for each output section.

Brief description of `ControlMemorySizePlugin` hooks:

`init`

This hook initializes the plugin for the output section.

`AddBlocks`

This hook gives access to the current output section to the plugin. `Block` parameter of the hook function contains the current output section. It is generally used in the creation of the new block which linker can use as an additional input section.

`Run`

This hook is generally used to process the current output section, and create any new sections that linker should use as additional input sections.

`GetBlocks`

This hook returns a vector of `Block`. These blocks are used by the linker as additional input sections.

`Destroy`

This hook is used to perform any required clean-up.

Linker script keyword for ControlMemorySize interface is `PLUGIN_CONTROL_MEMSZ`. Therefore, to run a SectionIterator plugin on an output section, `.out` using a linker script, add the following line to the output section description of `.out` in the linker script:

```
SECTIONS {
   .out PLUGIN_CONTROL_MEMSZ("LibraryName", "PluginName" [, "PluginOption"]) : { *(.out) }
 }
```

Linker Plugin Examples

This section introduces and explains plugin framework APIs in a hands-on manner. The goal of these examples is to demonstrate how to use plugin framework APIs, rather than to present practical plugin use cases. Each plugin example introduces some new plugin framework APIs while keeping the example short and simple.

Each example will include the plugin, instructions on how to build and run it, and an analysis of the plugin's run. To build and run the plugin, you will need access to the Hexagon toolchain. In the snippets below, the substitution *${HEXAGON}* refers to the Hexagon toolchain installation path. Let's get started.

Exclude symbols from the output image symbol table

This example plugin describes how to exclude symbols from the output image symbol table. Note that the plugin does not completely remove symbols from the link process, it only excludes symbols from the output image symbol table.

ExcludeSymbols plugin with self-contained documentation.

```
#include "ELD/PluginAPI/PluginVersion.h"
#include "ELD/PluginAPI/SectionIteratorPlugin.h"
#include <string>
#include <vector>

class ExcludeSymbols : public eld::plugin::SectionIteratorPlugin {
public:
  ExcludeSymbols() : eld::plugin::SectionIteratorPlugin("ExcludeSymbols") {}

  // 'Init' callback hook can be used for initialization and preparations.
  void Init(std::string Options) override {
    m_SymbolsToRemove = {"foo", "fooagain", "bar", "baragain"};
  }

  // 'processSection' callback hook is called for each input section that is not
  // garbage-collected.
  void processSection(eld::plugin::Section O) override {}

  // 'Run' callback hook is called after 'processSection' callback hook calls.
  // It is called once for each section iterator plugin run.
  eld::plugin::Plugin::Status Run(bool trace) override {
    for (auto symName : m_SymbolsToRemove) {
      eld::plugin::Symbol S = Linker->getSymbol(symName);
      if (S)
        Linker->removeSymbolTableEntry(S);
    }
    return eld::plugin::Plugin::Status::SUCCESS;
  }

  // 'Destroy' callback hook can be used for finalization and clean-up tasks.
  // It is called once for each section iterator plugin run.
  void Destroy() override {}

  uint32_t GetLastError() override { return 0; }

  std::string GetLastErrorAsString() override { return "Success"; }

  std::string GetName() override { return "ExcludeSymbols"; }

private:
  std::vector<std::string> m_SymbolsToRemove;
};

eld::plugin::Plugin *ThisPlugin = nullptr;

extern "C" {
```

```
bool RegisterAll() {
  ThisPlugin = new ExcludeSymbols{};
  return true;
}

eld::plugin::Plugin *getPlugin(const char *pluginName) { return ThisPlugin; }

void Cleanup() {
  if (!ThisPlugin)
    return;
  delete ThisPlugin;
  ThisPlugin = nullptr;
}
}
```

ExcludeSymbols plugin removes the symbols ‘foo’, ‘fooagain’, ‘bar’, and ‘baragain’, if they exist, from the output image symbol table.

To build the plugin, run the following command:

```
clang++-14 -o libExcludeSymbols.so ExcludeSymbols.cpp -std=c++17 -stdlib=libc++ -fPIC -shared \
-I${HEXAGON}/include -L${HEXAGON}/lib -lLW
```

Now, let’s see the effect of this plugin on a sample program.

1.c

```
int foo() { return 1; }
int fooagain() { return 2; }
int abc() { return 3; }

int bar = 3;
int baragain = 4;
int def = 5;

int main() { return 0; }
```

1.linker.script

```
PLUGIN_ITER_SECTIONS("ExcludeSymbols", "ExcludeSymbols");
```

Now, let’s build **1.c** with the *ExcludeSymbols* plugin enabled.

```
export LD_LIBRARY_PATH="${HEXAGON}/lib:${LD_LIBRARY_PATH}"
hexagon-clang -o 1.o 1.c -c -ffunction-sections -fdata-sections
hexagon-link -o 1.elf 1.o -T 1.linker.script
```

We can list the symbols present in an object file using hexagon-readelf.

```
hexagon-readelf -s 1.elf
```

You can observe in the symbol table output that ‘foo’, ‘fooagain’, ‘bar’, and ‘baragain’ symbols has been removed from the symbol table as directed by the plugin.

Add Linker Script Rules

This example plugin describes how to add and modify linker script rules. By doing so, you can perform section rule-matching at a more granular level than what is possible through linker scripts.

`LinkerScriptRule` is similar to an input section description in that it has a corresponding output section. However, unlike an input section description, it does not match input sections by pattern matching. Instead, a plugin must manually match chunks to a `LinkerScriptRule` object. This generally involves moving chunks from one `LinkerScriptRule` object to another. It is important to remove a chunk from the old `LinkerScriptRule` object once it has been added to a new one. It is an undefined behavior for a chunk to be part of multiple `LinkerScriptRule` objects.

Also, you must only move chunks from one `LinkerScriptRule` object to another in the *CreatingSections* link state. It is an undefined behavior to move chunks in other link states.

The *AddRule* plugin moves chunks from `foo` and `bar` output sections to the `var` output section. Let's see how to create this plugin.

AddRule plugin with self-contained documentation.

```
#include "ELD/PluginAPI/OutputSectionIteratorPlugin.h"
#include "ELD/PluginAPI/PluginVersion.h"

class AddRule : public eld::plugin::OutputSectionIteratorPlugin {
public:
  AddRule() : eld::plugin::OutputSectionIteratorPlugin("AddRule") {}

  // 'Init' callback hook can be used for initialization and preparations.
  // This plugin does not need any initialization or preparation.
  void Init(std::string cfg) override {}

  // 'processOutputSection' callback hook is called once for each output
  // section. In this function, the plugin stores 'var, 'foo' and 'bar' output
  // sections in member variables.
  void processOutputSection(eld::plugin::OutputSection O) override {
    // OutputSectionIterator plugin essentially runs three times.
    // It is run once for each of the following three link states: BeforeLayout,
    // CreatingSections and AfterLayout.
    // We are only interested in one link state, CreatingSections, as chunks can
    // only be moved from one LinkerScriptRule to another in the
    // CreatingSections link state. Thus, we simply return for the other link
    // states. We will do this for each callback hook function.
    if (Linker->getState() != eld::plugin::LinkerWrapper::State::CreatingSections)
      return;
    if (O.getName() == "var") {
      m_Var = O;
    } else if (O.getName() == "foo") {
      m_Foo = O;
    } else if (O.getName() == "bar") {
      m_Bar = O;
    }
  }

  // 'Run' callback hook is called after all the 'processSection' callback hook
  // calls.
  eld::plugin::Plugin::Status Run(bool trace) override {
    if (Linker->getState() != eld::plugin::LinkerWrapper::State::CreatingSections)
      return eld::plugin::Plugin::Status::SUCCESS;

    auto lastRule = m_Var.getLinkerScriptRules().back();
    // Create a new rule for m_Var output section.
    // Annotation is used to name the linker sript rule, and is useful
    // for diagnostic purposes.
    eld::plugin::LinkerScriptRule newRule = Linker->createLinkerScriptRule(
        m_Var, /*Annotation=*/"Move foo and bar chunks to var");
    // Insert the newly created linker script rule in the m_Var output section.
    // We can also insert the newly created rule before some already existing
    // rule using LinkerWrapper::insertBeforeRule API.
    Linker->insertAfterRule(m_Var, lastRule, newRule);
    moveChunks(m_Foo, newRule);
    moveChunks(m_Bar, newRule);

    return eld::plugin::Plugin::Status::SUCCESS;
  }
```

```
  // 'Destroy' callback hook can be used for finalization and clean-up tasks.
  // It is called once for each section iterator plugin run.
  void Destroy() override {}

  uint32_t GetLastError() override { return 0; }

  std::string GetLastErrorAsString() override { return "Success"; }

  std::string GetName() override { return "AddRule"; }

private:
  void moveChunks(eld::plugin::LinkerScriptRule oldRule,
                  eld::plugin::LinkerScriptRule newRule) {
    for (eld::plugin::Chunk C : oldRule.getChunks()) {
      // It is crucial to maintain that no two LinkerScriptRule objects contain
      // the same chunk. It is an undefined behavior for a chunk to be contained
      // by multiple linker script rules.
      Linker->addChunk(newRule, C);
      Linker->removeChunk(oldRule, C);
    }
  }

  void moveChunks(eld::plugin::OutputSection oldSection,
                  eld::plugin::LinkerScriptRule newRule) {
    for (eld::plugin::LinkerScriptRule rule : oldSection.getLinkerScriptRules())
      moveChunks(rule, newRule);
  }

private:
  eld::plugin::OutputSection m_Var = eld::plugin::OutputSection(nullptr);
  eld::plugin::OutputSection m_Foo = eld::plugin::OutputSection(nullptr);
  eld::plugin::OutputSection m_Bar = eld::plugin::OutputSection(nullptr);
};

eld::plugin::Plugin *ThisPlugin = nullptr;

extern "C" {
// RegisterAll should initialize all the plugins that a plugin library aims
// to provide. Linker calls this function before running any plugins provided
// by the library.
bool RegisterAll() {
  ThisPlugin = new AddRule{};
  return true;
}

// Linker calls this function to request an instance of the plugin
// with the plugin name pluginName. pluginName is provided in the plugin
// invocation command.
eld::plugin::Plugin *getPlugin(const char *pluginName) { return ThisPlugin; }

// Cleanup should free all the resources owned by a plugin library.
// Linker calls this function after all runs of the plugins provided
// by the library have completed.
void Cleanup() {
  if (!ThisPlugin)
    return;
  delete ThisPlugin;
  ThisPlugin = nullptr;
```

```
}
}
```

To build the plugin, run the following command:

```
clang++-14 -o libAddRule.so AddRule.cpp -std=c++17 -stdlib=libc++ -fPIC -shared \
-I${HEXAGON}/include -L${HEXAGON}/lib -lLW
```

Now, let's see the effect of this plugin on a sample program.

1.c

```
int foo() { return 1; }
int bar() { return 2; }
int baz() { return 3; }

int int_var = 1;
long long_var = 2;
double double_var = 3;

int main() {
  return 0;
}
```

1.linker.script

```
SECTIONS {
  foo : { *(.text.foo) }
  bar : { *(.text.bar) }
  baz : { *(.text.baz) }
  var : { *(*var) }
}

PLUGIN_OUTPUT_SECTION_ITER("AddRule", "AddRule");
```

Now, let's build 1.c with the *AddRule* plugin enabled.

```
export LD_LIBRARY_PATH="${HEXAGON}/lib:${LD_LIBRARY_PATH}"
hexagon-clang -o 1.o 1.c -c -ffunction-sections -fdata-sections
hexagon-link -o 1.elf 1.o -T 1.linker.script -Map 1.map.txt
```

We can see the symbol to section mapping using hexagon-readelf.

```
hexagon-readelf -Ss 1.elf
```

readelf output:

```
There are 8 section headers, starting at offset 0x12a0:

Section Headers:
  [Nr] Name              Type            Address  Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] baz               PROGBITS        00000000 001000 00000c 00  AX  0   0 16
  [ 2] .text.main        PROGBITS        00000010 001010 000014 00  AX  0   0 16
  [ 3] var               PROGBITS        00000030 001030 00002c 00 WAXp  0   0 16
  [ 4] .comment          PROGBITS        00000000 00105c 0000d2 01  MS  0   0  1
  [ 5] .shstrtab         STRTAB          00000000 00112e 000037 00      0   0  1
  [ 6] .symtab           SYMTAB          00000000 001168 0000e0 10      7   6  4
  [ 7] .strtab           STRTAB          00000000 001248 000054 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  R (retain), p (processor specific)
```

```
Symbol table '.symtab' contains 14 entries:
   Num:    Value  Size Type    Bind   Vis       Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT   UND
     1: 00000000     0 SECTION LOCAL  DEFAULT     1 baz
     2: 00000000     0 SECTION LOCAL  DEFAULT     4 .comment
     3: 00000010     0 SECTION LOCAL  DEFAULT     2 .text.main
     4: 00000030     0 SECTION LOCAL  DEFAULT     3 var
     5: 00000000     0 FILE    LOCAL  DEFAULT   ABS 1.c
     6: 00000000    12 FUNC    GLOBAL DEFAULT     1 baz
     7: 00000010    20 FUNC    GLOBAL DEFAULT     2 main
     8: 00000030     4 OBJECT  GLOBAL DEFAULT     3 int_var
     9: 00000034     4 OBJECT  GLOBAL DEFAULT     3 long_var
    10: 00000038     8 OBJECT  GLOBAL DEFAULT     3 double_var
    11: 00000040    12 FUNC    GLOBAL DEFAULT     3 foo
    12: 00000050    12 FUNC    GLOBAL DEFAULT     3 bar
    13: 00000061     0 NOTYPE  GLOBAL DEFAULT   ABS __end
```

You can observe in the readelf output that the section `var` contains the `foo` and the `bar` symbols, which is exactly what we expected from the plugin.

Reading INI Files Functionality

The plugin framework provides built-in support for the input/output operations of INI files. Linker plugins, like any other tool, may require external options and configurations. Traditionally, linker plugins use INI files for this purpose. This plugin demo demonstrates how to use INI files to provide options and configurations to a plugin. You can also use JSON, YAML, plain text, or any other format you prefer. However, for consistency with other linker plugins, I suggest using INI files for the plugin options and configurations.

The demo will feature the *PrintSectionsInfo* plugin. This plugin will print section information for all input sections that match any of the section name patterns specified in the plugin configuration file. The plugin will expect the configuration file to in INI format.

PrintSectionsInfo plugin with self-contained documentation.

```
#include "ELD/PluginAPI/PluginVersion.h"
#include "ELD/PluginAPI/SectionMatcherPlugin.h"
#include <iostream>
#include <set>
#include <string>

class PrintSectionsInfo : public eld::plugin::SectionMatcherPlugin {
public:
  PrintSectionsInfo() : eld::plugin::SectionMatcherPlugin("PrintSectionsInfo") {}

  // 'Init' callback hook can be used for initialization and preparations.
  // We will read the configuration file here.
  void Init(std::string cfg) override {
    // 'Linker' is an inherited LinkerWrapper member variable. LinkerWrapper
    // is to be used to make any calls to the Linker.
    // LinkerWrapper::findConfigFile searches the file in standard paths and
    // returns a resolved path if the file is found.
    std::string configPath = Linker->findConfigFile(cfg);
    auto expINIFile = Linker->readINIFile(configPath);
    // If plugin configuration file cannot be read, then report the error,
    // set linker to fatal error and return.
    if (!expINIFile) {
      Linker->reportDiagEntry(std::move(expINIFile.error()));
      Linker->setLinkerFatalError();
      return;
    }
    eld::plugin::INIFile I = std::move(expINIFile.value());
```

```
    // Read patterns from the plugin configuration file and store
    // them in a member variable. These patterns will be used later
    // to decide which sections information should be printed.
    auto sections = I.getSection("sections");
    for (auto item : sections) {
      if (item.second == "1")
        m_SectionPatterns.insert(item.first);
    }
  }

  // 'processSection' callback hook is called for each input section.
  void processSection(eld::plugin::Section S) override {
    if (shouldPrintSectionInfo(S)) {
      std::cout << S.getName() << "\n";
      std::cout << "Input file: " << S.getInputFile().getFileName() << "\n";
      std::cout << "Section index: " << S.getIndex() << "\n";
      std::cout << "Section alignment: " << S.getAlignment() << "\n";
      std::cout << "\n";
    }
  }

  // 'Run' callback hook is called after 'processSection' callback hook calls.
  // It is called once for each section iterator plugin run.
  eld::plugin::Plugin::Status Run(bool trace) override {
    return eld::plugin::Plugin::Status::SUCCESS;
  }

  // 'Destroy' callback hook can be used for finalization and clean-up tasks.
  // It is called once for each section iterator plugin run.
  void Destroy() override {}

  uint32_t GetLastError() override { return 0; }

  std::string GetLastErrorAsString() override { return "Success"; }

  std::string GetName() override { return "PrintSectionsInfo"; }

private:
  // Returns true if the section information should be printed.
  bool shouldPrintSectionInfo(eld::plugin::Section S) {
    // If the section name matches the pattern specified in the plugin
    // configuration file then return true.
    for (auto pattern : m_SectionPatterns) {
      if (S.matchPattern(pattern))
        return true;
    }
    return false;
  }

  // Stores section name patterns.
  std::set<std::string> m_SectionPatterns;
};

eld::plugin::Plugin *ThisPlugin = nullptr;

extern "C" {
// RegisterAll should initialize all the plugins that the plugin library aims
// to provide. Linker calls this function before running any plugins provided
// by the library.
bool RegisterAll() {
```

```
  ThisPlugin = new PrintSectionsInfo{};
  return true;
}

// Linker calls this function to request an instance of the plugin
// with the plugin name pluginName. pluginName is provided in the plugin
// invocation command.
eld::plugin::Plugin *getPlugin(const char *pluginName) { return ThisPlugin; }

// Cleanup should free all the resources owned by the plugin library.
// Linker calls this function after all runs of the plugins provided
// by the library have completed.
void Cleanup() {
  if (!ThisPlugin)
    return;
  delete ThisPlugin;
  ThisPlugin = nullptr;
}
}
```

To build the plugin, run:

```
clang++-14 -o libPrintSectionsInfo.so PrintSectionsInfo.cpp -std=c++17 -stdlib=libc++ -fPIC -shared \
-I${HEXAGON}/include \
-L${HEXAGON}/lib -lLW
```

Now, let's see the effect of this plugin on a sample program.

1.c

```
int i[5] = {1, 2, 3, 4, 5};
int arr[100];

int foo() { return 1; }
int bar() { return 2; }

int main() { return 0; }
```

1.linker.script

```
# PluginConf.ini is the plugin configuration file name.
PLUGIN_SECTION_MATCHER("PrintSectionsInfo", "PrintSectionsInfo", "PluginConf.ini");
```

PluginConf.ini

```
[sections]
*i=1
*arr=1
*foo=0
*bar=1
```

Now, let's build 1.c with *PrintSectionsInfo* plugin enabled.

```
export LD_LIBRARY_PATH="${HEXAGON}/lib:${LD_LIBRARY_PATH}"
hexagon-clang -o 1.o 1.c -c -ffunction-sections -fdata-sections
hexagon-link -o 1.elf 1.o -T 1.linker.script
```

The plugin gives the below output:

```
.text.bar
Input file: 1.o
Section index: 4
Section alignment: 16

.data.i
Input file: 1.o
```

```
Section index: 6
Section alignment: 8

COMMON.arr
Input file: CommonSymbols
Section index: 0
Section alignment: 8
```

As expected, the plugin prints the section information of the sections whose names match a pattern specified in the plugin configuration file, *PluginConf.ini*.

Modify Relocations

The *ModifyRelocations* example plugin demonstrates how to inspect and modify relocations. This plugin modifies the relocation symbol from `HelloWorld` to `HelloQualcomm`. Additionally, it prints the relocation source section and symbol for each relocation it iterates over.

To inspect and modify relocations, *ModifyRelocations* uses `LinkerPluginConfig`. `LinkerPluginConfig` provides a callback hook for relocations. This callback hook function is called for each relocation that is of a registered relocation type. Relocation types can be registered using `LinkerWrapper::registerReloc`.

ModifyRelocations plugin with self-contained documentation.

```
#include "ELD/PluginAPI/PluginVersion.h"
#include "ELD/PluginAPI/SectionIteratorPlugin.h"
#include <iostream>

class ModifyRelocations : public eld::plugin::SectionIteratorPlugin {
public:
  ModifyRelocations() : SectionIteratorPlugin("ModifyRelocations") {}

  // 'Init' callback hook can be used for initialization and preparations.
  // This plugin does not need any initialization or preparation.
  void Init(std::string cfg) override {}

  // 'processSection' callback hook of SectionIteratorPlugin is called for
  // each non-garbage collected section.
  void processSection(eld::plugin::Section S) override {}

  // 'Run' callback hook is called after all the 'processSection' callback hook
  // calls. It is called once for each section iterator plugin run.
  // This plugin does not need to run anything.
  eld::plugin::Plugin::Status Run(bool trace) override {
    return eld::plugin::Plugin::Status::SUCCESS;
  }

  // 'Destroy' callback hook can be used for finalization and clean-up tasks.
  // It is called once for each section iterator plugin run.
  // This plugin does not need any finalization and clean-up.
  void Destroy() override {}

  uint32_t GetLastError() override { return 0; }

  std::string GetLastErrorAsString() override { return "Success"; }

  std::string GetName() override { return "ModifyRelocations"; }

  eld::plugin::LinkerWrapper *getLinker() { return Linker; }
};

// LinkerPluginConfig allows to inspect and modify relocations.
class ModifyRelocationsPluginConfig : public eld::plugin::LinkerPluginConfig {
```

```
public:
  ModifyRelocationsPluginConfig(ModifyRelocations *P)
      : LinkerPluginConfig(P), P(P) {}

  void Init() override {
    // Register R_HEX_B22_PCREL relocation type.
    // Linker will call RelocCallBack callback hook function on each relocation
    // that is of a registered relocation type.
    std::string b22pcrel = "R_HEX_B22_PCREL";
    uint32_t relocationType =
        P->getLinker()->getRelocationHandler().getRelocationType(b22pcrel);
    P->getLinker()->registerReloc(relocationType);
  }

  // Relocation callback hook function.
  void RelocCallBack(eld::plugin::Use U) override {
    // Print relocation source section name and symbol names.
    std::string sourceSectionName = U.getSourceChunk().getName();
    std::cout << "Relocation callback. Source section: " << sourceSectionName
              << ", symbol: " << U.getName() << "\n";
    // Change relocation symbol from HelloWorld to HelloQualcomm.
    if (U.getSymbol().getName() == "HelloWorld") {
      eld::Expected<eld::plugin::Symbol> expHelloQualcommSymbol =
          P->getLinker()->getSymbol("HelloQualcomm");
      ELDEXP_REPORT_AND_RETURN_VOID_IF_ERROR(
          P->getLinker(), expHelloQualcommSymbol);
      eld::plugin::Symbol helloQualcommSymbol =
          std::move(expHelloQualcommSymbol.value());
      U.resetSymbol(helloQualcommSymbol);
    }
  }

private:
  ModifyRelocations *P;
};

eld::plugin::Plugin *ThisPlugin = nullptr;
eld::plugin::LinkerPluginConfig *ThisPluginConfig = nullptr;

extern "C" {
// RegisterAll should initialize all the plugins and plugin configs that a
// plugin library aims to provide. Linker calls this function before running
// any plugins provided by the library.
bool DLL_A_EXPORT RegisterAll() {
  ThisPlugin = new ModifyRelocations();
  ThisPluginConfig = new ModifyRelocationsPluginConfig(
      dynamic_cast<ModifyRelocations *>(ThisPlugin));
  return true;
}

// Linker calls this function to request an instance of the plugin
// with the plugin name pluginName. pluginName is provided in the plugin
// invocation command.
eld::plugin::Plugin *getPlugin(const char *pluginName) { return ThisPlugin; }

// Linker calls this function to request an instance of the plugin
// configuration for the plugin with the plugin name pluginName.
// pluginName is provided in the plugin invocation command.
eld::plugin::LinkerPluginConfig DLL_A_EXPORT *getPluginConfig(const char *pluginName) {
  return ThisPluginConfig;
```

```
}

// Cleanup should free all the resources owned by a plugin library.
// Linker calls this function after all runs of the plugins provided
// by the library have completed.
void DLL_A_EXPORT Cleanup() {
  if (ThisPlugin)
    delete ThisPlugin;
  if (ThisPluginConfig)
    delete ThisPluginConfig;
}
}
```

To build the plugin, run:

```
clang++-14 -o libModifyRelocations.so ModifyRelocations.cpp -std=c++17 -stdlib=libc++ -fPIC -shared \
-I${HEXAGON}/include -L${HEXAGON}/lib -lLW
```

Now, let's see the effect of this plugin on a sample program.

1.c

```
#include <stdio.h>

const char *HelloWorld() {
  return "Hello World!";
}

const char *HelloQualcomm() {
  return "Hello Qualcomm!";
}

int main() {
  printf("%s\n", HelloWorld());
}
```

1.linker.script

```
PLUGIN_ITER_SECTIONS("ModifyRelocations", "ModifyRelocations");
```

Now, let's build 1.c with *ModifyRelocations* plugin enabled.

```
export LD_LIBRARY_PATH="/local/mnt/workspace/partaror/llvm-project-formal/obj/bin:${LD_LIBRARY_PATH}"
hexagon-clang -o 1.elf 1.c -ffunction-sections -fdata-sections -Wl,-T,1.linker.script
```

Running the above commands gives the below output:

```
Relocation callback. Source section: .text, symbol: .start
...
...
Relocation callback. Source section: .text.main, symbol: HelloWorld
Relocation callback. Source section: .text.main, symbol: printf
...
```

This output is emitted by the plugin when it's iterating over relocations of registered types.

Now, let's try running the generated binary image, *1.elf*:

```
$ hexagon-sim ./1.elf

hexagon-sim INFO: The rev_id used in the simulation is 0x00008d68 (v68n_1024)
Hello Qualcomm!

Done!
        T0: Insns=5479 Packets=2946
        T1: Insns=0 Packets=0
```

```
        T2: Insns=0 Packets=0
        T3: Insns=0 Packets=0
        T4: Insns=0 Packets=0
        T5: Insns=0 Packets=0
        Total: Insns=5479 Pcycles=8841
```

Hello World! has been replaced by *Hello Qualcomm!*. It's the result of plugin changing the relocation symbol from `HelloWorld` to `HelloQualcomm` for all `R_HEX_B22_PCREL` relocations.

Section Rule-Matching

This example plugin describes how to modify section rule-matching using a plugin. A plugin can create section overrides using the `LinkerWrapper::setOutputSection` API. Section overrides created by a plugin override linker script section rule-matching. Plugins must call `LinkerWrapper::finishAssignOutputSections` after all section overrides have been created. `LinkerWrapper::finishAssignOutputSections` brings the section override change into effect.

Section overrides must only be used in the *BeforeLayout* link state. After the *BeforeLayout* state, chunks from input sections get merged into output sections, making section overrides meaningless.

The *ChangeOutputSection* plugin sets the output section of `.text.foo` to `bar`. That's it. Let's see how to create this plugin.

ChangeOutputSection plugin with self-contained documentation.

```
#include "ELD/PluginAPI/PluginVersion.h"
#include "ELD/PluginAPI/SectionIteratorPlugin.h"
#include <iostream>

class ChangeOutputSection : public eld::plugin::SectionIteratorPlugin {
public:
  ChangeOutputSection() : eld::plugin::SectionIteratorPlugin("ChangeOutputSection") {}

  // 'Init' callback hook can be used for initialization and preparations.
  // This plugin does not need any initialization or preparation.
  void Init(std::string cfg) override {}

  // 'processSection' callback hook of SectionIteratorPlugin is called for
  // each non-garbage collected section.
  void processSection(eld::plugin::Section S) override {
    if (S.matchPattern("*foo")) {
      // Changes the output section of the section S to
      // bar. LinkerWrapper::setOutputSection must only be
      // called in BeforeLayout link state. Section overrides created
      // after BeforeLayout link state do not work and can result in
      // undefined behavior.
      //
      // Annotation is useful for diagnostic purposes. Later, we will see where
      // to find these annotations.
      Linker->setOutputSection(
          S, "bar",
          /*Annotation=*/"Setting output section of '.text.foo' to 'bar'");
    }
  }

  // 'Run' callback hook is called after all the 'processSection' callback hook
  // calls. It is called once for each section iterator plugin run.
  eld::plugin::Plugin::Status Run(bool trace) override {
    return eld::plugin::Plugin::Status::SUCCESS;
  }

  // 'Destroy' callback hook can be used for finalization and clean-up tasks.
```

```
  // It is called once for each section iterator plugin run.
  void Destroy() override {
    // LinkerWrapper::finishAssignOutputSections must be called
    // after all section overrides have been created by the plugin.
    // It brings the created section overrides into effect.
    Linker->finishAssignOutputSections();
  }

  uint32_t GetLastError() override { return 0; }

  std::string GetLastErrorAsString() override { return "Success"; }

  std::string GetName() override { return "ChangeOutputSection"; }
};

eld::plugin::Plugin *ThisPlugin = nullptr;

extern "C" {
// RegisterAll should initialize all the plugins that a plugin library aims
// to provide. Linker calls this function before running any plugins provided
// by the library.
bool RegisterAll() {
  ThisPlugin = new ChangeOutputSection{};
  return true;
}

// Linker calls this function to request an instance of the plugin
// with the plugin name pluginName. pluginName is provided in the plugin
// invocation command.
eld::plugin::Plugin *getPlugin(const char *pluginName) { return ThisPlugin; }

// Cleanup should free all the resources owned by a plugin library.
// Linker calls this function after all runs of the plugins provided
// by the library have completed.
void Cleanup() {
  if (!ThisPlugin)
    return;
  delete ThisPlugin;
  ThisPlugin = nullptr;
}
}
```

To build the plugin, run the following command:

```
clang++-14 -o libChangeOutputSection.so ChangeOutputSection.cpp -std=c++17 -stdlib=libc++ -fPIC -shared \
-I${HEXAGON}/include -L${HEXAGON}/lib -lLW
```

1.c

```
int foo() { return 1; }
int bar() { return 2; }

int main() { return 0; }
```

1.linker.script

```
PLUGIN_ITER_SECTIONS("ChangeOutputSection", "ChangeOutputSection");

SECTIONS {
  foo : { *(.text.foo) }
  bar : { *(.text.bar) }
  text : { *(.text*) }
}
```

Now, let's build 1.c with the *ChangeOutputSection* plugin enabled.

```
export LD_LIBRARY_PATH="${HEXAGON}/lib:${LD_LIBRARY_PATH}"
hexagon-clang -o 1.o 1.c -c -ffunction-sections -fdata-sections
hexagon-link -o 1.elf 1.o -T 1.linker.script -Map 1.map.txt
```

Now, let's see the output section of `foo`.

```
hexagon-readelf -Ss 1.elf
```

hexagon-readelf output:

```
There are 7 section headers, starting at offset 0x1200:

Section Headers:
  [Nr] Name              Type            Address  Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] bar               PROGBITS        00000000 001000 00001c 00  AX  0   0 16
  [ 2] text              PROGBITS        00000020 001020 000014 00  AX  0   0 16
  [ 3] .comment          PROGBITS        00000000 001034 0000d2 01  MS  0   0  1
  [ 4] .shstrtab         STRTAB          00000000 001106 00002d 00      0   0  1
  [ 5] .symtab           SYMTAB          00000000 001134 000090 10      6   5  4
  [ 6] .strtab           STRTAB          00000000 0011c4 00002a 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  R (retain), p (processor specific)

Symbol table '.symtab' contains 9 entries:
   Num:    Value  Size Type    Bind   Vis       Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT   UND
     1: 00000000     0 SECTION LOCAL  DEFAULT     1 bar
     2: 00000000     0 SECTION LOCAL  DEFAULT     3 .comment
     3: 00000020     0 SECTION LOCAL  DEFAULT     2 text
     4: 00000000     0 FILE    LOCAL  DEFAULT   ABS 1.c
     5: 00000000    12 FUNC    GLOBAL DEFAULT     1 bar
     6: 00000010    12 FUNC    GLOBAL DEFAULT     1 foo
     7: 00000020    20 FUNC    GLOBAL DEFAULT     2 main
     8: 00000039     0 NOTYPE  GLOBAL DEFAULT   ABS __end
```

You can observe from the readelf output that the `bar` section contains the `foo` symbol. This means the output section of `.text.foo` was indeed changed to `bar`.

That's all well and good, but how do we see the nice annotation we added when calling `LinkerWrapper::setOutputSection`? All plugin actions, along with their annotations, are recorded in the map file. The text map file can be used to quickly view plugin actions and the annotations.

To see all the plugin action information, you can either view the text map file and find the "Detailed Plugin information" component, or run the command below to directly view the information from the text map file.

```
less +/"Detailed Plugin information" 1.map.txt
```

Output

```
# Detailed Plugin information
# Plugin #0     ChangeOutputSection
#       Modification #1 {C, Comment : Setting output section of '.text.foo' to 'bar'}
#       Section :.text.foo      1.o
#       Original Rule : *(.text.foo) #Rule 1, 1.linker.script
#       Modified Rule : *(bar) #Rule 4, Internal-LinkerScript (Implicit rule inserted by Linker)
```

Change Symbol Value to Another Symbol

The *ChangeSymbolValue* plugin demonstrates how to change the value of a symbol to another symbol. Specifically, it changes the value of the `HelloWorld` symbol to the value of the `HelloQualcomm` symbol. It also attempts (unsuccessfully) to change the value of the `HelloWorldAgain` symbol to the value of the `HelloQualcommAgain` symbol. We will see why this happens.

Now, let's see how to create the *ChangeSymbolValue* plugin.

The *ChangeSymbolValue* plugin with self-contained documentation.

```
#include "ELD/PluginAPI/OutputSectionIteratorPlugin.h"
#include "ELD/PluginAPI/PluginVersion.h"
#include <iostream>

class ChangeSymbolValue : public eld::plugin::OutputSectionIteratorPlugin {
public:
  ChangeSymbolValue()
      : eld::plugin::OutputSectionIteratorPlugin("ChangeSymbolValue") {}

  // 'Init' callback hook can be used for initialization and preparations.
  // This plugin does not need any initialization or preparation.
  void Init(std::string cfg) override {}

  // 'processOutputSection' callback hook is called once for each output
  // section.
  void processOutputSection(eld::plugin::OutputSection O) override {}

  // 'Run' callback hook is called after all the 'processSection' callback hook
  // calls.
  eld::plugin::Plugin::Status Run(bool trace) override {
    if (Linker->getState() != eld::plugin::LinkerWrapper::State::AfterLayout)
      return eld::plugin::Plugin::Status::SUCCESS;
    // Try to reset the 'HelloWorld' symbol value to the value of
    // 'HelloQualcomm' symbol.
    eld::plugin::Symbol helloWorldSymbol = Linker->getSymbol("HelloWorld");
    eld::plugin::Symbol helloQualcommSymbol = Linker->getSymbol("HelloQualcomm");
    bool resetSym =
        Linker->resetSymbol(helloWorldSymbol, helloQualcommSymbol.getChunk());
    if (resetSym)
      std::cout
          << "'HelloWorld' symbol value has been successfully reset to the "
          << "value of 'HelloQualcomm' symbol.\n";
    else
      std::cout << "Symbol value resetting failed for 'HelloWorld'.\n";

    // Try to reset the 'HelloWorldAgain' symbol value to the value of
    // 'HelloQualcommAgain' symbol.
    eld::plugin::Symbol helloWorldAgainSymbol = Linker->getSymbol("HelloWorldAgain");
    eld::plugin::Symbol helloQualcommAgainSymbol =
        Linker->getSymbol("HelloQualcommAgain");
    resetSym = Linker->resetSymbol(helloWorldAgainSymbol,
                                   helloQualcommAgainSymbol.getChunk());
    if (resetSym)
      std::cout << "'HelloWorldAgain' symbol value has been successfully reset "
                << "to the "
                << "value of 'HelloQualcommAgain' symbol.\n";
    else
      std::cout << "Symbol value resetting failed for 'HelloWorldAgain'.\n";
    return eld::plugin::Plugin::Status::SUCCESS;
  }
```

```
  void Destroy() override {}

  uint32_t GetLastError() override { return 0; }

  std::string GetLastErrorAsString() override { return "Success"; }

  std::string GetName() override { return "ChangeSymbol"; }
};

eld::plugin::Plugin *ThisPlugin = nullptr;

extern "C" {
// RegisterAll should initialize all the plugins that a plugin library aims
// to provide. Linker calls this function before running any plugins provided
// by the library.
bool RegisterAll() {
  ThisPlugin = new ChangeSymbolValue{};
  return true;
}

// Linker calls this function to request an instance of the plugin
// with the plugin name pluginName. pluginName is provided in the plugin
// invocation command.
eld::plugin::Plugin *getPlugin(const char *pluginName) { return ThisPlugin; }

// Cleanup should free all the resources owned by a plugin library.
// Linker calls this function after all runs of the plugins provided
// by the library have completed.
void Cleanup() {
  if (!ThisPlugin)
    return;
  delete ThisPlugin;
  ThisPlugin = nullptr;
}
}
```

To build the plugin, run:

```
clang++-14 -o libChangeSymbolValue.so ChangeSymbolValue.cpp -std=c++17 -stdlib=libc++ -fPIC -shared \
-I${HEXAGON}/include -L${HEXAGON}/lib -lLW
```

Now, let's see the effect of this plugin on a sample program.

1.c

```
#include <stdio.h>

const char *HelloWorld;
const char *HelloQualcomm = "Hello Qualcomm!";

const char *HelloWorldAgain = "Hello again World!";
const char *HelloQualcommAgain = "Hello again Qualcomm!";

int main() {
  printf("%s\n", HelloWorld);
  printf("%s\n", HelloWorldAgain);
}
```

1.linker.script

```
PLUGIN_OUTPUT_SECTION_ITER("ChangeSymbolValue", "ChangeSymbolValue");
```

Now, let's build 1.c with *ChangeSymbolValue* plugin enabled.

```
export LD_LIBRARY_PATH="${HEXAGON}/lib:${LD_LIBRARY_PATH}"
hexagon-clang -o 1.elf 1.c -ffunction-sections -fdata-sections -Wl,-T,1.linker.script
```

Running the above commands produces the following output:

```
'HelloWorld' symbol value has been successfully reset to the value of 'HelloQualcomm' symbol.
Symbol value resetting failed for 'HelloWorldAgain'.
```

The above output is printed by the plugin. It indicates whether the resetting of symbol values was successful or not. So, why was the resetting of 'HelloWorldAgain' not successful? `LinkerWrapper::resetSymbol` can only resets the value of the symbols that do not already have a value. `LinkerWrapper::resetSymbol` silently fails and returns false if the symbol requested to be reset already has a value.

Now, let's run the generated **1.elf** binary:

```
$ hexagon-sim ./1.elf

hexagon-sim INFO: The rev_id used in the simulation is 0x00008d68 (v68n_1024)
Hello Qualcomm!
Hello again World!

Done!
        T0: Insns=6810 Packets=3558
        T1: Insns=0 Packets=0
        T2: Insns=0 Packets=0
        T3: Insns=0 Packets=0
        T4: Insns=0 Packets=0
        T5: Insns=0 Packets=0
        Total: Insns=6810 Pcycles=10677
```

As expected, the output first prints, 'Hello Qualcomm!', as the value of the `HelloWorld'` symbol successfully got reset to the value of `HelloQualcomm`. The output then prints, 'Hello again World!', as the resetting of `HelloWorldAgain'` to `HelloQualcommAgain'` was unsuccessful.

Linker Plugin API Usage and Reference

Linker Wrapper

Plugins interacts with the linker using LinkerWrapper.

Plugin Abstract Data Types

Chunk

LinkerScriptRule

OutputSection

Section

Use

Symbol

INIFile

InputFile

PluginData

AutoTimer

Timer

RelocationHandler

LinkerPluginConfig

Utility Data Structures

JSON Objects

TarWriter

ThreadPool

Plugin Diagnostic Framework

Plugin diagnostic framework provides a uniform and consistent solution for reporting diagnostics from plugins. It is recommended to avoid custom solutions for printing diagnostics from a plugin.

First, let's go over some of the key benefits of using the Plugin Diagnostic Framework. After that, we'll see how to effectively use the framework.

The framework allows for creating and reporting diagnositcs on the fly. Diagnostics can be reported with different severities such as *Note*, *Warning*, *Error* and more. Each severity has a unique color, and diagnostics are prefixed with "PluginName:DiagnosticSeverity".

The linker keeps track of diagnostics emitted from the diagnostic framework. This is required to maintain extensive logs for diagnostic purposes. The framework also adjusts diagnostic behavior based on linker command-line options that affect diagnostics. For example, if the `–fatal-warnings` option is enabled, then warnings will be converted to fatal errors. If the `-Werror` option is used, warnings are promoted to regular errors instead of fatals.

The framework is thread-safe. Thus, diagnostics can be reported safely from multiple threads simultaneously. This is crucial in situations where thread-safety is neessary. In such cases, C++ `std::cout` and `std::cerr` output streams cannot be used, as they are not thread-safe.

That's enough about the theories. Now, let's take a look at how to use the Plugin Diagnostic Framework.

The linker assigns a unique ID to each diagnostic template. A diagnostic template has a diagnostic severity and a diagnostic format string. Diagnostic template unique IDs are necessary for creating and reporting diagnostics. The code below demonstrates how to obtain a diagnostic ID for a diagnostic template.

```
// There are similar functions for other diagnostic severities.
// Linker is a LinkerWrapper object.
DiagnosticEntry::DiagIDType errorID = Linker->getErrorDiagID("Error diagnostic with two args: %0 and %1");
DiagnosticEntry::DiagIDType warnID = Linker->getWarningDiagID("Warning diagnostic with one arg: %0");
```

`%0`, `%1` and so on in diagnostic format string are replaced by diagnostic arguments. Diagnostic arguments must be provided when reporting a diagnostic.

The below code demonstrates how to report a diagnostic using a diagnostic ID.

```
// arguments can be of any type. 'LinkerWrapper::reportDiag' is a
// variadic template function.
Linker->reportDiag(errorID, arg1, arg2);
Linker->reportDiag(warnID, arg1);
```

DiagnosticEntry

`DiagnosticEntry` packs complete diagnostic information, and can be used to pass diagnostics from one location to another. Many plugin framework APIs use `DiagnosticEntry` to return errors to the caller, where they can be properly handled.

```
// diag represents a complete diagnostic.
// diagID encodes diagnostic severity and diagnostic format string.
DiagnosticEntry diag(diagID, {arg1, arg2, ...});
```

eld::Expected<ReturnType>

Many plugin framework APIs return values of type `eld::Expected<ReturnType>`. At any given time, `eld::Expected<ReturnType>` holds either an expected value of type `ReturnType` or an unexpected value of type `std::unique_ptr<eld::plugin::DiagnosticEntry>`. Returning the error to the caller allows plugin authors to decide how to best handle a particular error for their use case. A typical usage pattern for this is.

```
eld::Expected<eld::plugin::INIFile> readFile = Linker->readINIFile(configPath);
if (!readFile) {
  if (readFile.error()->diagID() == eld::plugin::Diagnostic::errr_file_does_not_exit) {
    // handle this particular error
    // ...
  } else {
    // Or, simply report the error and return.
    Linker->reportDiagEntry(std::move(readFile.error()));
    return;
  }
}

eld::plugin::INIFile file = std::move(readFile.value());
```

Overriding Diagnostic Severity

Diagnostic severity can be overriden by using `DiagnosticEntry` subclasses. `DiagnosticEntry` has subclasses for each diagnostic severity level. Let's explore how to use them to override diagnostic severity.

```
eld::Expected<eld::plugin::INIFile> readFile = Linker->readINIFile(configPath);
eld::plugin::INIFile file;
if (readFile)
  file = std::move(readFile.value());
else {
```

```
  // Let's report the error as a Note, and move on by using a default INI file.
  eld::plugin::NoteDiagnositcEntry noteDiag(readFile.error()->diagID(), readFile.error()->args());
  Linker->reportDiagEntry(noteDiag);
  file = DefaultINIFile();
}
```

Diagnostic Framework types API reference

Linker Plugins

Overview

- The ELD allows one or more linker plugins to be loaded and called during link time.
- Develop the linker plug-ins to query the linker about objects being linked during link time, and to evaluate their properties.
- You can also change the order and contents of output sections using this approach.
- Use C++ to develop the linker plugins.
- They are built as shared libraries that allow the linker to dynamically load at link time.
- The plugins use a fixed API that allows you to communicate with the linker.
- Plugins are currently supported on all architectures where the functionality is available.
- The Map file records any changes made by the plugin for the layout. This is a good place to determine how the layout was affected due to the plug-in algorithm.

Plugin Usage

- Most developers use linker scripts, and the linker plugin functionality can therefore be exercised using linker scripts.
- This is to promote ease of use and have less maintenance overhead.

- It is also easier to integrate with existing builds.
- The linker can also provide better diagnostics.

Linker wrapper

- Build a plugin as a dynamic library and interact with the linker using an opaque LinkerWrapper handle.
- The linker wrapper exists as a shared library that you link with the user plugin.
- The linkerwrapper is named **libLW.so** on Linux platforms and **LW.dll** on Windows platforms.

User Plugin Types

- The intent of each user-developed plugin is associated with an appropriate interface.
- You can implement the plugin algorithm by deriving it from one such interface.
- You can use more than one interface in a single plugin.
- Plugin chaining can be used interchangeably to denote when you are mixing more than one interface in a single plug-in.
- The rest of this section might use PluginType interchangeably to denote this.
- Currently, four interface types are available. More interface types will be added based on new use cases in the future.

Interface type	Header file
Section iterator plugin type	SectionIteratorPlugin.h
Chunk iterator plugin type	ChunkIteratorPlugin.h
Control memory size plugin type	ControlMemorySizePlugin.h
Control file size plugin type	ControlFileSizePlugin.h

LinkerScript changes

Linker plugins can be enabled using linker scripts. The linker script keyword uses a fixed syntax:

```
<PluginType>("LibraryName", "PluginName" [, "PluginOptions"])
```

- **PluginType**

  Corresponds to one of the available interface types.

- **LibraryName**

  - Corresponds to the dynamic library that contains the plugin for the linker to load.
  - Uses the same linker semantics for linking to the linker in a library, allowing ease of use.
  - Uses the name of the library without the lib prefix on Linux and without the .so/.dll suffix on Linux/Windows, respectively

- **PluginName**

  Specifies the name of the plugin. The linker queries the dynamic library to provide an implementation for the specified interface type.

- **PluginOptions**

  Used to pass an option to the plug-in.

User Plugin Work Flow

Use the following steps to create a linker plugin.

1. Determine the appropriate interface(s).
2. Include the appropriate header file.
3. Create a C++ class derived from one of the interface types.
4. Associate the implementation of the interface to have a unique name.
5. Build a shared library.
6. Make the shared library report Plugin API version.
7. Write a RegisterAll function to register the plugin(s).
8. Write a getPlugin function to return the appropriate plugin when the linker queries with PluginName.

Linker Work Flow

- All user plugins that are loaded by the linker must be initialized properly before the plugin can communicate with the linker and perform the steps.

Linker Operation

The linker performs the following sequence of operations with respect to plugins:

1. Parses the linker script and finds all plug-ins.
2. **Loads the library specified by LibraryName.**
    a. Uses LD_LIBRARY_PATH on Linux.
    b. Uses a standard method for searching dynamic libraries in Windows.
3. Calls the RegisterAll() API in the library, which registers all the plug-ins that are contained in the library.
4. **Queries the library using the getPlugin() API with PluginName.**
    The library returns the appropriate object for the linker to use to run the plugin algorithm.
5. Inspects the plug-in to verify that the linker script keyword and the object have the same interface type.
6. Initializes the plug-in with any additional options provided.
7. Passes the appropriate content to the plug-in expressed as data structures for that PluginType.
8. **Runs the plug-in algorithm.**
    Because more than one plug-in can be used and because plug-ins can be chained, the linker unloads the library only after all user plug-ins have been called.
9. **Before the plug-in is unloaded, the linker calls a function to Destroy the plug-in.**
    This is the last step before the library is unloaded.
10. The linker then unloads the plug-in.

- **Plugin tracing**
    Trace the linker work flow with the following option:

    ```
    -trace=plugin
    ```

Plugin data structures

- Appropriate data structures are exchanged depending on the specified plug-in interface type.
- These data structures are also used to communicate with the linker and get appropriate information from the linker.

Section

Corresponds to an input section from an ELF file.

OutputSection

Corresponds to an OutputSection section in the LinkerScript or output ELF file.

Chunk

Corresponds to a piece of an input section. Examples: * Individual strings of a section that contains merged strings * Contents of an output section

Block

Corresponds to the content of the output section with corrected relocations.

Symbol

Corresponds to an ELF symbol.

Use

Corresponds to a relocation from a chunk or section.

LinkerScriptRule

Corresponds to a LinkerScriptRule in an OutputSection

LinkerWrapper commands

The plug-in uses the LinkerWrapper to communicate and exchange information with the linker. Following are the LinkerWrapper commands.

getVersion

Returns the version of the LinkerWrapper as a string. This command is useful for diagnostics.

AllocateMemory

- Allocates memory that must live for the duration of the link.
- The ControlMemorySize and ControlFileSize interface types are the most common users of this functionality.

getUses(Chunk)

Queries the linker to find out what a Chunk refers to. The API returns a vector of uses.

getUses(Section)

Queries the linker to find out what a Section refers to. The API returns a vector of uses.

getSymbol(SymbolName)

Gets more information from the linker for a symbol (specified by SymbolName).

getOutputSection(Section)

- Determines which OutputSection was chosen by the linker.
- The OutputSection is usually chosen by the linker by matching rules in the linker script.

setOutputSection(Section, OutputSectionName)

- Places the Section into the specified OutputSection in the linker script.
- This command allows linker script decisions to be overridden for that particular Section.

MatchPattern(Pattern, Name)

- Utility function that allows the plug-in to match a Glob Pattern with a string.

setLinkerFatalError

Used by the plug-in when it discovers that there is an unhandled case when trying to link input files to produce an output image.

resetError

Resets any error status

Note

Use the PluginADT.h file to learn about the APIs documented in the header file.

Linker plugin interface

Following is the plugin interface that the linker will use and provide implementation for the functions:

```
class Plugin {
public:
/* Initialize the plugin with options specified */
virtual void Init(std::string Options) = 0;
/* The actual algorithm that will be implemented */
virtual Status Run (bool Verbose)= 0;
/* Linker will call Destroy, and the client can free up any data
structures that are not relevant */
virtual void Destroy()= 0;
/* Returns the last error; a value of 0 means there was no error */
virtual uint32_t GetLastError()= 0;
/* Returns the error as a string */
virtual std::string GetLastErrorAsString ()= 0;
/* Returns the name of the plugin */
virtual std::string GetName()= 0;
/* Destructor */
virtual -Plugin(){}
};
```

Plugin interfaces

The following interfaces are available currently.

SectionIterator interface

- Use the linker script keyword, **PLUGIN_ITER_SECTIONS**.
- This interface allows you to process every input section from every object file by implementing processSection().

```
class SectionIteratorPlugin : public Plugin {
public:
/* Section that the linker will call the client with */
virtual void processSection(Section S)= 0;
};
```

SectionMatcher interface

- Use the linker script keyword, **PLUGIN_SECTION_MATCHER**.
- This interface allows you to process every input section, and it allows the plugin to read any section.
- You can use the interface to read metadata sections or read sections that were garbage collected by the linker.
- SectionMatcherPlugin differs from SectionIteratorPlugin in that it allows any section content to be read before assigning output sections.

```
class SectionMatcherPlugin : public Plugin {
public:
```

```
/* Sections that the linker will call the client with */
virtual void processSection(Section S) = 0;
};
```

ChunkIterator interface

- Use the linker script keyword, **PLUGIN_ITER_CHUNKS**.
- This interface allows you to process every input chunk in an output section using **PocessChunk()**.
- The processed chunks are returned when the linker calls **getChunks()**.

```
class ChunkIteratorPlugin : public Plugin {
public:
/* Chunks that the linker will call the client with */
virtual void processChunk(Chunk C) = 0;
virtual std::vector<Chunk> getChunks()= 0;
};
```

ControlFileSize interface

- Use the linker script keyword, **PLUGIN_CONTROL FILESZ**.
- This interface allows you to process the output memory block contained in an output section using **AddBlocks()**.
- The processed block is returned when the linker calls **GetBlocks()**.
- An example of such a plugin is to take the memory block, compress it, and then return it to the linker.

```
class ControlFileSizePlugin : public Plugin {
public:
/* Memory blocks that the linker will call the client with */
virtual void AddBlocks(Block memBlock) = 0;
/* Return memory blocks to the client */
virtual std::vector<Block> GetBlocks()= 0;
};
```

ControlMemorySize interface

- Use the linker script keyword, **PLUGIN_CONTROL MEMSZ**.
- This interface allows you to process the output memory block contained in an output section using **AddBlocks()**.
- The processed block is returned when the linker calls **GetBlocks()**.

```
class ControlMemorySizePlugin : public Plugin {
public:
/* Memory blocks that the linker will call the client with */
virtual void AddBlocks(Block memBlock) = 0;
/* Return memory blocks to the client */
virtual std::vector<Block> GetBlocks()= 0;
};
```

OutputSectionIterator interface

- Use the linker script keyword, **PLUGIN_OUTPUT_SECTION_ITER**.
- The interface is defined as below.

```
class OutputSectionIteratorPlugin : public Plugin { public:
/* OutputSection that the linker will call the client with */
virtual void processOutputSection(OutputSection O)= 0;
};
```

- The interface allows the plugin to iterate over all output sections and their contents.
- Contents of output sections are described by Rules.
- **The OutputSectionIterator is called at various times during the linking stages:**
    - BeforeLayout
    - CreatingSections
    - AfterLayout
- The state of the linker can be queried by calling a function **getState** in the LinkerWrapper.
- Depending on the state of the linker, the plug-in can query OutputSections and its contents appropriately.
- The OutputSectionIterator interface allows the plug-in to iterate over rules specified in the OutputSection and what sections are essentially contained in a LinkerScriptRule.
- Depending on the state of the Linker, the contents of the LinkerScriptRule can be modified.
- Modifications of LinkerScriptRule contents include changing the OutputSection for a Section, changing the OutputSection for a Chunk

BeforeLayout

- When the state of the linker is set to BeforeLayout, the plug-in can query each rule for its contents.
- The contents of the Rule can only be Sections.
- Call the setOutputSection function in the LinkerWrapper to set the Section point to a different OutputSection.
- Finish with all the assignments to different output sections by calling
- finishAssignOutputSections in the LinkerWrapper.

CreatingSections

- When the state of the linker is set to CreatingSections, the plug-in can query each rule for its contents. The contents of the Rule can only be Chunks.
- **Call APIs specified in the LinkerScriptRule to do either of the following:**
    - addChunk – Add a Chunk to a LinkerScriptRule
    - removeChunk – Remove a Chunk from a LinkerScriptRule.
    - updateChunk – Replace Chunks in a LinkerScriptRule.

AfterLayout

When the state of the linker is set to AfterLayout, the plugin can query for the size of the OutputSection.

Linker Optimization Features

- **Plugins**

  More Information about linker plugins can be found at Linker Plugins
- Garbage Collection
    - Unused function and data sections are garbage collected by linker when we provide **–gc-sections** option at the link time

- Garbage collected sections can be printed by providing **–print-gc-sections** option
- –gc-sections decides which input sections are used by examining symbols and relocations.
- The section containing the entry symbol and all sections containing symbols undefined on the command-line will be kept, as will sections containing symbols referenced by dynamic objects.
- Note that when building shared libraries, the linker must assume that any visible symbol is referenced.
- If the linker performs a partial link (-r linker option), then you will need to provide the entry point using the -e / –entry linker option.

ELF TOOLS

This sections documents llvm ELF command-line tools along with detailed examples of their usage

llvm-readelf <options> <file>

Inspect ELF files

- **-h**

  Display ELF header information:

```
$ llvm-readelf -h a.out
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           QUALCOMM DSP6 Processor
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          52 (bytes into file)
  Start of section headers:          228768 (bytes into file)
  Flags:                             0x68
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         4
  Size of section headers:           40 (bytes)
  Number of section headers:         17
  Section header string table index: 14
```

- **-S**

  List ELF Sections

```
$ llvm-readelf -S a.out
There are 17 section headers, starting at offset 0x37da0:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .start            PROGBITS        00000000 001000 0046c8 00 WAX  0   0 64
  [ 2] .init             PROGBITS        00005000 006000 000064 00  AX  0   0 32
  [ 3] .text             PROGBITS        00006000 007000 018bcc 00  AX  0   0 4096
  [ 4] .fini             PROGBITS        0001ebe0 01fbe0 000030 00  AX  0   0 32
  ...
  [11] .sdata            PROGBITS        00027000 028000 000170 00 WAp  0   0 4096
  [12] .bss              NOBITS          00027170 028170 0017d8 00  WA  0   0  8
  [13] .comment          PROGBITS        00000000 028170 000083 00  MS  0   0  1
```

```
  [14] .shstrtab         STRTAB          00000000 0281f3 000081 00      0   0  1
  [15] .symtab           SYMTAB          00000000 028274 006d50 10     16 530  4
  [16] .strtab           STRTAB          00000000 02efc4 008dbd 00      0   0  1
```

- **-l**

  List ELF segments:

```
$ llvm-readelf -l a.out

Elf file type is EXEC (Executable file)
Entry point 0x0
There are 4 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x001000 0x00000000 0x00000000 0x046c8 0x046c8 RWE 0x1000
  LOAD           0x006000 0x00005000 0x00005000 0x1f658 0x1f658 R E 0x1000
  LOAD           0x026000 0x00025000 0x00025000 0x02170 0x03948 RW  0x1000
  GNU_RELRO      0x026000 0x00025000 0x00025000 0x00044 0x00044 RW  0x4

Section to Segment mapping:
  Segment Sections...
  00     .start
  01     .init .text .fini .rodata .eh_frame .gcc_except_table
  02     .ctors .dtors .data .sdata .bss
  03     .ctors .dtors
```

- **-s**

  Display the symbol table

```
$ llvm-readelf -s hello.o

  Symbol table '.symtab' contains 324 entries:
    Num:    Value  Size Type    Bind   Vis      Ndx Name
      0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
      1: 00000000     0 FILE    LOCAL  DEFAULT  ABS hello.cc
      2: 00000090     0 NOTYPE  LOCAL  DEFAULT    7 GCC_except_table10
      3: 00000230     0 NOTYPE  LOCAL  DEFAULT    7 GCC_except_table100
      4: 00000240     0 NOTYPE  LOCAL  DEFAULT    7 GCC_except_table121
      5: 00000268     0 NOTYPE  LOCAL  DEFAULT    7 GCC_except_table128
      ...
```

- **-r** = **–relocations**

  Display relocation entries

```
$ cat 1.c
int foo() {
  printf("hello world\n");
  return 0;
}
$ clang -c 1.c
$ llvm-readelf -r 1.o
Relocation section '.rela.text' at offset 0x110 contains 3 entries:
Offset     Info    Type                Sym. Value  Symbol's Name + Addend
00000004  00000211 R_HEX_32_6_X          00000000   .rodata.str1.1 + 0
00000008  00000217 R_HEX_16_X            00000000   .rodata.str1.1 + 0
0000000c  00000401 R_HEX_B22_PCREL       00000000   printf
```

- **–string-dump** <section>

  Treat the contents of a section as a string and print it

```
$cat 1.c
int foo() {
  printf("hello world\n");
  return 0;
}
$clang 1.c -c
$llvm-readelf -S 1.o
There are 9 section headers, starting at offset 0x194:

Section Headers:
  [Nr] Name              Type            Address  Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .strtab           STRTAB          00000000 000131 000061 00      0   0  1
  [ 2] .text             PROGBITS        00000000 000040 000018 00  AX  0   0 16
  [ 3] .rela.text        RELA            00000000 00010c 000024 0c      8   2  4
  [ 4] .rodata.str1.1    PROGBITS        00000000 000058 00000d 01 AMS  0   0  1
...
$llvm-readelf --string-dump .rodata.str1.1 1.o
String dump of section '.rodata.str1.1':
[     0] hello world.
```

- **–hex-dump** <section>

  Print the contents of a section as hex

```
$cat 1.c
void foo() {printf("hello world\n")}
$clang 1.c -c
$llvm-readelf -S 1.o
There are 9 section headers, starting at offset 0x194:

Section Headers:
  [Nr] Name              Type            Address  Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .strtab           STRTAB          00000000 000131 000061 00      0   0  1
  [ 2] .text             PROGBITS        00000000 000040 000018 00  AX  0   0 16
  [ 3] .rela.text        RELA            00000000 00010c 000024 0c      8   2  4
  [ 4] .rodata.str1.1    PROGBITS        00000000 000058 00000d 01 AMS  0   0  1
...
$llvm-readelf --hex-dump .rodata.str1.1 1.o
Hex dump of section '.rodata.str1.1':
0x00000000 68656c6c 6f20776f 726c640a 00 hello world..
```

llvm-objdump <options> <file>

Disassembler

- **-d**

Show assembler contents of executable sections:

```
$ llvm-objdump -d a.out
a.out:  file format ELF32-hexagon
Disassembly of section .start:
0000000000000000 start:
0:       4c c0 00 58 5800c04c {  jump 0x98 }
4:       3e c0 00 58 5800c03e {  jump 0x80 }
8:       42 c0 00 58 5800c042 {  jump 0x8c }
...
```

- **-r**

Display relocation entries involved during linking:

```
$ llvm-objdump -d -r hello.o
hello.o:        file format ELF32-hexagon
Disassembly of section .text:
0000000000000000 __cxx_global_var_init:
       0:       01 c0 9d a0 a09dc001 {  allocframe(#8) }
       4:       00 40 00 00 00004000 {  immext(#0)
                        00000004:  R_HEX_32_6_X .bss
       8:       02 c0 00 78 7800c002    r2 = ##0 }
                        00000008:  R_HEX_16_X   .bss
       c:       00 c0 62 70 7062c000 {  r0 = r2 }
      10:       ff e2 9e a7 a79ee2ff {  memw(r30+#-4) = r2 }
      14:       00 c0 00 5a 5a00c000 {  call 0x14 }
                      00000014:  R_HEX_B22_PCREL      _ZNSt8ios_base4InitC1Ev
      ...
```

llvm-nm <options> <file>

Lists symbols in a file

- **no args**

  list regular symbols

  ```
  $cat 1.c
  int bar() { return 10; }
  int foo() { return 1 + bar(); }
  $clang -c 1.c
  $llvm-nm 1.o
  00000000 T bar
  0000000c T foo
  ```

- **-D** = **-dynamic**

  List dynamic symbols

  ```
  $cat 1.c
  int bar() { return 10; }
  int foo() { return 1 + bar(); }
  $clang 1.c -o lib.so -fPIC -shared
  $llvm-nm -D lib.so
  000003a0 T _fini
  00000240 T _init
  00000370 T bar
  0000037c T foo
  ```

- **-C**

  List symbols with demangling

llvm-objcopy <options> <input file> <output file>

Copy and translate binary files

- **no args** <input file> <output file>

  Create identical copy of one file to another, perform no translation

  if no output file is specified, input file will be modified in place

  ```
  $llvm-objcopy 1.o 1.o.copy
  ```

- **–input-target** <bfdname>, **–output-target** <bfdname>

  translate from input target to output target

  ```
  $file 1.o
  1.o: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped
  $llvm-objcopy --input-target elf32-i386 --output-target elf32-x86-64 1.o 1.o.copy
  ```

```
$file 1.o.copy
1.o.copy: ELF 32-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
```

- **–add-section** <section=file>

  insert a section into the output file with name *section* and contents of *file* for ELF files, inserted section will be of type *SHT_NOTE* if section name begins with ".note" otherwise will have type *SHT_PROGBITS*

```
$cat insert
hello world
$llvm-objcopy 1.o 1.o.copy --add-section newSection=insert
$llvm-readelf --string-dump newSection
String dump of section 'newSection':
[     0] hello world
```

llvm-ar <options>

Combine several input files into an archive library that can be linked with a program

- **-cr** <output file> <input files>

  Create an archive file containing given input files

```
$ llvm-ar -cr lib.a 1.o 2.o 3.o
$ file lib.a
lib.a: current ar archive
```

- **-t** <archive>

  list archive objects in an archive file

```
$ llvm-ar -cr lib.a 1.o 2.o 3.o
$ llvm-ar -t lib.a
1.o
2.o
3.o
```

- **-x** <archive> <files>

  extract files from archive

```
$ ls
lib.a
$ llvm-ar -t lib.a
1.o
2.o
3.o
$ llvm-ar -x lib.a 1.o 2.o
$ ls
1.o 2.o lib.a
```

llvm-strings <options> <files>

Dump printable strings from binary files to stdout

no-args <files>

Dump printable strings from input file(s) to stdout. If no file is specified, stdin is used

```
$ cat 1.c
$ void foo() {printf("hello world\n");}
$ clang 1.c -c
hello world
clang version 8.0.0 (tags/RELEASE_800/final)
.rela.text
.comment
.note.GNU-stack
.llvm_addrsig
```

```
printf
.rela.eh_frame
.strtab
.symtab
.rodata.str1.1
```

–print-file-name <files>

Display the file containing the string before each string

```
$llvm-strings 1.o a.out
1.o: hello world
...
1.o: .strtab
1.o: .symtab
1.o: .rodata.str1.1
a.out: /lib64/ld-linux-x86-64.so.2
a.out: libc.so.6
a.out: printf
a.out: __libc_start_main
...
a.out: __frame_dummy_init_array_entry
```

LTO Support

Overview

ELD supports Link Time Optimization (LTO) using the LLVM LTO libraries. The linker can consume LLVM bitcode directly or use embedded bitcode inside ELF objects and drive Full LTO or ThinLTO code generation. This chapter describes the LTO model used by ELD and the linker switches that control it.

Background: LLVM LTO

LLVM LTO performs intermodular (whole program) optimization at link time. Compared to traditional per-object optimization, LTO enables cross-module inlining, dead code elimination, and whole-program analysis by merging or summarizing LLVM IR across inputs. The LLVM LTO design emphasizes transparency in the build system: compile with LTO enabled and link with an LTO-capable linker, and the linker handles optimization without special build steps.

ThinLTO is LLVM's scalable LTO mode. It uses a summary index to avoid a full merge of all IR, supports parallel code generation, and allows caching of imported summaries or produced objects to speed up incremental builds. ELD provides knobs for ThinLTO parallelism and caching, alongside Full LTO support when modules are compiled for full LTO.

ELD LTO model

Inputs

ELD triggers LTO when it sees bitcode inputs or when it is instructed to use embedded bitcode sections in ELF files:

- **Bitcode inputs**: *.bc* inputs are always treated as LTO inputs.
- **Embedded bitcode**: ELD recognizes the *.llvmbc* section in ELF objects and can replace the native object with the embedded bitcode when LTO is enabled. This supports "fat" objects that carry both native code and LLVM bitcode.

Selecting LTO inputs

LTO is enabled in two ways:

- *-flto* enables LTO if any bitcode input or embedded bitcode is present.
- *–include-lto-filelist* can be used without *-flto* to select which ELF inputs with embedded bitcode should be upgraded to LTO inputs.

When *-flto* is used, embedded bitcode is used by default unless excluded using *–exclude-lto-filelist*. When *-flto* is not used, only file patterns listed in *–include-lto-filelist* are upgraded to LTO inputs.

File lists accept one glob pattern per line (wildcards *, *?*, and *[]* are supported). Patterns are matched against the input path as seen by the linker.

LTO phases in ELD

ELD performs LTO in distinct phases:

1. **Pre-LTO**: Inputs are read, symbols are collected, and linker scripts are evaluated. Bitcode inputs are registered for LTO.
2. **LTO**: LLVM performs symbol resolution across bitcode modules and emits native object files (Full LTO) or partitioned outputs (ThinLTO).
3. **Post-LTO**: The generated objects are inserted back into the link and linked like normal ELF objects. Map files can include pre-LTO and post-LTO details.

Linker scripts and LTO

ELD supports LTO with linker scripts. When a script uses *SECTIONS* ordering, ELD preserves input order for LTO-generated sections. If you need to disable this ordering for performance or to match non-script ordering, use *-flto-options=disable-linkorder*.

LTO switch reference

This section lists the LTO-related switches supported by ELD. Some switches also accept *–plugin-opt=…* aliases for compatibility with LLVMgold and lld.

Core LTO enablement and input selection

- *-flto* or *–flto* Enable LTO if a bitcode file or embedded bitcode section is present.
- *–include-lto-filelist=<list>* Use this list to select which ELF inputs with embedded bitcode should be used for LTO when *-flto* is not present.
- *–exclude-lto-filelist=<list>* Use this list to exclude embedded bitcode inputs when *-flto* is present.

LTO output control and artifacts

- *–save-temps* Save intermediate LTO artifacts. Temporary files use the prefix *<output>.llvm-lto.**.
- *–save-temps=<dir>* Save LTO temporary files under the specified directory.
- *–lto-emit-asm* Run LTO and emit assembly only. No final link is performed. Output files use the prefix *<output>.llvm-lto.<task>.s*. Alias: *–plugin-opt=emit-asm*.
- *–lto-emit-llvm* Run LTO and emit LLVM bitcode to the final output file (*-o*). No final link is performed. Alias: *–plugin-opt=emit-llvm*.
- *–lto-obj-path=<prefix>* Prefix for LTO-generated object files. When provided, output objects are not deleted after LTO. Alias: *–plugin-opt=obj-path=*.
- *-flto-options=lto-asm-file=<file[,file2,…]>* Use pre-generated LTO assembly files as inputs to the external assembler.
- *-flto-options=lto-output-file=<file[,file2,…]>* Use pre-generated LTO object files as outputs (bypasses LTO code generation).

LTO optimization control

- *–lto-O=<level>* Set the LTO optimization level (0-4). Alias: *–plugin-opt=O*.
- *–lto-partitions=<number>* Set the number of code generation partitions. Alias: *–plugin-opt=lto-partitions=*.
- *–thinlto-jobs=<number>* Set the number of ThinLTO backend jobs. Overrides *–threads=* for ThinLTO. Alias: *–plugin-opt=jobs=*.
- *–lto-sample-profile=<file>* Provide a sample PGO profile for LTO. Alias: *–plugin-opt=sample-profile=*.
- *–lto-cs-profile-generate* Enable context-sensitive PGO instrumentation during LTO. Alias: *–plugin-opt=cs-profile-generate*.
- *–lto-cs-profile-file=<file>* Provide a context-sensitive profile for LTO. Alias: *–plugin-opt=cs-profile-path=*.
- *–lto-debug-pass-manager* Enable debug output for the new pass manager. Alias: *–plugin-opt=debug-pass-manager*.
- *–disable-verify* Disable the LLVM IR verifier in the LTO pipeline. Alias: *–plugin-opt=disable-verify*.
- *–dwodir=<dir>* Directory for split DWARF *.dwo* files generated during LTO.

LLVM option passthrough

- *–plugin-opt=-<llvm-option>* Pass a raw LLVM option to LTO (LLVMgold compatibility). For example, *–plugin-opt=-debug-pass-manager*.
- *-flto-options=codegen=<llvm-arg>* Pass LLVM codegen options to LTO (supports *-O*, *-mcpu=*, *-mattr=*, and arbitrary LLVM options).
- *-flto-options=<string>* Passes additional LTO plugin options through. For example, tests use *-flto-options=no-merge-modules* and *-flto-options=codegen=-split-lto-cg*.

Assembler control

- *-flto-use-as* Use the external assembler instead of the integrated assembler for LTO.

- *-flto-options=asmopts=<arg>* Extra arguments passed to the external assembler during LTO.

LTO caching

- *-flto-options=cache* Enable ThinLTO caching. By default, cache output is written to *<output>.ltocache*.
- *-flto-options=cache=<dir>* Enable ThinLTO caching using the specified directory.

Symbol preservation and ordering

- *-flto-options=preserveall* Preserve all bitcode symbols during LTO.
- *-flto-options=preserve-sym=<sym1,sym2,…>* Preserve the specified symbols in LTO.
- *-flto-options=preserve-file=<file>* Preserve the symbols listed in the given file (one symbol per line).
- *-flto-options=disable-linkorder* Disable link order enforcement when linker scripts are present.
- *-flto-options=verbose* Enable verbose LTO diagnostics and trace output.

Diagnostics and tracing

- *–trace-lto* Trace LTO stages and emit additional diagnostics.
- *–opt-record-file* Emit LTO optimization remarks. ELD writes *-LTO.opt.yaml* alongside the output file.
- *–display-hotness* Display hotness information with optimization remarks.
- *–print-timing-stats*, *–emit-timing-stats* Emit or print timing stats, including LTO phases.

Examples

Full LTO with embedded bitcode:

```
clang -flto -c foo.c -o foo.o
clang -flto -c bar.c -o bar.o
ld.eld -flto foo.o bar.o -o app.elf
```

ThinLTO with parallel backends and cache:

```
clang -flto=thin -c foo.c -o foo.o
clang -flto=thin -c bar.c -o bar.o
ld.eld -flto --thinlto-jobs=8 --flto-options=cache=thinlto.cache \
  foo.o bar.o -o app.elf
```

Selective LTO on a subset of objects:

```
# lto_list.txt contains glob patterns like:
#   libfoo/*.o
#   *vector_math*.o
ld.eld --include-lto-filelist=lto_list.txt foo.o bar.o -o app.elf
```

Emit LTO assembly for inspection:

```
ld.eld -flto --lto-emit-asm foo.o bar.o -o app.elf
# Output files: app.elf.llvm-lto.0.s, app.elf.llvm-lto.1.s, ...
```

References

- LLVM Link Time Optimization: Design and Implementation (*https://llvm.org/docs/LinkTimeOptimization.html*)

Getting Image Details

Overview

- The linker provides options for getting information about the files it generates.
- You can use following options to get information about how your file is generated by the linker, and also extract useful diagnostic information during the linking process.

Supported Diagnostic Options

- **–Map**

  Displays the image memory map, and contains the address and the size of each load region, execution region, and input section in the image, including linker-generated input sections.
- **-MapStyle=[Text|YAML|Binary]**

  Display the Map information in Text/YAML/Binary format.
- **-MapDetail <value>**

  Detail information in the map file
- **–verbose**

  Displays detailed information about the link operation, including the objects that are included and the libraries that contain them.
- **–trace**

  Allows tracing of various information in linker.

  - **all-symbols** - tracing of all symbols
  - **reloc** - tracing of relocations
  - **symbol=<symbol name>** - Allows tracing of a particular symbol. It also supports regular expressions
  - **section=<section name>** - Allows tracing of a particular section. Supports regular expressions.
  - **GarbageCollection** - tracing of garrbage collected symbols
  - **Trampolines** - tracing of trampolines.
  - **LTO** - Allows tracing of LTO.
- **-color <value>**

  Enable color output for diagnostics
- **-display_hotness <value>**

  Display hotness information for optimization remarks
- **–emit-timing-stats <value>**

  Emit time statistics of various linker operations to the specified file
- **-print-timing-stats**

  Print time statistics of various linker operations to console
- **–time-region <value>**

  - Emit time statistics for specified region of linker operation.
  - Otherwise, we can provide particular plugin's name as <value> of –time-region option to display stats of a particular plugin.
  - Users can provide <value> as "plugin" to display timing stats of all user plugins

- **–version**
  Print the Linker version and also lists the targets supported by the linker.

Command-line options

Here’s the list of command-line options accepted by the ELD:

Common options for all the backends

Compatibility Or Ignored Options

-EL
Link little endian objects.

-Qy
This option is ignored for SVR4 compatibility

--add-needed
Deprecated

--copy-dt-needed-entries
Add the dynamic libraries mentioned to DT_NEEDED

--ignore-unknown-opts
Disable warnings of unimplemented options

--nmagic
Turn off page alignment of sections, and disable linking against shared libraries

--no-add-needed
Deprecated

--no-allow-shlib-undefined
Do not allow undefined symbols from dynamic library when creating executables

--no-copy-dt-needed-entries
Turn off the effect of the –copy-dt-needed-entries

-plugin `<pluginfile>`
Specify a plugin file

-plugin-opt `<plugin-opt>`
Specify plugin options

-thin-archive-rule-matching-compatibility, --thin-archive-rule-matching-compatibility
Provides rule-matching compatibility when fat archives are converted to thin archives

DIAGNOSITC OPTIONS

-W<warn-type>
Print warnings which are OFF by default

- -Wall : print all warnings
- -W[no]linker-script : display warnings detected while parsing linker scripts
- -W[no]attribute-mix : display a warning if object files of RISC-V with different attributes are being linked
- -W[no-]archive-file : display warnings detected while reading archive files
- -W[no-]linker-script-memory : display warnings detected while processing linker script memory command
- -W[no-]bad-dot-assignments : display warnings for bad dot assignments
- -W[no-]error : treat warnings as errors
- -Wwhole-archive : display a warning when whole-archive is enabled for any archive file
- -W[no-]osabi : display a warning when linking objects with different values for OS/ABI

-color <on/off>
Enable color output for diagnostics

-cref, --cref
Print the references for a symbol or section

-t
Print all files processed by the linker

-display_hotness <value>
Display hotness information for optimization remarks

--emit-timing-stats <filename>
Emit time statistics of various linker operatons to the specified file

-emit-timing-stats-in-output, --emit-timing-stats-in-output
Insert link-time stats in section .note.qc.timing

--error-limit <max-error-number>
Maximum number of errors to emit before stopping (0 = no limit)

--error-style <style>
Specify an error style, LLVM/GNU

--fatal-internal-errors
Enable fatal internal errors

--fatal-warnings
Enable fatal warnings

--gc-cref <symbol/section>
Print the references for a symbol or section when garbage collection is enabled

--no-fatal-internal-errors
Disable fatal internal errors

--no-fatal-warnings
Disable fatal warnings

--no-warn-mismatch
Do not Warn on incompatible files passed to the linker.

-opt-record-file, --opt-record-file
Create diagnostic yaml file

-print-timing-stats, --print-timing-stats
Print time statistics of various linker operatons to console

--progress-bar
Show Progress Bar

--summary
Display linker run summary at the end

--time-region <region>
Emit time statistics for specified region of linker operation

-trace <trace-type>
Allow tracing of

- –trace=linker-script : trace linker script
- –trace=files : trace input files
- –trace=reloc=<pattern> : trace relocations
- –trace=symbol=<symbol-name> : trace symbol
- –trace=all-symbols : trace all symbols
- –trace=LTO : trace LTO
- –trace=garbage-collection : trace linker garbage collection

- –trace=plugin : trace plugin
- –trace=threads : trace threads
- –trace=assignments : trace symbol assignments
- –trace=command-line : trace header info
- –trace=live-edges : trace reachable sections when garbage collection is enabled
- –trace=merge-strings : trace linker string optimization
- –trace=trampolines : trace trampolines
- –trace=wrap-symbols : trace symbol wrap options
- –trace=symdef : trace symbol resolution from symdef files
- –trace=dynamic-linking : trace dynamic linking
- –trace=symbol-versioning : trace symbol versioning

--trace-linker-script
Trace stages of linker script processing

--trace-lto
Trace stages of lto

-trace-merge-strings <option>
Emit diagnostics describing how strings were merged. Options: all(default), allocatable_sections, <output section (regex)>

-trace-reloc <relocation>
trace a particular relocation

-trace-section <section_name>
Show metadata about a particular section

-y <symbol>
Trace a particular symbol

--verbose
Enable verbose output

-verbose=<verbose-level>
Enable verbose output

-verify-options <option>
Verify certain internal linker computations - currently supports reloc

--warn-common
Warn on common symbols

--warn-limit <maxwarnings>
Maximum number of warnings to emit (0 = no limit)

--warn-mismatch
Warn on incompatible files passed to the linker.

DYNAMIC LIBRARY OPTIONS

-Bgroup
Enable runtime linker to handle lookups in this object and its dependencies to be performed only inside this group

-Bsymbolic
Bind references to global symbols to the definition within the shared library

-Bsymbolic-functions
Bind references to global functions to the definition within the shared library

-g
Enable debug output when building shared libraries or executables

-fPIC
Enable PIC mode

--hash-size <size>
Specify a hash size when creating the hash sections for the dynamic loader

-hash-style <hashstyle>
Specify a hash style when creating the hash sections for the dynamic loader

--no-warn-shared-textrel
Don't Warn if the linker adds a DT_TEXTREL

-soname=<name>, **--soname**=<name>
Set the internal DT_SONAME field to the specified name

--warn-shared-textrel
Warn if the linker adds a DT_TEXTREL

DYNAMIC LIBRARY/EXECUTABLE OPTIONS

-dynamic-linker <path>
Set the path to the dynamic linker

-export-dynamic, **--export-dynamic**
Add all symbols to the dynamic symbol table when creating executables

-export-dynamic-symbol <symbol>
Export specified symbol to dynamic symbol table

-force-dynamic, **--force-dynamic**
Build executable as a force dynamic executable

-no-dynamic-linker, **--no-dynamic-linker**
The program does not need a dynamic linker when creating static PIE executables

--no-export-dynamic

-rpath <path>
Add a path to the runtime library search path

-rpath-link <path>
Specifies the path to search

Extended Options

--about
Emit detailed information about the linker

--align-segments
Align segments to page boundaries

--allow-bss-conversion
Dont produce an error when mixing NOBITS, and PROGBITS sections in the same segment

-copy-farcalls-from-file <filename>
Copy far calls instead of using trampolines

-z <extended-opts>
Extended Options or Non standard options.

Available options are:

- -z=combreloc : Combines multiple reloc sections and sorts them to make dynamicsymbol lookup caching
- -z=now : Enables immediate binding
- -z=nocopyreloc : Disables Copy Relocation-z=text : Do not permit relocations in read-only segments (default)
- -z=notext : Allow relocations in read-only segments
- -z=separate-code : Create separate code segment

- -z=no-separate-code : Do not create separate code segment
- -z=separate-loadable-segments : Create separate loadable segments

--disable-linker-version
Disable the LINKER_VERSION linker script directive (default)

--disable-new-dtags
Disable new dynamic tags

-dump-mapping-file <outputfilename>
Dump mapping file to output file

-dump-response-file <outputfilename>
Dump response file to output file

-emit-relocs-llvm, --emit-relocs-llvm
Emit relocations sections

--enable-linker-version
Enable the LINKER_VERSION linker script directive

--enable-new-dtags
Enable new dynamic tags

-mapping-file <INI-file>
Reproduce link using a mapping file

--no-align-segments
Dont align segments to page boundaries

-no-emit-relocs, --no-emit-relocs
Dont emit relocations in the output file, applicable to –emit-relocs-llvm option too

--no-record-command-line
Do not record the linker command line in the comment section

-no-reuse-trampolines-file <filename>
Dont reuse trampolines for symbols specified in file

--no-verify
Dont verify link output

--record-command-line
Record the linker command line in the comment section

-reproduce <tarfilename>
Write a tar file containing input files and command line options to inspect and re-link

-reproduce-compressed <tarfilename>
Write compressed tar file in zlib format containing input files and command line options to inspect and re-link.

-reproduce-on-fail <tarfilename>
Write reproduce tar file when link failscontaining input files and command line options to inspect and re-link.

-rosegment, --rosegment
Put read-only non-executable sections in their own segment

--script-options <matchtype>
Specify a match type, match-gnu/match-llvm

--use-old-style-trampoline-name
Use old style of naming trampolines

GENERAL OPTIONS

--build-id
Request creation of “.note.gnu.build-id” ELF note section

-build-id=fast/md5/sha1/tree/uuid/0x<hexstring>/none

Request creation of ".note.gnu.build-id" ELF note section

-o <outputfile>
Path to file to write output

--repository-version
Print the Linker Repository version

-sysroot <sysroot-path>
Set the system root

```
INPUT(file, file, ...)
GROUP(file, file, ...)
```

The files specified using the INPUT and GROUP command are searched inside the sysroot when:

- The file begins with the / character, and
- The script containing the INPUT/GROUP command is located within the sysroot directory.

-L <search_directory>
When <search_directory> is prefixed with =, then = is expanded to the sysroot.

--version
Print the Linker version

Help!

--help, -help
Print option help

--help-hidden, -help-hidden
Print hidden option help

Input Format

-b <input-format>
Specify input format for inputs following this option. Supported values: binary,default

OUTPUT KIND

-Bdynamic
Link against dynamic library

-dynamic
Create dynamic executable (default)

-no-pie
Do not create a PIE executable

-pie
Create PIE executable

-r
Create relocatable object file

-shared
Create dynamic library

-static
Create static executable

Link Time Speedup

-enable-threads=<option>
make linker fully threaded, available options: all

-no-threads, --no-threads

Disable Threads at Link time

--thread-count <threadcount>
Specify the number of threads for all linker operations

-threads, **--threads**
Enable Threads at Link time

LLVM and Target Options

-m
Select target emulation

-mabi <mabi>
Target ABI

-march <march>
Target architecture

-mcpu <mcpu>
Target CPU

-mllvm <option>
Options to pass to LLVM

-mtriple <triple>
Target triple to link for

LTO Options

--disable-verify, **-disable-verify**
Do not run the verifier during the optimization pipeline

-dwodir <dir>
Directory to save .dwo files when split DWARF is used in LTO

-exclude-lto-filelist <list_of_files>
Specify a list of files that are to be disregarded while using embedded bitcode section for LTO. This has no effect if -flto switch is not used.

-flto, **--flto**
Enable LTO if a Bitcode file is present.

-flto-options <option>
Specify various options with LTO

-flto-use-as, **--flto-use-as**
Use the standalone assembler instead of the integrated assembler for LTO

-include-lto-filelist <list_of_files>
Specify a list of files with embedded bitcode sections that are to be used for LTO. This has no effect if -flto switch is used

--lto-O<opt-level>
Optimization level for LTO

--lto-cs-profile-file=<value>
Context sensitive profile file path

--lto-cs-profile-generate
Perform context sensitive PGO instrumentation

--lto-debug-pass-manager
Debug new pass manager

--lto-emit-asm
Emit assembly code

--lto-emit-llvm

Emit LLVM-IR bitcode

--lto-obj-path=<prefix>
Specify file path used as a prefix for output LTO object files. Files created with this prefix will not be deleted after LTO finishes.

--lto-partitions=<number>
Number of LTO codegen partitions

-lto-sample-profile=<file>
Sample PGO profile file to be loaded for LTO

--opt-remarks-filename <value>
YAML output file for optimization remarks

--opt-remarks-format <value>
The format used for serializing remarks (default: YAML)

--opt-remarks-hotness-threshold <value>

--opt-remarks-passes <value>
Regex for the passes that need to be serialized to the output file

--opt-remarks-with-hotness
Include hotness information in the optimization remarks file

--plugin-opt=-<value>, **-plugin-opt**=-<value>
Specify an LLVM option for compatibility with LLVMgold.so

--plugin-opt=stats-file=<value>, **-plugin-opt**=stats-file=<value>
Filename to write LTO statistics to

-save-temps, **--save-temps**
Save the temporary files produced by LTO

-save-temps=<dir>
Save the temporary files produced by LTO in the directory

--thinlto-jobs=<number>
Number of ThinLTO jobs. Default to –threads=

EXECUTABLE OPTIONS

-L
Path to search for libraries or linker scripts

-Y <path>
Add path to the default library search path

-emit-relocs, **--emit-relocs**
Make Emit relocs behave just like GNU.

-e <symbolname>
Name of entry point symbol

-fini <symbol>
Specify a finalizer function

--image-base <address>

-init <symbol>
Specify an initializer function

-l
Root name of library to use

--library-path <path>
Path to search for libraries or linker scripts

--library=<namespec>
library to use

--no-omagic
This option negates most of the effects of the -N option.Disable linking with shared libraries

-no-whole-archive, --no-whole-archive
Restores the default behavior of loading archive members

--noinhibit-exec
Retain the executable output file whenever it is still usable

-nostdlib
Disable default search path for libraries

--omagic
Set the text and data sections to be readable and writable. Also, do not page-align the data segment, and disable linking against shared libraries.

-whole-archive, --whole-archive
Force load of all members in a static library

Map Options

-Map <filename>
Dump the output layout to the map file

-MapDetail <option>
Detail information in the map file

-MapStyle <fileformat>
Dump the output layout to the map file in YAML/Text/Binary Form

-M
Emit the map file

-trampoline-map <filename>
Dump Trampoline Information in YAML format

--color-map
Color the map file

Misc Features

-allow-incompatible-section-mix, --allow-incompatible-section-mix
Allow incompatible section mix

OPTIMIZATION OPTIONS

--eh-frame-hdr
Create EH Frame Header section for faster exception handling

-gc-sections, --gc-sections
Enable garbage collection

-no-gc-sections, --no-gc-sections
Disable garbage collection

--no-merge-strings
Disable String Merging

--no-trampolines
Disable Trampolines

-print-gc-sections, --print-gc-sections
Print sections that are garbage collected

--sframe-hdr
Process and emit .sframe sections for stack unwinding

Features From other Linkers

-R <filename>
Read symbol names and addresses from filename

-symdef, --symdef
Output SymDef file to console

--symdef-file <filename>
Emit SymDef file

--symdef-style <style>
Determine how to handle symbol resolution for symbols from symdef file, Options available are provide

Plugin Options

-plugin-activity-file <filename>
Emit plugin activity JSON files

--no-default-plugins
Allow no plugins to be implicitly loaded

--plugin-config <config-file>
Specify a plugin configuration

SYMBOL RESOLUTION OPTIONS

--allow-multiple-definition
Allow multiple definitions

-allow-shlib-undefined, --allow-shlib-undefined
Allow undefined symbols from dynamic library when creating executables

--as-needed
This option affects ELF DT_NEEDED tags for dynamic libraries mentioned on the command line

-defsym symbol=<expression>
Create a global symbol in the output file containing the absolute address given by expression

-end-group, --end-group
End a group

--end-lib
End a library

--end-lib-thin
End a thin library

--no-as-needed
This option restores the default behavior of adding DT_NEEDED entries

-no-undefined, --no-undefined
Report unresolved symbol references from regular object files

--pop-state
Restore previously saved input handling flags

--push-state
Save current input handling flags (as-needed, whole-archive, etc)

-start-group, --start-group
Start a group

--start-lib
Start a library

--start-lib-thin
Start a thin library

-u
Force symbol to be entered in the output file as an undefined symbol

--use-shlib-undefines
Resolve undefined symbols from dynamic libraries

--warn-once
Warn only once for every undefined reference

SCRIPT OPTIONS

-T <linkerscriptfile>
Use the given linker script in place of the default script.

-Tbss <address>
Specify an address for the .bss section

-Tdata <address>
Specify an address for the .data section

-Ttext <address>
Specify an address for the .text section

-Ttext-segment <address>
Specify an address for the .text-segment segment

-default-script <pluginfile>
Use the linker script as the default linker script.

--no-check-sections
Do not check section addresses for overlaps

-dynamic-list <list-of-symbols>
Specify a list of symbols if present will be exported

--check-sections
Check section addresses for overlaps(default)

--exclude-libs <libs>
Specifies a list of archive libraries from which symbols should not be automatically exported

-extern-list <list-of-symbols>
Specify a list of symbols that exists as external dependencies

--global-merge-non-alloc-strings
merge non-alloc strings across output sections

--map-section <section>
Specify a input section that maps to a output section

-orphan-handling <mode>
Specify how to handle orphan sections. Options available are place,error, warn

--print-memory-usage
Print memory usage when MEMORY linker script directive is used

-section-start <address>
Specify a virtual output section address for a specified section

--sort-section <sort_section>

-unique-output-sections, --unique-output-sections
Place each input section in a unique output section

--unresolved-symbols <option>
Determine how to handle unresolved symbols, Options available are ignore-all,report-all(Default), ignore-in-object-files, ignore-in-shared-libs

-version-script <linkersciptfile>
Use the linker script as a version script.

SYMBOL OPTIONS

-d
Assign space to common symbols

--demangle
Demangle C++ symbols

--demangle-style <value>
Specify whether the linker should demangle symbols when emitting errors or emitting Map files

--discard-all
Discard all symbols

--discard-locals
Discard all local symbols

--no-demangle
Dont demangle C++ symbols

-portable <symbol>
Specify symbol to be portable wrapped to

--sort-common
sort common symbols by alignment

-sort-common=<option>
Sort common symbols by ascdending/descending order of alignment

--strip-all
Omit all symbol informations from output

--strip-debug
Omit all debug information from output

-wrap <symbol>
Specify symbol to be wrapped to

Using `–start-lib` / `–end-lib`

`–start-lib` and `–end-lib` let you pass one or more object files and have ELD treat them as if they were members of an archive for symbol resolution purposes.

Conceptually, ELD packages the object files between `–start-lib` and `–end-lib` into an in-memory archive and then processes that archive like a normal `.a` input.

Thin variant

ELD also supports `–start-lib-thin` / `–end-lib-thin`, which packages the enclosed object files into an in-memory *thin* archive (member paths are stored instead of embedding member bytes).

Linker scripts with archive member patterns

ELD supports using archive member patterns in linker scripts to match inputs from a `–start-lib` region.

Internally, ELD assigns a synthetic archive name to each `–start-lib` region: `<start-lib:1>`, `<start-lib:2>`, etc., in command-line order. Thin regions use `<start-lib-thin:N>`. You can reference these synthetic archive names using the standard `archive:member` pattern syntax in an input section description.

Example:

```
$ ld.eld main.o --start-lib foo.o bar.o --end-lib -T script.t -MapStyle txt -Map out.map
```

```
/* script.t */
SECTIONS {
  .foo_out : { "*<start-lib:1>:*foo.o"(.text*) }
  .bar_out : { "*<start-lib:1>:*bar.o"(.text*) }
}
```

Notes:

- The <start-lib:N> names are ELD-specific (they do not correspond to a real on-disk archive).
- If you need a stable archive name for portability, create a real archive (e.g. with `ar cr libname.a …`) and match against `*libname.a:*member.o` patterns in the script.

MEMORY-related options

When using linker scripts with `MEMORY` regions, these options are commonly useful for debugging and enforcing correctness:

- `–print-memory-usage` prints per-region usage when a `MEMORY` directive is present in the linker script.
- `-Wlinker-script-memory` enables warnings for suspicious `MEMORY` setups (for example, zero-sized regions or regions that end up unused).

ARM and AArch64 specific options

ARM/AArch64 Linker Options

-compact, --compact
Create a smaller output file. The Loader need to support such files

--disable-bss-conversion
Don’t convert BSS to NonBSS when BSS/NonBSS Sections are mixed

--enable-bss-mixing
Enable mixing BSS/NonBSS sections

--use-mov-veneer
Use movt/movw to load address in veneers with absolute relocation

ARM Linker Option ONLY

-execute-only, --execute-only
Mark executable sections execute-only on AArch64

-fix-cortex-a53-843419, --fix-cortex-a53-843419
Fix the cortex a53 errata 843419

--fix-cortex-a8
Fix the Cortex A8 bug

-fropi, --fropi
Enable Read write data access relative to a static base registers.

-frwpi, --frwpi
Enable Read write data access relative to a static base registers.

--no-fix-cortex-a8
Dont Fix the Cortex A8 bug

-target2 <type>
Interpret R_ARM_TARGET2 as <type>, where <type> is one of abs, rel, or got-rel (default).

Hexagon specific options

Hexagon Linker experimental Options

--relax
Enable Hexagon Relaxation (default behavior)

-relax=<regex>

Restrict relaxation to only certain sections

Hexagon Options

--gpsize=<maxsize>
Set the maximum size of objects to be optimized using GP

RISCV specific options

Symbol Patching

--patch-base <value>
Specify base image to generate patch for

--patch-enable
Enable symbol patching using indirection

RISCV Linker Options

--disable-bss-conversion
Don't convert BSS to NonBSS when BSS/NonBSS Sections are mixed

--enable-bss-mixing
Enable mixing BSS/NonBSS sections

--keep-labels
Keep all local labels for debugging purposes

--no-relax-gp
Disable GP relaxation

--no-relax-tlsdesc
Disable relaxing TLSDESC instruction sequences

--no-relax-zero
Disable zero-page relaxation

--no-relax
Disable relaxation

--no-relax-c
Disable relaxation to compressed instructions

--no-relax-xqci
Disable relaxing to/from Xqci instructions (default behavior)

--relax
Enable relaxation (default behavior)

--relax-xqci
Enable relaxing to/from Xqci instructions

-z keyword options

The following `-z` keywords are supported by ELD:

`-z combreloc`
Combine multiple dynamic relocation sections and sort them to improve dynamic symbol lookup caching.

`-z nocombreloc`
Disable relocation section combining.

`-z global`
When building a shared object, make its symbols available for resolution by subsequently loaded libraries.

`-z defs`
: Report unresolved symbol references from regular object files, even when creating a non-symbolic shared library.

`-z initfirst`
: When building a shared object, mark it to be initialized before other objects loaded at the same time, and finalized after them.

`-z muldefs`
: Allow multiple definitions.

`-z nocopyreloc`
: Disable linker-generated .dynbss variables used in place of shared-library variables. This may result in dynamic text relocations.

`-z nodefaultlib`
: Ignore default library search paths when resolving dependencies at runtime.

`-z relro`
: Create a PT_GNU_RELRO segment and request it be made read-only after relocation, if supported. This is disabled by `-z norelro`.

`-z norelro`
: Do not create a PT_GNU_RELRO segment.

`-z lazy`
: Mark the output for lazy binding (resolve function calls on first use).

`-z now`
: Mark the output for immediate binding (resolve all symbols at load time).

`-z origin`
: Require `$ORIGIN` handling in rpath/runpath entries.

`-z text`
: Report an error if DT_TEXTREL is set (dynamic relocations in read-only sections).

`-z notext`
: Allow dynamic relocations in read-only sections (do not report DT_TEXTREL).

`-z noexecstack`
: Mark the output as not requiring an executable stack.

`-z nognustack`
: Ignore `.note.GNU-stack` and do not create PT_GNU_STACK based on it.

`-z separate-code`
: Create a separate code PT_LOAD segment with instructions on pages disjoint from any other data. This is a no-op when a linker script with a `SECTIONS` command is used.

`-z noseparate-code`
: Disable separate code segment handling. This is the default behavior.

`-z execstack`
: Mark the output as requiring an executable stack.

`-z nodelete`
: Mark a shared object as non-unloadable at runtime.

`-z compactdyn`
: Emit a more compact dynamic section by omitting DT_PLTGOT and DT_DEBUG entries.

`-z force-bti`
: Force GNU property BTI feature bits to be recorded (AArch64).

`-z pac-plt`
: Force GNU property PAC feature bits to be recorded for PLT use (AArch64).

`-z common-page-size=<value>`

Set the most common page size used for segment alignment and layout optimization.

`-z max-page-size=<value>`

Set the maximum supported page size used for segment alignment.

Linker version directive

`--enable-linker-version`

Enable the GNU linker-script `LINKER_VERSION` directive. When this option is active, every `LINKER_VERSION` statement encountered in a linker script emits a NUL-terminated string containing the eld version at that position in the output section. This matches the GNU ld directive and is useful for embedding the linker version directly into a binary. The option applies to the entire link once specified.

`--disable-linker-version`

Disable the `LINKER_VERSION` directive. This is the default behaviour, so the directive is parsed but emits no data unless the feature has been explicitly enabled.

Record command line

`--record-command-line`

Record the linker command line in the `.comment` section. This is disabled by default.

`--no-record-command-line`

Do not record the linker command line in the `.comment` section.

Target Specific Features

Hexagon

- Plugins - Information can be found at Linker Plugins
- Small data
- Trampolines
- **–disable-guard-for-weak-undef** - Disable guard for weak undefined symbols
- **-gpsize=<value>** - Set the maximum size of objects to be optimized using GP

ARM

- Baremetal
- Thumb
- ARM
- Linux
- Android
- Supported errata fixes - `fix_cortex_a8` - `fix_cortex_a53_843419`
- **enable-bss-mixing** - Enable mixing of BSS and non-BSS sections
- **-execute-only** - Mark executable sections execute-only
- **–disable-bss-conversion** - Don’t convert BSS to NonBSS when BSS/NonBSS sections are mixed

- **–use-mov-veneer** - Use movt/movw to load address in veneers with absolute relocation
- **–compact**
  - `–compact` is used to reduce the output ELF size.
  - The loader must be able to load such ELF files and satisfy `segmentaddr - sectionaddr == segmentoffset - sectionoffset`.
  - Offsets are incremented by section size, not by virtual address difference.
  - Supported only in ARM/AArch64 baremetal environments (typically low memory).

Note

The impact of `–compact` can be observed by checking the segment sizes.

Note

- File offset of every section must be consistent with physical address of the segment:

```
Section_file_offset - segment_file_offset
  == section_physical_address - segment_physical_address
```

- If a section does not respect the rule, the linker reports an error.
- The image must still be a valid ELF; in particular:

```
offset % page_alignment
  == physical_address % page_alignment
  == virtual_address % page_alignment
```

Example showing `–compact` reducing segment size to `0x20` as seen from `readelf -l`:

```
$ cat script.t
PHDRS {
  A PT_LOAD;
}

SECTIONS {
  .A (0x1000) : AT(0x1000) { *(.text.foo) *(.ARM.exidx.text.foo) } :A
  .B (0x10000) : { *(.text.bar) *(.ARM.exidx.text.bar) } :A
}

$ arm-link 1.o -T script.t -o t4.out --compact
$ llvm-readelf -S -W t4.out
  There are 8 section headers, starting at offset 0x220:

  Section Headers:
  [Nr] Name Type Address Off Size ES Flg Lk Inf Al
  [ 0] NULL      00000000 000000 000000 00 0  0  0
  [ 1] .A PROGBITS 00001000 000054 000010 00 AX 0  0  4
  [ 2] .B PROGBITS 00010000 000064 000010 00 AX 0  0  4

$ llvm-readelf -l -W t4.out
  Elf file type is EXEC (Executable file)
  Entry point 0x0
  There are 1 program headers, starting at offset 52

  Program Headers:
    Type Offset   VirtAddr   PhysAddr   FileSiz MemSiz Flg Align
```

```
  LOAD 0x000054 0x00001000 0x00001000 0x00020 0x0f010 R E 0x4

Section to Segment mapping:
  Segment Sections...
  00      .A .B
  None    .ARM.attributes .comment .shstrtab .symtab .strtab
```

AArch64

- Baremetal
- AArch64
- Linux
- Android
- **enable-bss-mixing** - Enable mixing of BSS and non-BSS sections
- **–disable-bss-conversion** - Don’t convert BSS to NonBSS when BSS/NonBSS sections are mixed
- **–use-mov-veneer** - Use movt/movw to load address in veneers with absolute relocation
- **–compact**
- **-z nognustack** - Do not create a `GNU_STACK` segment

RISCV

- Baremetal
- Linux
- **-gpsize=<value>** - Set the maximum size of objects to be optimized using GP
- Supports relaxation - `–relax` (enabled by default) - `--no-relax` to disable relaxation
- **enable-bss-mixing** - Enable mixing of BSS and non-BSS sections
- **-W[no]attribute-mix** - Warn about RISC-V attributes mix instead of failing to link
- **–disable-bss-conversion** - Don’t convert BSS to NonBSS when BSS/NonBSS sections are mixed

SFrame Support

Overview

SFrame (Simple Frame) is a lightweight debugging format that provides stack unwinding information for debuggers and profilers. ELD provides comprehensive support for processing *.sframe* sections and creating SFrame headers in linked executables and shared libraries.

The SFrame format is designed to be much simpler and more compact than DWARF debug information, focusing solely on the minimal information needed for stack unwinding:

- **Canonical Frame Address (CFA)**: The address of the call frame
- **Frame Pointer (FP)**: Location of the frame pointer
- **Return Address (RA)**: Location of the return address

ELD implements support for **SFrame version 2 (errata 1)** as specified in the SFrame Format documentation.

SFrame vs EhFrame

SFrame offers several advantages over the traditional EhFrame format:

Size Efficiency
: SFrame sections are significantly smaller than equivalent EhFrame sections, resulting in reduced binary size and memory usage.

Parsing Speed
: The simplified format allows for faster parsing during stack unwinding operations.

Reduced Complexity
: SFrame eliminates much of the complexity of DWARF-based unwinding while maintaining the essential functionality.

Better Cache Performance
: The compact format leads to better cache utilization during runtime stack unwinding.

Command Line Options

ELD provides the following command line option for SFrame support:

`--sframe-hdr`

Create an SFrame header section and process *.sframe* sections in input files. This option enables:

- Processing of *.sframe* sections from input object files
- Creation of consolidated SFrame sections in the output
- Proper linking and address resolution for SFrame data
- Integration with ELD's section layout and memory management

Usage Examples

Basic Linking with SFrame

To link object files containing *.sframe* sections and create an SFrame header:

```
# Create source file with SFrame section
cat > source1.s << 'EOF'
.text
.globl func1
func1:
    ret
.section .sframe,"a"              # Type automatically set to SHT_GNU_SFRAME (llvm-mc 22.x+)
.byte 0xe2, 0xde, 0x02, 0x08      # SFrame header with magic and version
EOF

# Assemble files containing SFrame sections (requires llvm-mc 22.x or newer)
llvm-mc -filetype=obj -triple=x86_64-linux-gnu source1.s -o source1.o
llvm-mc -filetype=obj -triple=x86_64-linux-gnu source2.s -o source2.o

# Link with SFrame header creation
eld --sframe-hdr source1.o source2.o -o executable
```

Note

For llvm-mc versions prior to 22.x, the section type must be specified explicitly:

```
.section .sframe,"a",@0x6ffffff4  # Explicit SHT_GNU_SFRAME type for older assemblers
```

Shared Library with SFrame

SFrame sections can also be included in shared libraries:

```
# Create shared library with SFrame information
eld --sframe-hdr -shared libfoo.o -o libfoo.so
```

Combining with Other Options

SFrame support works seamlessly with other ELD features:

```
# SFrame with garbage collection
eld --sframe-hdr --gc-sections input.o -o output

# SFrame with optimization
eld --sframe-hdr -O2 input.o -o output

# SFrame with debug symbols
eld --sframe-hdr --debug input.o -o output
```

Section Processing

Input Processing

ELD processes *.sframe* sections from input object files by:

1. **Recognition**: Identifying *.sframe* sections in ELF input files
2. **Parsing**: Validating SFrame headers and parsing FRE (Frame Row Entry) data
3. **Merging**: Combining multiple *.sframe* sections from different input files
4. **Address Resolution**: Resolving function addresses and updating SFrame data

Output Generation

When the *–sframe-hdr* option is specified, ELD creates:

- **Consolidated .sframe section**: Contains all SFrame data from input files
- **Proper section alignment**: Ensures correct alignment for runtime access
- **Address fixups**: Updates all address references to final linked addresses
- **Section headers**: Creates appropriate ELF section headers for SFrame data

Supported Architectures

ELD's SFrame support is available for the architectures with ABI identifiers defined in the SFrame specification:

- **AArch64**: ARM 64-bit architecture (little-endian and big-endian)
- **AMD64 (x86_64)**: AMD/Intel 64-bit architecture (little-endian)

Other architectures are not yet assigned ABI identifiers in the SFrame specification and are not currently supported.

Integration with Debugging Tools

SFrame sections created by ELD are compatible with tools that support the SFrame format:

- **GDB**: GNU Debugger (version 16+) can use SFrame information for stack unwinding
- **libsframe**: The libsframe library (from GNU binutils) can parse SFrame sections

Garbage Collection Behavior

When using `–gc-sections` with SFrame support:

- **Preservation**: `.sframe` sections are not subject to garbage collection and are preserved in the output

Error Handling

ELD provides comprehensive error detection for SFrame processing:

Invalid Magic Number
: Reports errors for SFrame sections with incorrect magic numbers (expected: 0xdee2)

Unsupported Version
: Detects and reports unsupported SFrame versions (ELD supports version 2)

Truncated Sections
: Identifies and reports SFrame sections with incomplete data

Address Resolution Failures
: Reports errors when SFrame function addresses cannot be resolved

Example error output:

```
Error: SFrame Read Error : invalid SFrame magic number from file input.o
Error: SFrame Read Error : unsupported SFrame version from file input.o
Error: SFrame Read Error : section too small for SFrame header from file input.o
```

Performance Considerations

SFrame support in ELD is designed for optimal performance:

Link Time

- Minimal overhead during linking
- Efficient parsing of SFrame sections
- Fast address resolution and fixups

Runtime

- Compact sections reduce memory usage
- Fast parsing during stack unwinding
- Improved cache locality

Binary Size

- Significantly smaller than equivalent DWARF information
- Optional inclusion based on build requirements

Best Practices

1. **Consistent Usage**: Use *–sframe-hdr* consistently across all objects in a project
2. **Testing**: Verify SFrame functionality with debugging tools after linking
3. **Architecture Specific**: Ensure SFrame generation tools target the correct architecture
4. **Integration**: Combine with appropriate optimization levels for best results
5. **Validation**: Use readelf or similar tools to verify SFrame section contents

Troubleshooting

Missing SFrame Sections
: Ensure input objects were compiled with SFrame generation enabled

Incorrect Function Addresses
: Verify that all input objects use consistent compilation flags

Debugger Issues
: Check that debugging tools support SFrame format version 2

Performance Problems
: Consider the trade-off between SFrame inclusion and binary size requirements

For additional support and troubleshooting, consult the ELD diagnostic output when using the *–verbose* option along with *–sframe-hdr*.

Linker support and frequently asked questions!!!

General Linker Questions

How do I pass arguments to linker using clang/clang++ driver ?

clang/clang++ driver provides **“-Wl,<args>”** option to pass args to linker.

-Wl,<args>

Pass the comma separated arguments in args to the linker

E.g. clang <compiler_options> **-Wl,-Map=test.map,-e=main**

where -Map and -e are linker specific options/args.

These args will be passed to the underlying linker used for processing.

Alternatively, user can also use **“-Xlinker <arg>”** option to pass argument one at a time to linker.

-Xlinker <arg>

Pass arg to the linker.

E.g. clang <compiler_options> **-Xlinker -Map=test.map -Xlinker -e=main**

How to remove unused functions from the final ELF ?

This can be achieved in two ways:

- **With LTO**
    - All source files should be compiled with compiler option “-flto”.
    - During linking stage, enable LTO optimization by specifying “-flto” option.

      E.g. clang <compiler_options> **-flto -c** foo.c -o foo.o

      > **Note**
      >
      > When -flto option is used during compilation, the output object file (say, foo.o) is not in ELF format.It will be in LLVM-IR bitcode binary format.

      ```
      clang <compiler_options> -flto -c bar.c -o bar.o
      ld.eld <other_linker_options> -flto foo.o bar.o -e <entry_point> -o final.out
      ```
- **Without LTO**
    - All source files should be compiled with compiler option “-ffunction-sections”.
    - During linking stage, enable garbage collection by specifying “–gc-sections” option. Example:

      ```
      clang <compiler_options> -ffunction-sections -c foo.c -o foo.o
      clang <compiler_options> -ffunction-sections -c bar.c -o bar.o
      ld.eld <other_linker_options> --gc-sections -e <entry_point> foo.o bar.o -o final.out
      ```

How do I trace a symbol ?

The user can trace a symbol with –trace-symbol <symbolname> or –trace=symbol=<symbolname>

Example:

```
$ cat 1.c
int foo() {return 0;}
int bar() {return 1;}
$ cat 2.c
int baz() {return foo() + bar ();}
$ ld.eld 1.o 2.o --trace-symbol bar # equivalent to --trace=symbol=bar
Note: Symbol `bar' from Input file `1.o' with info `(ELF)(FUNCTION)(DEFINE)[Global]{DEFAULT}' being added to Namepool
Note: Symbol `bar' from Input file `2.o' with info `(ELF)(NOTYPE)(UNDEFINED)[Global]{DEFAULT}' being added to Namepool
Note: Symbol `bar' from Input file `1.o' with info `(ELF)(FUNCTION)(DEFINE)[Global]{DEFAULT}' being resolved from Namepool
Symbol bar, application site: 0x2c
```

How do I trace a relocation ?

The user can trace a relocation with the option –trace=reloc="<relocname>"

Figure out garbage collected symbols

The user can find the list of sections that the linker garbage collected using the option *–print-gc-sections* option.

The user then needs to go over the sections in the input files, and list the symbols using objdump tool.

Linker is garbage collecting a symbol

The linker can garbage collect the symbol only if the symbol is not referenced. You can use the option –trace=live-edges and see if you are finding a reference to the section that contains the symbol.

If you don't see the section that contains the symbol, there might be a missing link.

You might want to use the option

- –entry=<entry symbol>
- Use the KEEP linker script directive to add to the list of sections that needs to be kept.
- -u *symbol_name* to keep a symbol from being garbage collected

Print Timing Information

Linker can print timing information with the option –print-timing-stats. This option can be used to see where the linker is spending most of its time.

Linker is taking a long time to link

The linker may be taking a long time to link due to the following patterns

- **A lot of linker script rules**
    - Linker script rules with EXCLUDE_FILE
- A lot of object files
- Bad image layout

For figuring out any of the above, you should check the Map file and see the statistics when the link is successful.

How to figure out why a archive member is being loaded by the linker

If you are trying to figure out why an archive member is being loaded you need to look at the archive records.

Example:

```
cat > 1.c << \!
int foo() { return bar(); }
!

cat > 2.c << \!
int bar() { return baz(); }
!

cat > 3.c << \!
int baz() { return bar(); }
!


clang -target hexagon -c 1.c -ffunction-sections
clang -target hexagon -c 2.c 3.c -ffunction-sections
```

```
hexagon-ar cr mylib.a 2.o 3.o
hexagon-link 1.o mylib.a -Map x
```

If you see the map file section for archive reference records, you will find this information :-

```
Archive member included because of file (symbol)
mylib.a(2.o)
                1.o (bar)
mylib.a(3.o)
                mylib.a(2.o) (baz)
```

You can interpret the information as below :-

- There was a reference in 1.o for bar, hence mylib.a(2.o) was loaded.
- There was a reference in 2.o for baz, hence mylib.a(3.o) was loaded.

Controlling Page size

You can use the option z max-page-size=<x> to set the page size used by the linker.

How to fail a build for linker warnings

A linker warning may be considered fatal when the switch –fatal-warnings is used.

You can turn off this behavior –no-fatal-warnings or removing the switch –fatal-warnings.

Weak references and definitions

Symbols can be given weak binding by the compiler and assembler. Weak references and definitions are typically references to library functions.

The linker does not load an object from a library to resolve a weak reference. It is able to resolve the weak reference only if the definition is included in the image explicitly.

This can be done by specifying the object file containing the definition on the link command line.

Alternatively, using “–whole-archive <archive-file> –no-whole-archive” linker options includes all objects files in the archive to the link.

Also, removing the weak attribute on the symbol will make it a normal or non-weak. For non-weak symbols, linker will scan the library/archive and loads the required member(s).

Note

An unresolved weak function call is replaced with a no-operation instruction, NOP (only for aarch64 and if the linker has not reserved a PLT).

An example showing that the linker does not load an object from a library to resolve a weak reference.

By inspecting the disassembly of “a.out”, user can observe that call to bar() in main() is replaced with a NOP.

```
cat > 1.c << \!
__attribute__((weak))  int bar();
int main() {
 bar();
 return 0;
}
!

cat > 2.c << \!
int bar() { return 0; }
!
```

```
clang -c 1.c 2.c -target aarch64
rm -f lib2.a
llvm-ar cr lib2.a 2.o
ld.eld 1.o lib2.a -t -march aarch64
```

Reproducing a failure

If you are having an issue, and you want to pass the link step for someone else to debug, you can use the *–reproduce <tarball>* option.

The *–-reproduce <tarball>* option creates a tar ball with all the inputs that were fed to the linker, and rewrites the link command to make it easy for users to reproduce the problem.

Note

If the tarball is big and you want the linker to compress the tarball automatically, you can use the switch -reproduce-compressed.

Note

If you are having a link time failure and the problem does not reproduce everytime, but in specific builds, it may be because of non determinism in the builds. In that case use the option –reproduce-on-fail.

The reproduce-on-fail switch only creates a tarball when the link step fails.

Multiple ways to invoke ELD linker

- arm-link can be invoked since it is a symbolic link to ld.eld

```
> cat bar.c
int main(){return 0;}

clang -target  armv7-none-eabi  -g  -c bar.c  -o bar.o
arm-link -m armelf_linux_eabi -dynamic-linker /lib/ld-linux.so.3 -o bar.elf --strip-debug bar.o
```

- Pass -fuse-ld=eld flag to clang driver, so that eld is used for linking

```
clang -target armv7-none-eabi -fuse-ld=eld -fuse-baremetal-sysroot bar.c -o bar.elf -fno-use-baremetal-crt
```

Note

ELD linker flags can be passed to clang driver using -Wl prefix. Example : clang …… -Wl,-shared -Wl,-gc-sections bar.o -o bar.elf

- Pass -v (verbose) when linking using clang binary, pick the link command line, edit and execute it

Ignore multiple definition errors and continue with the link

Sometimes a developer might want to ignore all multiple definition errors and continue with the link.

This can be achived by using –allow-multiple-definition switch.

Example:

```
cat > 1.c << \!
int foo() { return 0; }
!

cat > 2.c << \!
int foo() { return 1; }
!

#compile
clang -target hexagon -c 1.c 2.c
#produces error
hexagon-link 1.o 2.o
#produces a successful link
hexagon-link 1.o 2.o --allow-multiple-definition
```

Multiple ways to provide entry point to linker

- Using linker flag : -e <value> –> Name of entry point symbol
- Specifying ENTRY(symbol) command in a linker script
- Initialising the value of linker symbol “start”
- Specifying the start address for the first input section in linker script (eg: .text : AT(0))

How to obtain a non-executable stack

Non-executable stack (NX) is a virtual memory protection mechanism to block shell code injection from executing on the stack by restricting a particular memory and implementing the NX bit.

ELD has the following equivalent option :

-z noexecstack : Mark the executable as not requiring an executable stack

What is the difference between section type NOBITS and PROGBITS

NOBITS section do not occupy size on the disk.

PROGBITS section occupies space on disk.

How to link libraries to resolve circular dependencies

The order in which the libraries are loaded matter. Incorrect order leads to “undefined reference” errors.

Keeping the libraries to linked within “–start-group” and “–end-group” linker flags takes care of the circular dependencies, so that the user need not worry about the order in which libs are loaded.

Circular Dependies:

```
cat > 1.c << \!
extern int bar();
int foo() { return bar(); }
!

cat > 2.c << \!
extern int fred();
int bar() { return fred(); }
!

cat > 3.c << \!
int baz() { return 0; }
!

cat > 4.c << \!
```

```
extern int baz();
int fred() { return baz(); }
!

clang -target hexagon -c 1.c 2.c 3.c 4.c
# lib2.a creates a dependency of fred in lib4.a in a different group
hexagon-ar cr lib2.a 2.o 3.o
# lib3.a creates a reverse-dependency to lib2.a which was previously visited
hexagon-ar cr lib3.a 4.o
# problem
hexagon-link 1.o --start-group lib2.a --end-group --start-group lib3.a --end-group
# solution 1
hexagon-link 1.o --start-group lib2.a --end-group --start-group lib3.a lib2.a --end-group
# solution 2
hexagon-link 1.o --start-group 2.o 3.o --end-group --start-group lib3.a --end-grou
```

How to disable use of standard system startup files

“-nostartfiles” : Does not use the standard system startup files when linking.

If this option is specified, user has to specify their own program entry point and write their own start files

How to disable use of standard system libraries

“-nostdlib” : Disable default search path for libraries

This option will prevent the compiler from picking the standard libraries, rtlibs provided by the toolchain. User has to come up with their own definitions or libs so that there are no undefined reference errors reported.

The compiler rt libraries have to be specified explicitly if there are calls to __aeabi* functions and are not defined by the users.

How to remove a section from a OBJ file

llvm-objcopy tool can be used to remove a section from an ELF file.

Example:

```
llvm-objcopy --remove-section=<section_name>
```

How to extract OBJ files from an archive

llvm-ar is the utility to extract obj files from the archive

Example:

```
llvm-ar -x <archive_file>
```

How to obtain a deterministic output from linker

By default deterministic behaviour is supported. To get maximum throughput from linker, –enable-threads=all linker option must be passed.

How to obtain memory statistics from an ELF file or OBJ file

llvm-size utility can provide details of size of .text, .data, .rodata, .bss, etc.

How to compare layout of two output images

Mapfile contains the layout of the image. You can compare the layout section of mapfiles of the two output images which you are interested in.

By default, mapfiles layout section also contains virtual addresses associated with the layout. This brings in too much unnecessary noise when comparing the two layouts, because a small change in the layout can result in change of virtual address of many sections. This can be fixed by using '-MapDetail only-layout' option.

```
// Run link commands with '-MapDetail only-layout' when generating the two output images.
vim -d image1.map image2.map # To compare the two output images layouts.
```

How to embed a binary and use it in 'C' code

You can include a binary file and use it in 'C' code to reference the binary content.

Here is an example which builds a simple table and embeds that table in 'C' code.

Simple python code to create a binary file

The below python code when run creates a binary file by name table.bin

```
f = open('table.bin', 'w+b')
byte_arr = [1,2,3,4,5]
binary_format = bytearray(byte_arr)
f.write(binary_format)
f.close()
```

Create a simple assembly file to include this binary file

This snippet of assembly creates a section named table and includes

```
.section "table","a",@progbits
.incbin "table.bin"
```

Include this assembly file to build a static executable

The program snippet below prints the contents of the binary file to stdout.

```
extern char __start_table;
extern char __stop_table;
int main() {
  char *s = &__start_table;
  char *e = &__stop_table;
  while (s < e) {
    printf("%d\\n", *s);
    s++;
  }
  return 0;
}
```

Linker defines a variable __start_<section> and __stop_<section> if the linker is going to create an output section that matches a 'C' identifier.

Using the snippet above, and a simple assembly file, you will be able to iterate over the binary file in 'C' code

Where should .note.gnu.property section map to in elf

The .note.gnu.property section is mapped to PT_NOTE segment by default which should have only read permission.

How to garbage-collect symbols in a shared library?

One way to garbage-collect symbols in a shared library is to mark the symbols as hidden. Hidden symbols are garbage collected from a shared library if they are not used from any of the entry symbols in the shared library (entry symbols are symbols that are marked with visibility default).

Example:

```
cat >1.c <<\EOF
// can not be garbage-collected
int foo() {
  return 1;
}

// can be garbage-collected
__attribute__((visibility("hidden")))
int bar() {
  return 2;
}

int baz() {
  return 0;
}
EOF

clang -target hexagon -o 1.o 1.c -c -fPIC -ffunction-sections
hexagon-link -o 1.elf 1.o --gc-sections --print-gc-sections -e baz
```

Running the above script gives the below output:

```
cat
clang -target hexagon -c 1.c -ffunction-sections -fPIC
hexagon-link 1.o -shared --gc-sections --print-gc-sections -e baz
Trace: GC : 1.o[.text]
Trace: GC : 1.o[.text.foo]
```

We can see that ‘foo’, a hidden symbol, got garbage-collected during the link process as it was not reachable from the entry point (baz).

How are zero-sized sections handled?

- A relocation refers to the zero sized section
- A symbol is present in the zero sized section

Note

Releases

This change is observed on the below toolchain / patch releases on Hexagon:

- Hexagon 8.7.04

Releases that will be published after summer of 2023 will also have this support.

Relocation referring to a zero sized section

In the below example, the section .text.baz has size zero and is a relocation target for .text.c1, hence .text.baz is useful and is made a part of the layout.

Example:

The example illustrates a reference to the section .text.baz references by the section .text.c1

```
cat > x.s << \EOF
.section .text.foo
.global foo
.set foo, bar
.section .text.bar
bar:
```

```
.word 100
.section .text.baz
.section .text.c1
.word .text.baz
.section .text.empty
EOF

cat > main.c << \EOF
int main() { return foo(); }
EOF

cat > script.t << \EOF
SECTIONS {
 . = 0x10000;
 .foo : {
  *(.text.*)
  *(.text)
 }
}
EOF
```

Compile and Link step

```
$ clang -c x.s  main.c
$ ld.eld main.o x.o -T script.t -Map x
```

Image layout

The image layout is adjusted to keep **.text.baz** in the output, as there is a relocation that is referred from the section .text.c1

```
# Output Section and Layout
.(0x10000) = 0x10000; # . = 0x10000; script.t

.foo    0x10000 0x24 # Offset: 0x1000, LMA: 0x10000, Alignment: 0x10, Flags: SHF_ALLOC|SHF_EXECINSTR, Type: SHT_PROGBITS
*(.text.*) #Rule 1, script.t [9:0]
.text.foo       0x10000 0x0     x.o     #SHT_PROGBITS,SHF_ALLOC|SHF_EXECINSTR,1
.text.bar       0x10000 0x4     x.o     #SHT_PROGBITS,SHF_ALLOC|SHF_EXECINSTR,1
        0x10000         bar
        0x10000         foo
.text.baz       0x10004 0x0     x.o     #SHT_PROGBITS,SHF_ALLOC|SHF_EXECINSTR,1
.text.c1        0x10004 0x4     x.o     #SHT_PROGBITS,SHF_ALLOC|SHF_EXECINSTR,1
*(.text) #Rule 2, script.t [2:0]
PADDING_ALIGNMENT       0x10008 0x8     0x0
.text   0x10010 0x14    main.o  #SHT_PROGBITS,SHF_ALLOC|SHF_EXECINSTR,16
        0x10010         main
*(.foo) #Rule 3, Internal-LinkerScript (Implicit rule inserted by Linker) [0:0]
```

The section has associated symbols

Sections containing symbols are preserved in the layout. Such sections are not considered to be empty sections.

Warning

Such cases should be avoided at all times due to the following reasons

- The code is very sensitive to layout for it to work properly.
- When link time garbage collection is enabled, this may end up garbage collecting sections which would rather be needed at runtime.

Example:

Warning

In the below example

- The code is very sensitive to layout for it to work properly because the program relies on .text.foo and .text.bar to be placed together
- When link time garbage collection is enabled, this may end up garbage collecting sections which would rather be needed at runtime
    - In the example below the linker may end up garbage collecting .text.bar

```
cat > x.s << \!
.section .text.foo
.type foo, @function
.global foo
foo:
.section .text.bar
.type bar, @function
.global bar
bar:
nop
!

cat > main.c << \!
int main() { return foo(); }
!

cat > script.t << \!
SECTIONS {
 . = 0x10000;
 .text: {
  *(.text.*)
  *(.text)
 }
}
!
```

Compile and link step

```
$ clang -c x.s  main.c
$ ld.eld main.o x.o -T script.t -Map x
```

Image Layout

```
# Output Section and Layout
.(0x10000) = 0x10000; # . = 0x10000; script.t

.text   0x10000 0x24 # Offset: 0x1000, LMA: 0x10000, Alignment: 0x10, Flags: SHF_ALLOC|SHF_EXECINSTR, Type: SHT_PROGBITS
*(.text.*) #Rule 1, script.t [6:0]
.text.foo       0x10000 0x0     x.o     #SHT_PROGBITS,SHF_ALLOC|SHF_EXECINSTR,1
        0x10000         foo
.text.bar       0x10000 0x4     x.o     #SHT_PROGBITS,SHF_ALLOC|SHF_EXECINSTR,1
        0x10000         bar
*(.text) #Rule 2, script.t [2:0]
PADDING_ALIGNMENT       0x10004 0xc     0x0
.text   0x10010 0x14    main.o  #SHT_PROGBITS,SHF_ALLOC|SHF_EXECINSTR,16
        0x10010         main
*(.text) #Rule 3, Internal-LinkerScript (Implicit rule inserted by Linker) [0:0]
```

Can a zero-sized section ever affect the layout? - Yes it can

Example:

```
cat > x.s << \EOF
.section .text.foo
```

```
.global foo
.set foo, bar
.section .text.bar
bar:
.word 100
.section .text.baz
.p2align 4
.section .text.baz1
.section .text.baz2
.section .text.c1
.word .text.baz
.section .text.empty
EOF

cat > main.c << \EOF
int main() { return foo(); }
EOF

cat > script.t << \EOF
SECTIONS {
    . = 0x10003;
    .baz :  {
        . = 0x10005;
        *(.text.baz1)
        . = . + 16;
        *(.text.baz)
        . = . + 28;
        *(.text.baz2)
        . = . + 40;
    }
    .foo : {
        *(.text.foo)
        *(.text*)
    }
}
EOF
```

Compile and Link Step:

```
$clang -c x.s  main.c
$ld.eld main.o x.o -T script.t -Map x.map
```

Image Layout

Image Layout from xmap:

```
# Output Section and Layout
.(0x10003) = 0x10003; # . = 0x10003; script.t

.baz    0x10010 0x54 # Offset: 0x1010, LMA: 0x10010, Alignment: 0x10, Flags: SHF_ALLOC|SHF_EXECINSTR, Type: SHT_PROGBITS
        .(0x10005) = 0x10005; # . = 0x10005; script.t
*(.text.baz1) #Rule 1, script.t [1:0]
        .(0x100010015) = .(0x100010005) + 0x10; # . = . + 0x10; script.t
*(.text.baz) #Rule 2, script.t [1:0]
PADDING_ALIGNMENT       0x10015 0xb     0x0
.text.baz       0x10020 0x0     x.o     #SHT_PROGBITS,SHF_ALLOC|SHF_EXECINSTR,16
        .(0x1003c) = .(0x10020) + 0x1c; # . = . + 0x1c; script.t
*(.text.baz2) #Rule 3, script.t [1:0]
PADDING 0x100010005     0x10    0x0
PADDING 0x1003c 0x28    0x0
        .(0x10064) = .(0x1003c) + 0x28; # . = . + 0x28; script.t
*(.baz) #Rule 4, Internal-LinkerScript (Implicit rule inserted by Linker) [0:0]


.foo    0x10070 0x1c # Offset: 0x1070, LMA: 0x10070, Alignment: 0x10, Flags: SHF_ALLOC|SHF_EXECINSTR, Type: SHT_PROGBITS
*(.text.foo) #Rule 5, script.t [1:0]
.text.foo       0x10070 0x0     x.o     #SHT_PROGBITS,SHF_ALLOC|SHF_EXECINSTR,1
```

```
*(.text*) #Rule 6, script.t [9:0]
.text   0x10070 0x14    main.o  #SHT_PROGBITS,SHF_ALLOC|SHF_EXECINSTR,16
        0x10070         main
.text.bar       0x10084 0x4     x.o     #SHT_PROGBITS,SHF_ALLOC|SHF_EXECINSTR,1
        0x10084         bar
        0x10084         foo
.text.c1        0x10088 0x4     x.o     #SHT_PROGBITS,SHF_ALLOC|SHF_EXECINSTR,1
```

Here because of p2align directive for .text.baz the PADDING_ALIGNMENT of size 0xb at address 0x100015 was added

Debugging zero-sized sections related issues

ELD has 2 ways that help in debugging issues wrt to zero-sized sections

- Linker diagnostics as warnings tied to options `-Wall` and `-Wzero-sized-sections`. Look for traces containing `Warning: Zero sized fragment`.

```
cat > x.s << \EOF
.section .text.baz
.type baz, %function
.global baz
baz:
.section .text.bar
.type bar, %function
.global bar
bar:
nop
.section .foo,"a",%progbits
.local sym
sym:
.size sym, 40
EOF

cat > main.c << \EOF
int main() { return baz(); }
EOF

cat > script.t << \EOF
SECTIONS {
 . = 0x10000;
 .text: {
  *(.text.*)
  *(.text)
 }
}
EOF

$ clang -c x.s  main.c
$ ld.eld main.o x.o -T script.t -Map x
Warning: Zero sized fragment .foo for non zero sized symbol sym from input file x.o
```

- Linker stats emitted in the map file as * ZeroSizedSections * ZeroSizedSectionsGarbageCollected

```
cat > x.s << \!
.section .text.foo
.global foo
.set foo, bar
.section .text.bar
bar:
.word 100
.section .text.baz
.section .text.c1
```

```
.word .text.baz
.section .text.empty
 !

cat > main.c << \!
int main() { return foo(); }
!

cat > script.t << \!
SECTIONS {
 . = 0x10000;
 .text: {
  *(.text.*)
  *(.text)
}
}
!

$ clang -c x.s  main.c
$ ld.eld main.o x.o -T script.t -Map x
$ cat x
...
# LinkStats Begin
# ObjectFiles : 2
# LinkerScripts : 2
# ThreadCount : 64
# NumInputSections : 29
# ZeroSizedSections : 17
# SectionsGarbageCollected : 5
# ZeroSizedSectionsGarbageCollected : 4
# NumLinkerScriptRules : 2
# NumOutputSections : 6
# NumOrphans : 5
# NoRuleMatches : 2
# LinkStats End
...
```

How to interpret the new trampoline naming convention?

The Linker trampoline naming convention has been updated to the following format

trampoline_for_<target symbol name>_from_<source input section name>_<Input file ordinality of the source input section>#<Optional: relocation addend>_<Optional: Additional trampoline count>

The trampoline naming convention can be broadly classified into 4 cases:

Case 1:

Format: trampoline_for_<target symbol name>_from_<source input section name> _<Input file ordinality of the source input section>

Example:

```
cat > main.c << \!
int baz() {
  return 0;
}
int main ()
{
  return baz() ;
}
!
```

```
cat > script.t << \!
SECTIONS
{
  .text           :
  {
    *(.text.main)
  } =0x00c0007f
  . = 0x08000000;
  .text.baz :
  {
    *(.text.baz)
  }
}
!
```

Compile and Link Steps:

```
$ clang -c  main.c -ffunction-sections
$ ld.eld main.o -T script.t -Map x
```

Symbols from readelf:

```
$ llvm-readelf -s a.out
Symbol table '.symtab' contains 9 entries:
   Num:    Value  Size Type    Bind   Vis       Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT   UND
     1: 00000000     0 SECTION LOCAL  DEFAULT     1 .text
     2: 00000000     0 SECTION LOCAL  DEFAULT     3 .comment
     3: 08000000     0 SECTION LOCAL  DEFAULT     2 .text.baz
     4: 00000000     0 FILE    LOCAL  DEFAULT   ABS main.c
     5: 00000014     8 FUNC    LOCAL  DEFAULT     1 trampoline_for_baz_from_.text.main_25
     6: 00000000    20 FUNC    GLOBAL DEFAULT     1 main
     7: 08000000    12 FUNC    GLOBAL DEFAULT     2 baz
     8: 08000011     0 NOTYPE  GLOBAL DEFAULT   ABS __end
```

Trampoline Symbol: `trampoline_for_baz_from_.text.main_25` → call baz from main

This is the most vanilla case where the target symbol name, source input section and input file are used to coin the trampoline name.

Case2:

Format: trampoline_for_<target symbol name>_from_<source input section name>_ <Input file ordinality of the source input section>_<Additional trampoline count>

Example:

```
cat > main.c << \!
int far() {
  return 0;
}
int callfar() {
  return far() + far();
}
int main ()
{
  return callfar();
}
!

cat > noreuse << \!
{
    far;
}
```

```
!

cat > script.t << \!
SECTIONS
{
  .text           :
  {
    *(.text.main)
    *(.text.callfar)
  } =0x00c0007f
  . = 0x08000000;
  .text.far :
  {
    *(.text.far)
  }
}
!
```

Compile and Link Steps:

```
$ clang -c  main.c -ffunction-sections
$ ld.eld main.o -T script.t -Map x -no-reuse-trampolines-file=noreuse
```

Symbols from readelf:

```
$ llvm-readelf -s a.out
Symbol table '.symtab' contains 11 entries:
   Num:    Value  Size Type    Bind   Vis       Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT   UND
     1: 00000000     0 SECTION LOCAL  DEFAULT     1 .text
     2: 00000000     0 SECTION LOCAL  DEFAULT     3 .comment
     3: 08000000     0 SECTION LOCAL  DEFAULT     2 .text.far
     4: 00000000     0 FILE    LOCAL  DEFAULT   ABS main.c
     5: 00000040     8 FUNC    LOCAL  DEFAULT     1 trampoline_for_far_from_.text.callfar_25
     6: 00000048     8 FUNC    LOCAL  DEFAULT     1 trampoline_for_far_from_.text.callfar_25_1
     7: 00000000    20 FUNC    GLOBAL DEFAULT     1 main
     8: 00000020    32 FUNC    GLOBAL DEFAULT     1 callfar1
     9: 08000000    12 FUNC    GLOBAL DEFAULT     2 far
    10: 08000011     0 NOTYPE  GLOBAL DEFAULT   ABS __end
```

Trampoline Symbol: `trampoline_for_far_from_.text.callfar1_25_1` → call far from callfar1

The additional trampoline count is added as a suffix in cases where the reuse of the existing trampoline is not possible.

> ## Note
>
> The reuse was not possible because of `-no-reuse-trampolines-file=noreuse`.

Case3:

Format: trampoline_for_<target symbol name>_from_<source input section name>_#(relocation addend)

Example:

```
cat > main.c << \!
static int baz() {
  return 0;
}
int main ()
{
  return baz();
```

```
}
!

cat > script.t << \!
SECTIONS
{
  .text           :
  {
    *(.text.main)
  } =0x00c0007f
  . = 0x08000000;
  .text.baz :
  {
    *(.text.baz)
  }
}
!
```

Compile and Link Steps:

```
$ clang -c  main.c -ffunction-sections
$ ld.eld main.o -T script.t -Map x
```

Symbols from readelf:

```
$ llvm-readelf -s a.out
Symbol table '.symtab' contains 9 entries:
   Num:    Value  Size Type    Bind   Vis       Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT   UND
     1: 00000000     0 SECTION LOCAL  DEFAULT     1 .text
     2: 00000000     0 SECTION LOCAL  DEFAULT     3 .comment
     3: 08000000     0 SECTION LOCAL  DEFAULT     2 .text.baz
     4: 00000000     0 FILE    LOCAL  DEFAULT   ABS main.c
     5: 00000014     8 FUNC    LOCAL  DEFAULT     1 trampoline_for_.text.baz_from_.text.main_25#(0)
     6: 08000000    12 FUNC    LOCAL  DEFAULT     2 baz
     7: 00000000    20 FUNC    GLOBAL DEFAULT     1 main
     8: 08000011     0 NOTYPE  GLOBAL DEFAULT   ABS __end
```

Trampoline Symbol: `trampoline_for_.text.bar_from_.text.main_25#(0)` → call to bar from main here (`#(0)` represents the relocation addend)

The relocation addend 0 is added to the trampoline symbol name in case the trampoline jumps to the section symbol for the symbol

Case4:

Format: trampoline_for_<target symbol name>_from_<source input section name> _<Input file ordinality of the source input section>#<relocation addend> _<Additional trampoline count>

Example:

```
cat > main.c << \!
static int baz() {
  return 0;
}
int main ()
{
  return baz() + baz();
}
!

cat > noreuse << \!
{
    baz;
    .text.baz;
```

```
}
!

cat > script.t << \!
SECTIONS
{
  .text           :
  {
    *(.text.main)
  } =0x00c0007f
  . = 0x08000000;
  .text.baz :
  {
    *(.text.baz)
  }
}
!
```

Compile and Link Steps:

```
$ clang -c  main.c -ffunction-sections
$ ld.eld main.o -T script.t -Map x -no-reuse-trampolines-file=noreuse
```

Symbols from readelf:

```
$ llvm-readelf -s a.out
Symbol table '.symtab' contains 10 entries:
   Num:    Value  Size Type    Bind   Vis       Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT   UND
     1: 00000000     0 SECTION LOCAL  DEFAULT     1 .text
     2: 00000000     0 SECTION LOCAL  DEFAULT     3 .comment
     3: 08000000     0 SECTION LOCAL  DEFAULT     2 .text.baz
     4: 00000000     0 FILE    LOCAL  DEFAULT   ABS main.c
     5: 00000028     8 FUNC    LOCAL  DEFAULT     1 trampoline_for_.text.baz_from_.text.main_25#(0)
     6: 00000030     8 FUNC    LOCAL  DEFAULT     1 trampoline_for_.text.baz_from_.text.main_25#(0)_1
     7: 08000000    12 FUNC    LOCAL  DEFAULT     2 baz
     8: 00000000    40 FUNC    GLOBAL DEFAULT     1 main
     9: 08000011     0 NOTYPE  GLOBAL DEFAULT   ABS __end
```

Trampoline Symbol: `trampoline_for_.text.bar_from_.text.main_25#(0)_1` → 2nd call for bar from main

The duplicate trampoline count was added as a prefix since the symbol name .text.bar was added to the noreuse list.

Benefits of the new trampoline naming convention:

The new format has benefits as follows

- Helps determine/infer a huge amount of info about the target symbol and the source sections inherently
- Helps make diff between map files easier since the naming convention is now more context-based

Understanding “loadable segments are unsorted by virtual address” warning from llvm-readelf

The llvm-readelf can sometimes emit a warning: “loadable segments are unsorted by virtual address”. This occurs in very specific scenarios where below items are involved

- Dynamic sections
- “-shared” in linker commandline
- The virtual addresses of dynamic sections not in increasing order

Below is a small example to illustrate this:

Example:

```
cat > 1.c << \!
int foo() { return 0; }
!

cat > script.t << \!
PHDRS {
    A PT_LOAD;
    B PT_LOAD;
    DYN PT_DYNAMIC;
}

SECTIONS {
      .dynsym (0x900) : { *(.dynsym) } :A
      .dynamic (0x800) : { *(.dynamic) } :B :DYN
}
!

clang -target hexagon -c 1.c -ffunction-sections
hexagon-link 1.o -T script.t -shared
llvm-readelf -S -W a.out
```

Important thing to note in above script is the non-increasing order of virtual addresses of the dynamic sections

When the *"llvm-readelf"* in above script runs, a specific code branch for dynamic images in ELF.cpp emits this warning

Output:

```
llvm-readelf: warning: 'a.out': loadable segments are unsorted by virtual address
```

This warning in its current form is rather vague and very general as it doesn't inform the user about the specific dynamic builds this warning is meant for.

Currently, there is **no** option available to suppress this warning.

Linker overlap checks

Linker has been improved to detect overlaps in the image layout. Linker does this by default. The following overlaps are detected by the linker. * Virtual address overlaps * Physical address overlaps * File offset overlaps

If the overlaps are known and you want to turn this behavior OFF, you can use *–no-check-sections* flag.

Input File to Linker

The linker takes the following kinds of input files as input :-

- Object files
- Shared libraries
- Linker scripts
- Multiple command line options
- **Other kinds of input files such as**
    - Extern list
    - dynamic list
    - version scripts
    - linker plugin configuration files

Most recently the linker also was modified to support taking fully built static executables as part of the link step.

ELF executable as input files to the linker

Linker allows a fully built static executable as input to the linker.

The LLVM community linker does not support this option. GNU linker used to support but now does not as per the below bug

https://sourceware.org/bugzilla/show_bug.cgi?id=26223

ELD allows fully built static executables as input to the linker as per the below example.

```
cat > script.t << \!
SECTIONS {
  .text : {
    *(.text)
  }
  . = 0x2000;
  anotherelf : {
    anotherexec.elf(.bar)
  }
}
!

cat > foo.c << \!
extern int bar();
int foo() { return bar(); }
!

cat > bar.c << \!
__attribute__((section(".bar"))) int bar() { return 0; }
!

$clang -O2 -c bar.c -fno-exceptions -fno-asynchronous-unwind-tables
$clang -c foo.c -fno-exceptions -fno-asynchronous-unwind-tables
$link bar.o -o anotherexec.elf -Ttext=0x2000
$link foo.o anotherexec.elf -T script.t
```

Warning

This is strongly discouraged by the community and got accidentally removed in the GNU linker.

It is upto the user to make sure that the linker script places the executable code previously linked at the same address.

Linking dynamic executables is not possible with this functionality and will not be supported

Note

If you really need to link with executables, we recommend to use the –just-symbols option

Linker Script General Questions

Debugging section placements

Linker Script Rules

Input sections can be stored in files or archives. They are specified in the SECTIONS command with the following syntax:

[path][archive:][file](section…)

The standard wildcard characters (*, ?, etc.) can be used anywhere in an input section specification to do the following:

- Specify multiple paths, archives, or files where sections will be searched for
- Specify multiple sections as input sections

Input section patterns

Specification	Description
dir/subdir/init.lib:init.o(.text.*)	Specify one or more .text. sections from a specific object file (init.o) in a specific archive (init.lib)
dir/subdir/init.lib:(.text.*)	**Specify one or more .text. sections from any object file in the** specified archive (init.lib)
dir/subdir/*:(.text.*)*	Specify one or more .text. sections from any archive in the specified directory (dir/subdir)
(.text.)	Specify one or more .text. sections from any archive or object file in the entire file system

Fill Values

Output Section Fill

You can set the fill pattern for an entire section by using '=fillexp'. fillexp is an expression.

Any otherwise unspecified regions of memory within the output section (for example, gaps left due to the required alignment of input sections) will be filled with the value, repeated as necessary.

In all cases, the number specified by the fillexp is big-endian.

Here is a simple example:

```
SECTIONS { .text : { *(.text) } =0x90909090 }
```

I am getting an error, no linker script rule for ".bss"

The way that you would investigate this is to figure out if there is an actual .bss section that is present in the input files, not selected by any linker script rule. Emit the Map file and look at the section .bss and see what kind of sections are present.

- If it shows that a input file with a particular section is being listed and not present in any of the patterns, you will likely need to go and add a rule such as `*(.bss)` so that you are sure of where you want to place it.
- If all the input file and sections are selected, you should go and look at if the section corresponds to a common symbol (common symbol). You will most probably be missing a rule such as `*(COMMON)`.
- Try to do the link step again with any of the above changes and your error should have gone.

I am getting an error, no linker script rule for ".data.bar"

This is most often a problem if there is a compiler flag that has changed in your environment or a change of tools. You would need to fix the linker script and add a rule as below.

- *(.data.bar)

Is there a way to find all used/unused rules in the Linker script ?

You can use a simple grep pattern to check for used/unused rules in the Map file.

UnUsed Rules

```
grep "Rule.*\[0:" link.map | grep -v Implicit
```

Used Rules

```
grep "Rule.*\[[1-9]*:" link.map  | grep -v Implicit
```

How do I exclude sections from object files/archive files

Excluding sections from object files

The example illustrates users trying to exclude all text sections from being placed in section .text1 and when the input file is an object file.

```
cat > a.c << \!
int foo() { return 0; }
int bar() { return 0; }
!


cat > script.t << \!
SECTIONS {
  .text1 : {
    *(EXCLUDE_FILE(a.o) .text.*)
  }
  .text2 : {
    *(.text*)
  }
}
!

clang -target hexagon -ffunction-sections -fdata-sections -c a.c -G0
rm -f lib1.a
hexagon-ar cr lib1.a a.o
hexagon-link -T script.t a.o -Map x
```

Exclude all patterns of text section from one file

The example illustrates an user trying to exclude all text sections from being placed in section .text1.

In this example, all code from lib1.a/a.o is not emitted to the the .text1 section

```
cat > a.c << \!
int foo() { return 0; }
int bar() { return 0; }
!


cat > script.t << \!
```

```
SECTIONS {
  .text1 : {
    *lib1.a:(EXCLUDE_FILE(a.o) .text.*)
  }
  .text2 : {
    *(.text*)
  }
}
!

clang -target hexagon -ffunction-sections -c a.c
rm -f lib1.a
hexagon-ar cr lib1.a a.o
hexagon-link -T script.t --whole-archive lib1.a
```

Excluding all patterns of text section from more than one file

The example illustrates an user trying to exclude all text sections from being placed in section .text1.

In this example, all code from lib1.a/a.o and lib1.a/b.o is not emitted to the .text1 section

```
cat > a.c << \!
int foo() { return 0; }
int bar() { return 0; }
!

cat > b.c << \!
int baz() { return 0; }
!


cat > script.t << \!
SECTIONS {
  .text1 : {
    *lib1.a:(EXCLUDE_FILE(a.o b.o) .text.*)
  }
  .text2 : {
    *(.text*)
  }
}
!

clang -target hexagon -ffunction-sections -fdata-sections -c a.c -G0
clang -target hexagon -ffunction-sections -fdata-sections -c b.c -G0
rm -f lib1.a
hexagon-ar cr lib1.a a.o b.o
hexagon-link -T script.t --whole-archive lib1.a -Map x
```

Specifying multiple exclude patterns

You can specify more than one EXCLUDE_FILE pattern in a linker script rule.

This example below illustrates a way that the user can specify multiple EXCLUDE_FILE linker script commands.

```
cat > a.c << \!
int foo() { return 0; }
int bar() { return 0; }
!

cat > b.c << \!
int baz() { return 0; }
!
```

```
cat > script.t << \!
SECTIONS {
  .text1 : {
    *lib1.a:(EXCLUDE_FILE(a.o) .text.foo EXCLUDE_FILE(a.o) .text.bar EXCLUDE_FILE(b.o) .text.baz)
  }
  .text2 : {
    *(.text*)
  }
}
!

clang -target hexagon -ffunction-sections -fdata-sections -c a.c -G0
clang -target hexagon -ffunction-sections -fdata-sections -c b.c -G0
rm -f lib1.a
hexagon-ar cr lib1.a a.o b.o
hexagon-link -T script.t --whole-archive lib1.a -Map x
```

Common mistakes with exclude files

Specifying more than one pattern

> **Warning**
>
> A common mistake users do is when they specify more than one pattern with one occurrence of an EXCLUDE_FILE rule. Only the first pattern following the EXCLUDE_FILE rule is used for matching

In the above example if the user did the following

```
SECTIONS {
  .text1 : {
    *lib1.a:(EXCLUDE_FILE(a.o) .text.foo .text.bar EXCLUDE_FILE(b.o) .text.baz)
  }
  .text2 : {
    *(.text*)
  }
}
```

> **Note**
>
> The function specified by .text.bar would be placed in .text1 since .text.bar is following the pattern .text.foo.

Not specifying all the files in the EXCLUDE_FILE pattern

When the file pattern does not match the filename specified in EXCLUDE_FILE, pattern following the EXCLUDE_FILE is used to select input sections that do not originate from that file.

Below example illustrates that. In this example .text.baz from file b.o is selected by the rule, which results in .text1 housing .text.baz.

```
cat > a.c << \!
int foo() { return 0; }
int bar() { return 0; }
!

cat > b.c << \!
int baz() { return 0; }
!
```

```
cat > script.t << \!
SECTIONS {
  .text1 : {
    *lib1.a:(EXCLUDE_FILE(a.o) .text.*)
  }
  .text2 : {
    *(.text*)
  }
}
!

clang -target hexagon -ffunction-sections -fdata-sections -c a.c -G0
clang -target hexagon -ffunction-sections -fdata-sections -c b.c -G0
rm -f lib1.a
hexagon-ar cr lib1.a a.o b.o
hexagon-link -T script.t --whole-archive lib1.a -Map x
```

Specifying EXCLUDE_FILE directive that applies to multiple section patterns

We can specify EXCLUDE_FILE directive that applies to multiple section patterns by placing EXCLUDE_FILE before the corresponding <file-patterns>(<section-patterns>).

For Example:

```
#!/usr/bin/env bash

cat >1.c <<\EOF
int foo() { return 1; }
int baz() { return 3; }
EOF

cat >2.c <<\EOF
int bar() { return 3; }
EOF

cat >script.t <<\EOF
SECTIONS {
  FOO_BAZ : { EXCLUDE_FILE(2.o) *(.text.foo .text.baz .text.bar) }
  BAR : { *(*.text*) }
}
EOF

clang -target hexagon -o 1.o 1.c -c -ffunction-sections
clang -target hexagon -o 2.o 2.c -c -ffunction-sections
hexagon-link -o a.out 1.o 2.o -T script.t
```

In the above example, EXCLUDE_FILE(2.o) applies to all the section patterns present in the sub-rule: *(.text.foo .text.baz .text.bar)

NOCROSSREFS

The NOCROSSREFS command takes a list of output section names. If the linker detects any cross references between the sections, it reports an error and returns a non-zero exit status. Note that the NOCROSSREFS command uses output section names, not input section names.

```
cat > 1.c << \!
extern int bar();
int foo() { return bar(); }
!
```

```
cat > 2.c << \!
int bar() { return 0; }
!

cat > script.t << \!
NOCROSSREFS(.foo .bar)
SECTIONS {
  .foo : { *(.text.foo) }
  .bar : { *(.text.bar) }
}
!

clang -c 1.c 2.c -ffunction-sections
ld.eld 1.o 2.o -T script.t
```

The above linker script usage of **NOCROSSREFS** produces an error because content in output section foo is calling into bar.

Example error:

```
Error: 1.o:(.text.foo:0x8): prohibited cross reference from .foo to `bar'(2.o) in .bar
```

How to discard an input section?

Input sections can be explicitly discarded from the output image by assigning the input section to a special output section named '/DISCARD/'. It can also discard sections marked with the ELF flag SHF_GNU_RETAIN which would otherwise have been saved from linker garbage collection.

For Example:

```
cat >1.c <<\EOF
__attribute__((retain)) int foo() { return 0; }
int bar() { return 0; }
int main() { return 0; }
EOF

cat >1.linker.script <<\EOF
SECTIONS {
  /DISCARD/ : { *(.text.foo) }
}
EOF

clang -target hexagon -o 1.o -c 1.c -ffunction-sections
hexagon-link -o 1.elf 1.o --gc-sections --print-gc-sections -e main
hexagon-link -o 1.with_script.elf 1.o -T 1.linker.script --gc-sections --print-gc-sections -e main --trace-section .text.foo
```

hexagon-link -o 1.elf 1.o –gc-sections –print-gc-sections -e main

Running the above command gives the following output:

Trace: GC : 1.o[.text] | Trace: GC : 1.o[.text.bar]

Please note that '.text.foo' is preserved even though there are no references to the function foo from the entry point. It is because '.text.foo' is marked with the ELF flag SHF_GNU_RETAIN using '__attribute__((retain))'.

The above command can be run with the verbose option to see the 'retain section…' diagnostic for '.text.foo':

Verbose: Retaining section .text.foo from file 1.o

'.text.foo' can be discarded by assigning it to /DISCARD/ special output section.

hexagon-link -o 1.with_script.elf 1.o -T 1.linker.script –gc-sections –print-gc-sections -e main –trace-section .text.foo

Running the above command gives the following output:

Note: Input Section : .text.foo InputFile : 1.o Alignment : 0x10 Size : 0xc Flags : SHF_ALLOC|SHF_EXECINSTR | Note: Input Section : .text.foo Symbol : foo | Note: Section : .text.foo is being discarded. Section originated from input : 1.o | Trace: GC : 1.o[.text] | Trace: GC : 1.o[.text.bar]

We can see in the 'Note' diagnostic that '.text.foo' has been discarded.

Marking a loadable output section as non loadable

Linker now supports loadable sections to be listed after non loadable sections, so the linker is able to place them in a loadable segment.

There have been various assumptions in the past that customers relied on their linker scripts, about linker not supporting loadable sections after non loadable sections.

If these assumptions are still valid for your use case(s) :

- Make sure to list the ouput section to be of type INFO * Example : OutputSection **(INFO)** * This will mark the output section non loadable
- Always inspect the output of your section to segment mapping before loading the image * llvm-readelf -l -W *outputfile*

If you have been waiting for this feature, remove the virtual address assigned to the section.

Most users have these output sections listed in the linker script to use a virtual address of 0. You will need to remove it.

Always inspect the output image to check for any discrepancies.

What does KEEP mean for input section descriptions in /DISCARD/? (Added in HEX 8.8)

KEEP marks all the input sections matched by an input section description as an entry section. Entry sections are not subject to garbage-collection and are used to calculate live-edges for garbage-collection. However, entry sections can still be discarded. Therefore, behavior of KEEP is the same for input section description of all output sections, including the /DISCARD/ output section. Important thing to remember is when KEEP is used for input section descriptions in the/DISCARD/ output section then the matched input sections are treated as entry sections for live-edge computation but are still discarded nonetheless.

Example:

```
#!/usr/bin/env bash

cat >1.c <<\EOF
int bar() {
  return 1;
}

int foo() {
  return bar();
}

int baz() {
  return 3;
}

int abc() {
  return 5;
}

int main() {
  return baz();
}
EOF

cat >script.t <<\EOF
SECTIONS {
  /DISCARD/ : { KEEP(*(*.foo)) }
  .text : { *(.text.*) }
}
EOF

clang -target hexagon -o 1.o 1.c -c -ffunction-sections -fdata-sections
hexagon-link -o 1.elf 1.o -T script.t --gc-sections --print-gc-sections -e main -Map 1.map.tx
```

In this example, bar is not garbage-collected because it is reachable by foo. foo is an entry section due to KEEP specifier even though it is discarded. Map file can be used to confirm the behavior.

How to build executables by linking symbols from an another ELF file?

Users can build executables by linking symbols from another ELF file by using the SYMDEF feature.

ELD mimics the functionality available in the ARM linker for this.

This method is also used by ARM baremetal builds for linking against symbols that exist in another image.

SymDef file

A symdef file is a file that consists of all global symbols that can be used in future link steps.

The linker produces a symdefs file during a successful final link stage. It is not produced for partial linking or for unsuccessful final linking.

The symdef file is produced by using the linker symdef option.

Example:

```
cat >1.c << \!
int foo() {return bar(); }
!

cat > 2.c << \!
int boo() { return baz(); }
!

cat > 3.c << \!
int bar() { return 0; }
int baz() { return 0; }
!

cat > script.t << \!
SECTIONS {
  .foo : { *(.text.*) }
}
!

cat > image.t << \!
SECTIONS {
  . = 0xF0000000;
  .text : { *(.text*) }
}
!

$CLANG -c 1.c -ffunction-sections
$CLANG -c 2.c -ffunction-sections
$CLANG -c 3.c -ffunction-sections
$LINK -o otherimage.elf 3.o --symdef-file r.symdef -T image.t
# otherimage.elf should be preserved for the below link command to work
$LINK 1.o 2.o r.symdef -T script.t -o out.elf
```

The symdef file gets produced that contains the address of bar and baz in the image.

```
#<SYMDEFS>#
#DO NOT EDIT#
0xf0000000      FUNC    bar
0xf0000010      FUNC    baz
```

Later the file is used in the secondary link to generate the complete binary.

How to specify Load Memory Address (LMA) of a section?

Load Memory Address (LMA) of a section can be specified using the ‘AT’ command. Please note that it is different from the virtual memory address.

```
FOO 0x1000 : AT(0x4000) {
   *(*foo*)
 }
```

In the above example, ‘0x1000’ is the virtual memory address (VMA) of the output section ‘FOO’, and ‘0x4000’ is the load memory address (lma).

A concrete example:

```
#!/usr/bin/env bash

cat >1.c <<\EOF
int foo() { return 1; }
int bar() { return 3; }
EOF

cat >script.t <<\EOF
SECTIONS {
  FOO 0x1000 : AT(0x4000) {
    *(*foo*)
  }

  BAR 0x2000 : AT(0x6000) {
    *(*bar*)
  }
}
EOF

cat >script.without_at.t <<\EOF
SECTIONS {
  FOO 0x1000 : { *(*foo*) }
  BAR 0x2000 : { *(*bar*) }
}
EOF

CCs=("clang -target hexagon" clang-15 clang-15)
LDs=(hexagon-link ld.bfd ld.lld)
SFs=(eld bfd lld)

clang -target hexagon -o 1.o 1.c -c -ffunction-sections
hexagon-link -o 1.elf 1.o -T script.t
hexagon-llvm-objdump -h 1.elf

# Output:
# 1.eld.elf:     file format elf32-hexagon
#
# Sections:
# Idx Name          Size     VMA      LMA      Type
#  0               00000000 00000000 00000000
#  1 FOO           0000000c 00001000 00004000 TEXT
#  2 BAR           0000000c 00002000 00006000 TEXT
#  3 .comment      00000106 00000000 00000000
#  4 .shstrtab     0000002c 00000000 00000000
#  5 .symtab       00000080 00000000 00000000
#  6 .strtab       00000024 00000000 00000000
```

```
Please note the VMA and LMA of output sections FOO and BAR:

FOO            0000000c 00001000 00004000
BAR            0000000c 00002000 00006000
```

Image layout using LMA

When sections are placed in a segment, the LMA address of the section is calculated as the sum of LMA address of the segment and virtual address offset of the section from the beginning of the segment.

User can query the LMA address of the section using the linker script keyword LOADADDR.

```
cat > 1.c << \!
int bss[100] = { 0 };
int data[100] = { 0 };
int foo() { return 0; }
!

cat > script.t << \!
PHDRS {
  A PT_LOAD;
}

SECTIONS
  .foo : AT(0x1000) { *(.text.foo) }  :A
  .bar : { *(.bss.bss*) }  :A
  .baz : { *(.bss.data*) }  :A
  .nothing : { } :A
  load_addr_baz = LOADADDR(.nothing);
}
!

$clang -c 1.c -ffunction-sections -fdata-sections -G0 -fno-asynchronous-unwind-tables
$link 1.o -T script.t -Map x
```

With the above example you can see that the linker sets the value of load_addr_baz to the load address of the section .nothing.

For Hexagon architecture, you can see that the load address thats calculated accounts to be 0x1330.

```
Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
LOAD           0x001000 0x00000000 0x00001000 0x0000c 0x00330 RWE 0x1000
```

This change in behavior is observed from linkers on release

- Hexagon 8.9 and above
- RISC-V 18.0 release

If you have a linker script that assumes the behavior of LOADADDR, you might want to fix that with the recent change to behavior.

What are BYTE/SHORT/LONG/QUAD/SQUAD linker script commands?

These commands allow including explicit bytes of data in an output section. Each of these commands is followed by an expression in parentheses providing the value to store. The value is stored at the current value of the location counter

The BYTE, SHORT, LONG, and QUAD commands store one, two, four, and eight bytes (respectively). After storing the bytes, the location counter is incremented by the number of bytes stored.

An example demonstrating these commands:

```
#!/usr/bin/env bash
```

```
cat >1.c <<\EOF
int foo() { return 1; }
EOF

cat >script.t <<\EOF
SECTIONS {
  FIVE: { BYTE(0x5) }
  SIXTEEN : { LONG(0x10) }
}
EOF

clang -target hexagon -o 1.o 1.c -c
hexagon-link -o 1.elf 1.o -T script.t
hexagon-readelf -x 'FIVE' 1.elf
hexagon-readelf -x 'SIXTEEN' 1.elf
```

What is the order of linker script assignment evaluations?

Linker script contains assignment expressions. Assignment expressions can be categorized into 3 distinct categories: before-sections, in-sections and after-sections. We will use the term outside-sections to collectively refer before-sections and after-sections categories. When an assignment expression gets evaluated depends on the assignment expression category.

First, all the assignment expressions outside the SECTIONS commands are evaluated and then the assignment expressions within the SECTIONS command are evaluated. There are few catches, which we will soon explore.

Let's look at some concrete examples to understand assignment evaluation order behavior.

Basic

```
#!/usr/bin/env bash

cat >script.t <<\EOF
bar = 0x3;
SECTIONS {
  baz = bar;
}
EOF

touch 1.c
clang -target hexagon -o 1.o 1.c -c
hexagon-link -o 1.out 1.o -T script.t
hexagon-readelf -s 1.out
# Output:
# Symbol table '.symtab' contains 7 entries:
#   Num:    Value  Size Type    Bind   Vis       Ndx Name
#     // ...
#     // ...
#     5: 00000003     0 NOTYPE  GLOBAL DEFAULT   ABS bar
#     6: 00000003     0 NOTYPE  GLOBAL DEFAULT   ABS baz
```

Both bar and baz have the value 0x3 as expected.

Use an after-sections variable to initialize an in-sections variable

```
#!/usr/bin/env bash

cat >script.t <<\EOF
SECTIONS {
  baz = bar;
}
bar = 0x3;
EOF

touch 1.c
clang -target hexagon -o 1.o 1.c -c
hexagon-link -o 1.out 1.o -T script.t
hexagon-readelf -s 1.out
# Output:
# Symbol table '.symtab' contains 7 entries:
#   Num:    Value  Size Type    Bind   Vis       Ndx Name
#     // ...
#     // ...
#     5: 00000003     0 NOTYPE  GLOBAL DEFAULT   ABS bar
#     6: 00000003     0 NOTYPE  GLOBAL DEFAULT   ABS baz
```

Both bar and baz have the value 0x3 even though at first glance it seems that bar is defined later than baz.

That's not true. Remember that all outside-sections assignments are evaluated before in-sections assignments. Hence, bar is evaluated before baz.

Use a variable, that is defined in both before-sections and after-sections assignment, to initialize an in-sections variable

```
#!/usr/bin/env bash

cat >script.t <<\EOF
bar = 0x3;
SECTIONS {
  baz = bar;
}
bar = 0x5;
EOF

touch 1.c
clang -target hexagon -o 1.o 1.c -c
hexagon-link -o 1.out 1.o -T script.t
hexagon-readelf -s 1.out
# Output:
# Symbol table '.symtab' contains 7 entries:
#   Num:    Value  Size Type    Bind   Vis       Ndx Name
#     // ...
#     // ...
#     5: 00000005     0 NOTYPE  GLOBAL DEFAULT   ABS bar
#     6: 00000005     0 NOTYPE  GLOBAL DEFAULT   ABS baz
```

Now, that's a little interesting. If the assignments were evaluated in the natural-order then we would have seen baz value as 0x3 and bar value as 0x5. However, this is not what we have observed.

The reason is that outside-sections assignments are evaluated before in-sections assignments. Hence, the order of evaluations is:

1. bar = 0x3
2. bar = 0x5

3. baz = bar

So remember that the golden rule is outside-sections assignments are evaluated before in-sections assignments.

Using a `-defsym` symbol in linker script assignment expression

```
#!/usr/bin/env bash

cat >script.t <<\EOF
SECTIONS {
  baz = bar;
}
EOF

touch 1.c
clang -target hexagon -o 1.o 1.c -c
hexagon-link -o 1.out 1.o -T script.t --defsym bar=0x5
hexagon-readelf -s 1.out
# Output:
# Symbol table '.symtab' contains 7 entries:
#   Num:    Value  Size Type    Bind   Vis       Ndx Name
#     // ...
#     // ...
#     5: 00000005     0 NOTYPE  GLOBAL DEFAULT   ABS bar
#     6: 00000005     0 NOTYPE  GLOBAL DEFAULT   ABS baz
```

`-defsym` symbols are treated as outside-sections symbols, and hence they are always evaluated before in-sections symbols. This explains the readelf output that we see.

What linker script changes are required for supporting thread-local storage (TLS)?

If any of the linker inputs use thread-local storage (TLS), then some linker script changes are required to correctly support thread-local storage functionality.

To properly understand these linker script changes, it is helpful to understand the TLS functionality and how TLS is allocated and initialized.

Thread-local storage functionality makes a variable local to each thread. This means that each thread has its own copy of the variable. This is unlike ordinary global/static variables that are shared across all threads. The runtime library allocates thread-local storage for each thread and it initializes the thread-local storage to the region pointed by the `PT_TLS` segment. Among other things, `PT_TLS` segment describes to the runtime library where to find the initial contents of the thread-local storage and the alignment requirements.

The runtime library needs `PT_TLS` segment for properly initializing TLS region for each block. Thus, the linker needs to emit `PT_TLS` segment for the images that are utilizing TLS functionality. The linker script changes are required for instructing the linker how to properly emit `PT_TLS` segment. In particular, we need to add a `PT_TLS` program header and put the TLS sections (`.tdata` and `.tbss`) into both the `PT_TLS` and a `PT_LOAD` section. For example:

```
PHDRS {
  ...
  DATA PT_LOAD;
  TLS PT_TLS;
  ...
}

SECTIONS {
  ...
  .tbss : { *(*.tbss) } :DATA :TLS
  .tdata : { *(*.tdata) } :DATA :TLS
  .data : { *(*.data) } :DATA // It is important to specify :DATA here
                              // otherwise ':DATA :TLS' will be assumed
```

```
  ...
}
```

Why am I getting the warning 'Space between archive:member file pattern is deprecated'

Linker script allows to specify archive members in input section descriptions using the *<archive-pattern>:<member-pattern>* syntax. For example in the below linker script *SECTIONS* command snippet, *foo_text* output section contains the *.text** sections from the *foo.o* member of *libfoobar.a* archive.

```
SECTIONS {
  foo_text : { libfoobar.a:foo.o(.text*) }
}
```

Please note that there should be no space between the *<archive-pattern>:* and the *<member-pattern>*. Previously, the linker accepted the space between the two patterns for convenience. This behavior is now deprecated and will be removed in the future.

The below linker script snippet demonstrates some of the cases where the warning will be emitted and why.

```
cat > 1.c << \!
int data = 10;
!

cat > script.t << \!
SECTIONS {
  .data : {
     /* The warning will be displayed because there is a space between
        '*lib1.a:' and '1.o' */
    *lib1.a: 1.o(.*data*)
     /* No warning. When member-pattern is missing, all members of the
        matched archives are matched. */
    /tmp/lib1.a*: (.*data*)
    *lib1.a: (.*data*)
    /* The warning will be displayed because there is a space between
       '*lib1.a:' and '*' */
    *lib1.a: *(.*data*)
    /* No warning. '*lib1.a:' and '*(.data*)' are considered as separate
       input section descriptions. */
    *lib1.a:
    *(.data*)
  }
}
!

clang -c   -c 1.c -g
llvm-ar cr lib1.a 1.o
ld.eld --whole-archive lib1.a -T script.t -Map x -Wlinker-script
```

Linker Script PHDRS

When should I use PHDRS command

PHDRS command should be used only when you want to control how segments are created by the linker.

If you don't list a segment in the linker script when using PHDR's, the segment will not be created by the linker.

What this means, is that the user needs to explicit specify a segment in the linker script for any specific needs of the loader or the image at runtime.

My output file is big!

An output file may become huge or big depending on how you have created the linker script file. Most use cases

- Will show that the user uses the linker script directive PHDRS.
- The user has specified a virtual address hardcoded in the linker script

For figuring out the issue, its always useful to look at the Map file.

Look for the section where the Map file shows a bigger offset than expected. Most often you will need to change the linker script to remove the hardcoded virtual address for the output section.

Loadable section not in any load segment

This error is experienced when the user has a loadable section, but the user has not assigned a PT_LOAD segment for it. You might want to look at the list of segments and the segment assignments for each output section.

Segments need to be mentioned contiguously in the linker script

Segments need to be contiguous in the linker script, or the virtual address assigned to the segment need to be correctly assigned based on where the previous segment ended for that particular segment.

This is because file size and memory size of the segment needs to be calculated correctly, and the segment needs to be declared contiguously.

The file size / memory size is calculated by the difference between the first output section in the segmet and the last output section in the segment.

If there are segments declared in between, they will create these kinds of duplicate sections.

Below is an example that has duplicate sections in the segment list.

```
cat > 1.c << \!
int bss[10000] = { 0 };
int data = 100;
int foo() { return 0; }
!cat > script.t << \!
PHDRS {
  A PT_LOAD;
  B PT_LOAD;
}
SECTIONS {
  .text : { *(.text*) } :A
  .bss : { *(.bss*) } :B
  .data : { *(.data*) } :A
}
!
clang -target hexagon -c 1.c -ffunction-sections -fdata-sections -G0
hexagon-link 1.o -T script.t
```

readelf output:

```
$ readelf -l -W a.outElf file type is EXEC (Executable file)
Entry point 0x0
There are 2 program headers, starting at offset 52Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x001000 0x00000000 0x00000000 0x09c54 0x09c54 RWE 0x1000
  LOAD           0x00b010 0x00000010 0x00000010 0x00000 0x09c40 RW  0x1000 Section to Segment mapping:
  Segment Sections...
   00     .text .bss .data
   01     .bss
```

Getting file header and program header to be loaded

Often times, it may be required by the runtime loader to inspect the file header and the program header.

You can use FILEHDRS and PHDRS as part of using PHDRS command.

```
cat > 1.c << \!
int data = 20;
int foo() { return 0; }
```

```
!

cat > script.t << \!
PHDRS {
  text PT_LOAD FILEHDR PHDRS;
  data PT_LOAD;
}

SECTIONS {
  . = SIZEOF_HEADERS;
  .foo : { *(.text.foo) } :text
  .data : { *(.data.data) } :data
}
!

$CLANG -c 1.c -ffunction-sections -fdata-sections -G0
$LD 1.o -T script.t
```

To verify that the linker did the right thing, you can use readelf output to figure out.

readelf output

```
Elf file type is EXEC (Executable file)
Entry point 0x0
There are 2 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x000000 0x00000000 0x00000000 0x0008c 0x0008c R E 0x1000
  LOAD           0x00008c 0x0000008c 0x0000008c 0x00004 0x00004 RW  0x1000

 Section to Segment mapping:
  Segment Sections...
   00     .foo
   01     .data
```

If you see the above, the first load segment starts from offset 0, which means the file header gets loaded when the executable runs.

Often times this may be required for building shared libraries, as the dynamic loader needs the file header and program header for relocating it.

Using AT along with loading file headers and program headers

Users need to be careful when including file headers and program headers to be loaded using PHDRS linker script command, and also set a physical address for the section.

Since the file headers and program headers are loaded, linker needs to account for the physical address when creating the segments.

Below is an example that shows an image built with a linker script and uses AT linker script directive.

PHDRS with AT Command

```
cat > 1.c << \!
int foo() { return 0; }
!

cat > script.t << \!
PHDRS {
  A PT_LOAD FILEHDR PHDRS;
}

SECTIONS {
```

```
  . = 0xd820000;
  . = . + SIZEOF_HEADERS;
  .text : AT(0xd820000) { *(.text*) } :A
}
!

clang -c -target aarch64 -c 1.c  -ffunction-sections -fno-asynchronous-unwind-tables
ld.eld -march aarch64 1.o -T script.t
```

Here the linker script shows that the text section has a physical address of 0xd820000, but the user also has said that the FILEHDR and PHDRS to be loaded as part of the load segment.

Linker automatically moves the physical address of the segment to satisfy that the text section has a physical address that can be met as per user needs.

To fix the problem and account for the adjustment of physical addresses, the user needs to make sure that the physical address assigned accounts for the SIZEOF_HEADERS increase.

Recommendation : The user also can drop the usage of AT which seems to be unnecessary to simplify the build step.

Linker Script Fix

```
cat > script.t << \!
PHDRS {
  A PT_LOAD FILEHDR PHDRS;
}

SECTIONS {
  . = 0xd820000;
  . = . + SIZEOF_HEADERS;
  .text : AT(0xd820080) { *(.text*) } :A
}
!
```

Using PHDR flags

Developers can use PHDR flags to convey information from the elf image built to the loader.

The ELF specification provides a convenient way to record this information using phdr flags.

```
#define PF_MASKOS       0x0ff00000      /* OS-specific */
#define PF_MASKPROC     0xf0000000      /* Processor-specific */
```

All bits included in the *PF_MASKOS* mask are reserved for operating system-specific semantics.

All bits included in the PF_MASKPROC mask are reserved for processor-specific semantics. If meanings are specified, the processor supplement explains them.

An example of setting this flag is as below :-

```
PHDRS {
  A PT_LOAD FLAGS (0x03000000);
}

SECTIONS {
.text : { *(.text.*) } :A
}
```

Using the above linker script, you have a segment “A” that contains loadable text but with a OS specific property recorded in the program header.

Loading only the program header

In the above example, we illustrated how an user can load the file header and the program header.

Sometimes it may be required that the user does not want file header but only program header to be loaded.

Below example illustrates the behavior:

Example:

```
cat > 1.c << \!
int data = 20;
int foo() { return 0; }
!

cat > script.t << \!
PHDRS {
  text PT_LOAD PHDRS;
  data PT_LOAD;
}

SECTIONS {
  . = SIZEOF_HEADERS;
  .foo : { *(.text.foo) } :text
  .data : { *(.data.data) } :data
}
!

$CC -c 1.c -ffunction-sections -fdata-sections -G0
$LD 1.o -T script.t
```

You can inspect the resulting executable to see if program headers have been loaded.

readelf output

```
$ readelf -l -W a.out

Elf file type is EXEC (Executable file)
Entry point 0x0
There are 2 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x000034 0x00000034 0x00000034 0x00058 0x00058 R E 0x1000
  LOAD           0x00008c 0x0000008c 0x0000008c 0x00004 0x00004 RW  0x1000

 Section to Segment mapping:
  Segment Sections...
   00     .foo
   01     .data
```

If you see the above, the first load segment starts from offset 34, which means the program header gets loaded when the executable runs.

MEMORY command: common issues and fixes

Why did an orphan section end up in a different region than GNU ld/LLD?

If your script uses `MEMORY` and a section is not explicitly matched by any rule in `SECTIONS`, it becomes an *orphan output section*. In ELD, allocatable orphans (and other allocatable output sections without an explicit `>REGION`) are auto-assigned based on region attributes (for example, `(rx)` vs `(!rwx)`), scanning regions in `MEMORY` order.

Some other linkers may keep an orphan section in the “current” region/cursor instead of re-running attribute matching, which can produce different results.

Fix:

- Add an explicit output section rule and a `>REGION` assignment for the section (for example, `.rodata` or `.eh_frame`), or discard it explicitly if you do not want it in the image.

See also: https://github.com/qualcomm/eld/issues/127

Why do I get `Error: No memory region assigned to section <name>`?

This occurs when `MEMORY` is present and an allocatable output section is not explicitly assigned to a region and does not match any region's attribute constraints.

Fix checklist:

- Assign the section explicitly (`.data : { *(.data*) } >RAM`).
- Ensure you have a region whose attributes match the section type (for example, a writable region for `.data`/`.bss`).
- If you are using `PHDRS`, remember that segment assignment (`:phdr`) does not imply a memory region assignment (`>REGION` / `AT>REGION`).

Why does a MEMORY region overflow even though LENGTH looks sufficient?

Region usage is based on placed output section sizes and their addresses, which can include padding due to:

- section alignment,
- page/segment alignment (especially with separate-code / permissions changes),
- header space when `FILEHDR PHDRS` is used,
- gaps introduced by explicit addresses in the script.

Fix:

- Inspect the map file (`-MapStyle txt -Map out.map`) to see per-section addresses and which regions they were charged against.
- Use `–print-memory-usage` to see per-region usage totals.
- Increase the region `LENGTH` or reduce alignment/padding requirements.

String Merging

Merging order

On 8.7, the linker deduplicates strings according to link order. On >= 8.8 duplicate strings are decided by script rule order. Example below:

```
cat > 1.s << \!
.section .rodata1.str1.1,"aMS",@progbits,1
.string "abc"
!

cat > 2.s << \!
.section .rodata1.str1.2,"aMS",@progbits,1
.string "abc"
!

cat > script.t << \!
SECTIONS {
  .rodata : {
      2.o(.rodata*)
      1.o(.rodata*)
   }
}
!
```

```
# link order 1.o, 2.o
# rule order 2.o, 1.o
ld.eld 1.o 2.o
```

On 8.7, the string “abc” from 1.o will be included and the string from 2.o will be merged with it (according to link order). On >=8.8, the string from 2.o will be included and string from 1.o merged with it (according to rule order). This difference is reflected in the map file.

Hexagon

Convert binary file to Hexagon

If you want to convert a binary file file1.bin to hexagon, you can use objcopy.

```
hexagon-llvm-objcopy -I binary file1.bin -O elf32-hexagon test.o
```

Hexagon specific behavior

PHDRS

Hexagon static link and dynamic linked executables dont rely on PHDR’s to be available for the dynamic loader.

If there is a need for you to load PHDR’s, you need to look at Gettingfileheaderandprogramheadertobeloaded

> **Warning**
>
> **PHDRS not covered by load segment**
>
> Often times, if you are using readelf on a dynamic executable for hexagon, you may get an error
>
> readelf: Error: the PHDR segment is not covered by a LOAD segment
>
> This is because of the above assumption that Hexagon dynamic executables dont rely on PHDR’s to be loaded and available to the dynamic loader.
>
> Use hexagon-llvm-readelf to overcome this error.

Why *COMMON* input section description does not affect all the common symbols?

For the hexagon target, by default, the linker maps small common symbols to internal sections, .scommon.x, where x can be 1, 2, 4 and 8. The x represents the size of the common symbol in bytes. For example, .scommon.4 will match common symbols having size 4 bytes. Therefore, to write rules for small common symbols, .scommon.x input section description should be used. This allows writing rules affecting common symbols with greater precision

For Example, to match:

- all common symbols of size less than or equal to 2 bytes to the output section *.small_commons_2*
- all common symbols of size greater than 2 bytes but less than or equal to 8 bytes to the output section *.small_commons_8.*

The following linker script SECTIONS command can be used:

```
SECTIONS {
    .text : {*(.text*) }
    .small_commons_2: { *(.scommon.1 .scommon.2) }
    .small_commons_8: { *(.scommon.4 .scommon.8) }
    .bss: { *(COMMON) }
}
```

*(COMMON) input section description will match all common symbols of size greater than 8 bytes.

To verify the linker script

Test C code

```
char a;
short b;
int c;
long long d;
double e[100];
```

Compile and Link Step:

```
clang -target hexagon -c test.c -o test.o
hexagon-link test.o -o test.elf -T test.linker.script
```

Verify:

```
hexagon-readelf -Ss common.elf
```

readelf output:

```
There are 8 section headers, starting at offset 0x11c0:

Section Headers:
  [Nr] Name              Type            Address  Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .small_commons_2  NOBITS          00000000 001000 000004 00  WA  0   0  2
  [ 2] .small_commons_8  NOBITS          00000008 001000 000010 00  WA  0   0  8
  [ 3] .bss              NOBITS          00000018 001000 000320 00  WA  0   0  8
  [ 4] .comment          PROGBITS        00000000 001000 000065 01  MS  0   0  1
  [ 5] .shstrtab         STRTAB          00000000 001065 00004b 00      0   0  1
  [ 6] .symtab           SYMTAB          00000000 0010b0 0000c0 10      7   6  4
  [ 7] .strtab           STRTAB          00000000 001170 00004a 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  R (retain), p (processor specific)

Symbol table '.symtab' contains 12 entries:
   Num:    Value  Size Type    Bind   Vis       Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT   UND
     1: 00000000     0 SECTION LOCAL  DEFAULT     1 .small_commons_2
     2: 00000000     0 SECTION LOCAL  DEFAULT     4 .comment
     3: 00000008     0 SECTION LOCAL  DEFAULT     2 .small_commons_8
     4: 00000018     0 SECTION LOCAL  DEFAULT     3 .bss
     5: 00000000     0 FILE    LOCAL  DEFAULT   ABS common.c
     6: 00000000     1 OBJECT  GLOBAL DEFAULT     1 a
     7: 00000002     2 OBJECT  GLOBAL DEFAULT     1 b
     8: 00000008     4 OBJECT  GLOBAL DEFAULT     2 c
     9: 00000010     8 OBJECT  GLOBAL DEFAULT     2 d
    10: 00000018   800 OBJECT  GLOBAL DEFAULT     3 e
    11: 00000339     0 NOTYPE  GLOBAL DEFAULT   ABS __end
```

readelf output shows what we expected:

- symbols a, and b are matched to the output section *.small_commons_2*
- symbols c, and d are matched to the output section *.small_commons_8*
- symbol e is matched to the output section *.bss.*

> **Note**
>
> The linker option -G<size> can be used to specify the maximum size for the small common symbols. For example, -G4 option specifies the maximum size for the small common symbols to be 4 bytes. By default, the linker assumes the maximum size of the small common symbols to be 8 bytes.

How to disable small common symbol functionality?

Small common symbol functionality can effectively be disabled by using -G0 linker command-line option. -G<size> option specifies the maximum size for the small common symbols.

-G0 will have the following effects:

- Small common symbols will map to COMMON input section description instead of .scommon.x input section descriptions in the linker script
- If not specified otherwise, small common symbols will be stored in .bss output section instead of .sdata output section.

RISC-V

Show Linker relaxation output

The linker shows relaxation output by using the option –verbose. In future there will be a better option to annotate where linker performed relaxation.

Disable Relaxation

You can disable relaxation in the linker using the option, –no-relax. This option disables all of linker relaxation except handling of alignment.

Disable compressed relaxation

You can disable linker relaxing to compressed instructions by using –no-c-relax flag.

Usage

Response Files

Response files let you pass linker arguments via a file referenced with @.

Example:

```
ld.eld @x.cmd -o <elf>
```

Notes:

- The response file contains command-line arguments. Any valid linker argument can be passed via the file and is expanded in place of `@x.cmd`.
- Response files help avoid host command-line length limits for very long argument lists.
- A `@response-file` can appear inside another response file (recursive expansion).
- Unrecognized arguments in the file are ignored with a warning (for example `Warning: Unrecognized option '–bad'`).
- Missing response files or invalid arguments cause the link to fail.

Reference: https://llvm.org/docs/CommandLine.html#response-files

Linker Plugin Framework

What are the differences between SectionMatcherPlugin and SectionIteratorPlugin?

- SectionMatcherPlugin is run before rule-matching and garbage-collection whereas SectionIteratorPlugin is run after rule-matching and garbage-collection.As a consequence of this, SectionIteratorPlugin callbacks have access to garbage-collected sections, discarded sections and rule matching information among other things.
- SectionIteratorPlugin does not call 'SectionIteratorPlugin::Process' callback hook for garbage-collected sections. It is though called for discarded sections. On the other hand, SectionMatcherPlugin call 'SectionMatcherPlugin::Process' callback hook for each input section.

Symbol Resolution

Symbol wrapping

What is symbol wrapping and how to use it?

Symbol wrapping allows to use a wrapper symbol in-place of the original symbol, without any modification to the source code. When symbol wrapping is used for the symbol 'symbol', then all undefined references to 'symbol' is resolved to '__wrap_symbol', and all undefined references to '__real_symbol' is resolved to 'symbol'. To enable symbol wrapping, use '–wrap=symbol' option.

Where is symbol wrapping helpful?

Symbol wrapping is typically used for wrapping system/standard function calls. For example, a 'malloc' wrapper function can be used to keep track of bytes allocated by 'malloc', and a 'printf' wrapper function can be used to add a timestamp in the printf output.

Example

The below program provides a wrapper function for 'my_malloc' function. The wrapper function prints the number of bytes that is requested by user using 'my_malloc', and then calls 'my_malloc' to do the actual memory allocation.

```
#!/usr/bin/env bash

cat >1.c <<\EOF
#include <stdio.h>

void *my_malloc(size_t sz);

int main() {
  int *p = my_malloc(sizeof(int)); // Resolves to __wrap_my_malloc
  *p = 11;
  printf("p: %d\n", *p);
  return *p;
}
EOF

cat >2.c <<\EOF
#include <stdlib.h>
#include <stdio.h>

void *my_malloc(size_t sz) { return malloc(sz); }

void *__real_my_malloc(size_t sz);
void *__wrap_my_malloc(size_t sz) {
  printf("'%u' bytes requested using my_malloc\n", sz);
  return __real_my_malloc(sz); // Resolves to 'my_malloc'
```

```
}
EOF

clang -target hexagon -o 1.wrapped.elf 1.c 2.c 3.c -ffunction-sections -g -Wl,--wrap=my_malloc
hexagon-sim 1.wrapped.elf
# Output:
# '4' bytes requested using my_malloc
# p: 11
```

Build time issues

Common to all targets

LTO is showing undefined symbol when symbol is defined

The questions to answer when there is an issue like this is to, see if the symbol is in an archive library or an object file.

If its an archive library, make sure that the library is built using **llvm-ar**

The default archiver which handles ELF files does not handle **Bitcode** which is the format that LLVM uses.

This is a must for LTO use.

If you have a reproducer using the reproduce option, you can use this snippet to replace all your archive files using llvm-ar

```
for file in `grep "\.lib" mapping.ini`; do file=`echo $file | sed 's/.*=//'`; echo $file; mkdir -p x; cd x; cp ../$file .; ar x $file; rm -f $file; llvm-ar cr $file *.o; cp $file ..; cd ..; done
```

LTO is showing could not set section name for the symbol

This note is emitted while reading the bitcode files (only part of LTO flow, as in non-LTO flow object files are read) for symbols defined using the ‘.set’ directive.

> **Note**
>
> extern void someSymbol_relocated (void) __attribute__((__visibility__(“hidden”)));
>
> __asm(“.set someSymbol_relocated, 0x00005730”);

This note means that no input section was found for this symbol in the input bitcode file as these symbols are found to be undefined, due to assembler directive usage

The LTO flow, though, happens at link time but doesn’t have visibility to mapping symbol address set with .set directive to symbols.

Hexagon

Relocation overflows to function symbols

A relocation overflow can show up when the linker cannot insert a trampoline to reach that symbol.

Most often the problems would be that the symbol may be *Undefined Weak.*

The programmer would have specified a *__attribute__((weak))* while declaring the function in the header file.

Also look at Weakreferencesanddefinitions why the linker was not seeing the definition.

> **Note**
>
> A common work around for this sort of problem is to remove the __attribute__((weak)) from the prototype of the function.

Debugging Linker crash issues

Step 1

Look for the error message produced by the linker, if it says Unexpected Linker behavior continue to the next step.

If it shows the problem was in the user plugin, debug the plugin and figure out what the issue is.

Step 2

Check to see if there any system environment variables set. Important variables to note are :-

- LD_LIBRARY_PATH
- PATH (Windows)

Remove the values set and see if the error disappears

Step 3

If the linker is still crashing, check to see if the cause is due to multithreading plugins.

You can then use –no-threads and see if the error disappears

Step 4

Sometimes the plugin may write some data into linker memory and corrupt some data structures.

Build the plugin with sanitizer enabled, and see if you can spot any sanitizer issues in the plugin.

Step 5

If it still is crashing then contact the support teams for help debugging the issue.

ARM

SBREL relocations : Fixing relocation error when applying relocation ‘R_ARM_SBREL32’

SBREL relocation stands for segment base relative relocations. There needs to be only one segment defined that is segment base relative in the image.

All variables and access to the variables need to be placed in that segment.

Example and how to fix :-

```
cat > 1.c << \!
int foo;
int bar = 20;
int main() { return foo + bar; }
!

cat > script.t << \!
PHDRS {
  TEXT PT_LOAD;
  SBREL_SEGMENT PT_LOAD;
  SOMETHING_ELSE PT_LOAD;
}
```

```
SECTIONS {
  .text : { *(.text) } :TEXT
  .ARM.exidx : { *(.ARM.exidx*) }
  .data : { *(.data.bar) } :SBREL_SEGMENT
  .bss : { *(COMMON) } :SOMETHING_ELSE
}
!

clang -c -frwpi 1.c -target arm -fdata-sections -g3
ld.eld -march arm 1.o -T script.t -Map x
```

Example error output:

```
Error: R_ARM_SBREL32 Relocation Mismatch for symbol bar defined in 1.o[.text] has a different load segment
Error: Relocation error when applying relocation `R_ARM_SBREL32' for symbol `bar' referred from 1.o[.text] symbol defined in 1.o[.data.bar]
Error: R_ARM_SBREL32 Relocation Mismatch for symbol bar defined in 1.o[.debug_info] has a different load segment
Error: Relocation error when applying relocation `R_ARM_SBREL32' for symbol `bar' referred from 1.o[.debug_info] symbol defined in 1.o[.data.bar]
Fatal: Linking had errors.
```

Fixed linker script for this example

```
PHDRS {
  TEXT PT_LOAD;
  SBREL_SEGMENT PT_LOAD;
  SOMETHING_ELSE PT_LOAD;
}

SECTIONS {
  .text : { *(.text) } :TEXT
  .ARM.exidx : { *(.ARM.exidx*) }
  .data : { *(.data.bar) } :SBREL_SEGMENT
  .bss : { *(COMMON) } :SBREL_SEGMENT
}
```

How to resolve the warning ‘Compact Option needs physical address aligned with File offsets’

Before we see how to resolve this warning, let’s first discuss what the `–compact` option is and the reason for this warning.

The `–compact` option, as the name suggests, allows generating more compact (smaller) images. However, this benefit comes with the added restriction that for each `LOAD` segment, physical addresses must be aligned with the file offsets. This means the the difference between physical address and file offset should should be the same for each byte within a `LOAD` segment.

This implies that the physical addresses must also be aligned with virtual memory addresses within a `LOAD` segment. This is because the linker ensures that virtual memory addresses are always aligned with file offsets.

It is user’s responsibility to ensure that this restriction is satisfied. The linker reports the warning *Compact Option needs physical address aligned with File offsets* if the restriction is not satified. This warning should not be ignored because images not satisfying this restriction may have runtime errors.

Let’s analyze this issue with the help of an example:

```
#!/usr/bin/env bash

cat >1.c <<\EOF
int foo = 1;

__attribute__((aligned(8)))
int bar = 3;

int baz = 5;
EOF
```

```
cat >script.t <<\EOF
PHDRS {
  TEXT PT_LOAD;
}

SECTIONS {
  FOO 0x1000 : AT(0x4) { *(*foo*) } :TEXT
  BAR : { *(*bar*) } :TEXT
  BAZ : { *(*baz*) } :TEXT
}
EOF

clang -o 1.o 1.c -c -ffunction-sections -fdata-sections
arm-link -o 1.out 1.o -T script.t -Map 1.map.txt --compact
```

We get the following warnings on running this example:

```
Warning: Physical address and the offset of a segment must be congruent modulo the alignment of the segment. Mismatch found at segment FOO
Warning: Compact Option needs physical address aligned with File offsets. Mismatch found at section BAR
Warning: Compact Option needs physical address aligned with File offsets. Mismatch found at section BAZ
```

Why do we get this warning here and how do we fix it? Let's see.

For brevity, I will use VMA for virtual memory address and LMA (load memory address) for physical address.

FOO size is 0x4, VMA is 0x1000 and LMA is 0x4. The difference between the VMA and the LMA is 0xffc. We may expect BAR VMA to be 0x1004 (0x1000 + 0x4) and the LMA to be 0x8 (0x4 + 0x4). However, the linker has to bump this VMA to 0x1008 to satisfy BAR 8-byte alignment requirement. No bump is required for the LMA as it is already 8-byte aligned. This bump to VMA causes misalignment between VMA and LMA, as seen from VMA - LMA = 0x1000, whereas previously the difference was 0xffc. Hence we also have misalignment between file offsets and LMA (VMA and file offsets are *always* aligned) and the `–compact` option alignment restriction is violated!

Note that the linker cannot simply bump the LMA by 0x4 as well to align with the VMA bump because then the LMA (0xc) will not be 8-byte aligned.

To fix this issue, we have to ensure that for each LOAD segment, **the VMA is congruent to the LMA, modulo the maximum alignment of any section within the segment**.

For this particular case, this means that TEXT segment VMA and LMA must be congruent modulo 8. 8 is the alignment of BAR section and is the maximum alignment requirement of sections within TEXT. This congruency ensures that no bump for satisfying alignment will cause misalignment between VMA and LMA. Hence, to fix this issue we can change VMA to 0x1004 (0x1004 and 0x4 are congruent to each other, modulo 8), or change LMA to 0x8 (0x1000 and 0x8 are congruent to each other, modulo 8). Any other value of VMA and LMA satifying this congruency is, of course, valid as well.

Runtime issues

ARM

Crash with load / stores

Things to look at when there is a crash :-

- Look at the point of crash, and the memory from where the instruction is loading from or storing to
- If the load or store is happening with a variable stored in the stack, bounce the problem to the compiler.
- Otherwise, check:
  - Where the variable is located (from the linker map file).
  - Alignment restrictions.
  - TLB permissions.
  - Whether the linker script overrides default alignment for the output section that contains the variable.

- Page alignment. Some operating systems may require a higher page alignment; you can change the default page size using `-z max-page-size`.

Improving your image/build

The below section documents how users can improve the image / build by looking into build time warning messages emitted by the linker.

Warnings

Convert warning to error

Users can add `-Werror` to convert all warnings that the linker emits into non-fatal errors. A stronger form is `–fatal-warnings`, which promotes warnings to fatal errors.

This is synonymous to using `-Wall -Werror` with the compiler

Non allocatable section assigned to an output section

Users might want to avoid assigning a non allocatable section to an output section which is allocatable.

Though this is not a serious issue to be looked into, some of the issues which this will result in are :-

- It may be that tools may be not be able to decode or disassemble contents from the object files properly when there is a bug.
- **The build also may become flaky**
    - When section which converts the output section to be allocatable garbage collected, which will make the section that user added as a non allocatable section.
- Non allocatable sections are not loaded by the linker
- Relocations in non allocatable sections are not performed in the way allocatable sections are treated.

Linker detects that a non allocatable section is being assigned to an output section which is allocatable.

```
cat > code.s << \!
.section .foo
sym:
nop
.section .text
call sym
!

cat > script.t << \!
SECTIONS {
  .text : {
    *(.text)
    *(.foo)
  }
}
!

$clang -c code.s
$link code.o -T script.t -o code.out
```

Linker would issue the below warning in this case.

> **Warning**
>
> Warning: Non-allocatable section ‘.foo’ from input file ‘code.o’ is being merged into loadable output section ‘.text’

These warnings can be converted to errors with –fatal-warnings switch.

```
$link code.o -T script.t --fatal-warnings
```

Error

Fatal: Non-allocatable section '.foo' from input file 'code.o' is being merged into loadable output section '.text'

LinkerPlugin API

Build Errors

Why am I receiving the error 'functions that differ only in their return type cannot be overloaded' error while building plugins?

Fix: Build the plugin with '-std=c++14' (or greater C++ standard).

But what is the cause of the error?

The plugin framework overloads member functions based on the reference-qualifier of 'this' argument to provide optimal performance. However, C++11 does not support specifying reference-qualifier for the 'this' argument of member functions. This means that when C++11 is used, C++ cannot distinguish between member function prototypes that only differ due to the reference-qualifier of the 'this' argument.

Why am I getting an error about "Plugin Error referenced chunk <seen from last rule> deleted from Output section"

This is a very common error, when a plugin takes chunks from linker script rules, and sometimes the chunk are not put back by the linker plugins.

These chunks may be referred from other locations in the image, and is needed to satisfy image layout requirements.

When such an error happens

- **Please debug the plugin by inspecting calls to**
    - addChunk
    - removeChunk
    - updateChunks

You may also rely on linker map file to see that chunks are added and removed properly by inspecting plugin behavior.

Runtime errors

Why am I receiving the error 'Unable to load library ${libYourPlugin.so}: undefined symbol: …'?

Fix: Build the plugin with '-stdlib=libc++'.

But what is the cause of this runtime error?

Plugin framework is compiled with 'libc++' implementation of standard library. Therefore, the mangled names of standard library components in the plugin framework are as per 'libc++' implementation. By default, Clang uses 'libstdc++' implementation of standard library, which has different mangled names of standard library components as compared to 'libc++' implementation. This difference of C++ standard library implementation causes the conflict in the symbol names which in turn causes undefined symbol error.

Concepts

What are the differences between SectionMatcherPlugin and SectionIteratorPlugin?

- *SectionMatcherPlugin* interface is run before garbage-collection whereas SectionIteratorPlugin interface is run after rule-matching and garbage-collection. As a consequence of this, SectionIteratorPlugin callbacks have access to the information which sections are garbage-collected and discarded.
- *SectionIteratorPlugin* interface does not call 'SectionIteratorPlugin::Process' callback hook for garbage-collected sections. It is though called for discarded sections. On the other hand, SectionMatcherPlugin call 'SectionMatcherPlugin::Process' callback hook for each input section.

What is the search order for plugin invocation configuration file?

Plugin invocation configuration file is the YAML file that is specified to the linker using *–plugin-config <file>* option.

The linker searches this file as follows:

- If the *<file>* is an absolute path, then no search is performed and the path is taken as it is.
- **If the *<file>* is a relative path, then the search is performed as follows:**
    - In the current working directory. i.e. the directory from which linker is invoked.
    - In the search directories specified by the -L option, in the order they are listed in the link command.

What is the search order for LinkerWrapper::findConfigFile function?

LinkerWrapper::findConfigFile searches the file as follows:

- If the *<file>* is an absolute path, then no search is performed and the path is taken as it is.
- **If the *<file>* is a relative path, then the search is performed as follows:**
    - In the current working directory. i.e. the directory from which linker is invoked
    - In the search directories specified by the -L option, in the order they are listed in the link command.
    - Plugin default configuration file directory that is shipped with the toolchain. This directory is different for each plugin that is shipped with the toolchain. This directory is searched so that it is convenient to use the default plugin configuration files.

Debugging

How to see which plugins are successfully loaded?

Plugin workflow can be traced using the option *'–trace=plugin'.* This option tells the linker to print logs describing the plugin workflow, including information about plugin loading.

Is it expected to see decrease in link-time performance when plugins are used?

Yes, user can is expected to see a decrease in link-time performance when

- **Plugins are built with a wrong compiler optimization options.**
    - Using -g in release builds
- Plugins are developed without using Multi-threading features available.

If you are developing plugins and plan to deploy it, use –print-timing-stats to figure out which plugin / what part of the plugin is taking more time during the link step. The plugin framework is feature rich and does not cause link time degradation in any way.

How to verify which plugin config file was selected by the linker?

Run the link command with –verbose option, and search the build logs for 'Trying to open. *<plugin config file name>' and '<plugin config file name>.*: found'. These two searches will tell you which directories were searched by the linker (and in which order), and where the plugin configuration file was finally found.

Plugin compatibility

Plugin API version

Starting with version 19, linker verifies that the loaded plugin is build using a compatible API version. Linker will refuse to load a plugin which API version is incompatible or the plugin does not report its API version.

Plugin API version consists of major and minor version numbers. For example, the current Plugin API version is 3.2, which can be inspected with the –version linker option:

```
$ ld.eld --version
Supported Targets: hexagon
Linker from QuIC LLVM Hexagon Clang version Version
Linker based on LLVM version: 19.0
Linker Plugin Support Enabled
Linker Plugin Interface Version 3.2
LTO Support Enabled
```

A plugin is compatible with given Plugin API version if their major version numbers are equal and plugin's minor API version is the same or lower than the current minor API version. This means that a plugin compiled for a certain linker API is compatible with future linkers as long as the major version number stays the same.

Plugins must be recompiled to be used with linker with a different major API version.

Reporting API version by plugins

Each plugin must report its API version by exporting the following function from the plugin shared library:

```
extern "C" void getPluginAPIVersion(unsigned int *Major, unsigned int *Minor);
```

Plugin API library provides a helper implementation of this function, which as defined in the PluginVersion.h file. To report the plugin API, a plugin developer can include this header file into one of the C++ source files:

```
#include <PluginVersion.h>
```

Linker Plugin Config Search Paths

What are the search paths for linker plugin config files?

The *LinkerWrapper::findConfigFile()* API searches for a config file in the following directories in the following order:

- Try to find the file path directly
- Search directories passed to the linker via -L. Some directories are implicitly included, such as the current working directory, and /lib and /usr/lib on linux.
- Default path for config files: *${ORIGIN}/../etc/ELD/Plugins/<Plugin name>*

Note

${ORIGIN} is the directory in which the linker binary resides

How to verify that the plugin search config was found and debug related issues?

- The –verbose option will print extra logs to help determine if the plugin config file was found

- **To debug issues related to search path for linker plugin file analyze the traces obtained via the below 2 steps:**
  - To get the list of paths at which the plugin config file path was searched for grep for “Trying to open.*<plugin config file name>” on verbose logs.
  - To see if the linker plugin config file was found grep for “<plugin config file name>.*: found” on verbose logs.

Example:

An example of using the API and verbose option to debug search paths is shown below for linux and for windows:

```
eld::Expected<std::string> expConfigPath =
  getLinker()->findConfigFile("foo.ini");
```

Link command (linux):

```
ld.eld -L/tmp/ ...
```

```
// Search for the file directly
Verbose: Trying to open `foo.ini' for plugin configuration INI file `foo.ini' (file path): not found
// look in -L directories
Verbose: Trying to open `/tmp/foo.ini' for plugin configuration INI file `foo.ini' (search path): not found
// implicit directories (cwd, /lib, /usr/lib)
Verbose: Trying to open `/usr2/aregmi/foo.ini' for plugin configuration INI file `foo.ini'
Verbose: Trying to open `/lib/foo.ini' for plugin configuration INI file `foo.ini' (search path): not found
Verbose: Trying to open `/usr/lib/foo.ini' for plugin configuration INI file `foo.ini' (search path): not found
// try the default config directory
Note: Using default config path for linker plugins
Verbose: Trying to open `/local/mnt/workspace/aregmi/llvm-project/llvm/build/bin/../etc/ELD/Plugins/ConfigFile/foo.ini' for plugin configuration INI file `foo.ini' (search path): found
Verbose: Found plugin config file /local/mnt/workspace/aregmi/llvm-project/llvm/build/etc/ELD/Plugins/ConfigFile/foo.ini
```

Link command (Windows):

```
ld.eld -L/ C:\Users\ ...
```

Verbose logs:

```
// Search for the file directly
Verbose: Trying to open `foo.ini' for plugin configuration INI file `foo.ini' (file path): not found
// look in -L directories
Verbose: Trying to open `C:\Users\foo.ini' for plugin configuration INI file `foo.ini' (search path): not found
// implicit directories (cwd)
Verbose: Trying to open `C:\Users\aregmi\tmp\foo.ini' for plugin configuration INI file `foo.ini' (search path): not found
Note: Using default config path for linker plugins
// default config directory
Verbose: Trying to open `C:\Users\aregmi\install\Tools\bin/../etc/ELD/Plugins/ConfigFile\foo.ini' for plugin configuration INI file
`foo.ini' (search path): found
Verbose: Found plugin config file C:\Users\aregmi\install\Tools\etc\ELD\Plugins\ConfigFile\config\foo.ini
```

Linker Script NOLOAD handling

Description

NOLOAD sections are used as a way to tell the loader to reserve one or more regions in memory and not refresh its contents for repeated loads of the same image.

Assumptions of the regions are

- The region has a reserved virtual address space
- The region has a reserved physical address space
- The region gets translated to “BSS”

This means that the image when it runs, can write some content to the region and re-read their contents.

Examples of use cases are:

- Store state of the application before the application crashes, resume from where it left off
- Store state of why the application crashed for inspection

References

https://mcuoneclipse.com/2014/04/19/gnu-linker-can-you-not-initialize-my-variable/

Usecases

NOLOAD region assigned to PT_NULL segment

Code:

```
cat > fb.c << \!
int foo() { return 0; }
int bar() { return 0; }
int baz() { return 0; }
!

cat > noload.lcs << \!
PHDRS {
  A PT_LOAD;
  B PT_NULL;
}

SECTIONS {
  .foo : { *(.text.foo) } :A
  .bar (NOLOAD) : { *(.text.bar) } :B
  .baz (NOLOAD) : { *(.text.baz) } :B
}
!
```

The above test places bar and baz in NOLOAD regions and places them in a segment that is non loadable (PT_NULL).

Readelf Output:

```
$ llvm-readelf -l -W a.out

Elf file type is EXEC (Executable file)
Entry point 0x0
There are 2 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x001000 0x00000000 0x00000000 0x0000c 0x0000c R E 0x1000
  NULL           0x000000 0x00000010 0x00000010 0x00000 0x00320 RW  0x1000

 Section to Segment mapping:
  Segment Sections...
   00     .foo
   01     .bar .baz
   None   .comment .shstrtab .symtab .strtab
```

Warning

PT_NULL segments in this case will have the virtual address set and the segment memory size appropriately set. If you do not want to see virtual addresses assigned or memory size set, remove the segment assignment(s).

Placing a loadable section to PT_NULL

Code:

```
cat > fb.c << \!
int foo() { return 0; }
int bar[100] = { 0 };
int baz[100] = { 0 };
```

```
!

cat > noload.lcs << \!
PHDRS {
  A PT_LOAD;
  B PT_NULL;
}

SECTIONS {
  .foo : { *(.text.foo) } :A
  .bar (NOLOAD) : { *(.bss.bar) } :B
  .baz  : { *(.bss.baz) } :B
}
!
```

The test places baz which is loadable into a PT_NULL segment

Output:

Error

Error: Loadable section .baz not in any load segment Fatal: Linking had errors.

Placing NOLOAD region to LOAD segment

Code:

```
cat > fb.c << \!
int foo() { return 0; }
int bar[100] = { 0 };
int baz[100] = { 0 };
!

cat > noload.lcs << \!
PHDRS {
  A PT_LOAD;
  B PT_LOAD;
}

SECTIONS {
  .foo : { *(.text.foo) } :A
  .bar (NOLOAD) : { *(.bss.bar) } :B
  .baz (NOLOAD) : { *(.bss.baz) } :B
}
!
```

This tests produces valid results with a section that is set to NOLOAD and placed in a PT_LOAD segment.

Output:

```
$ llvm-readelf -l -W a.out

Elf file type is EXEC (Executable file)
Entry point 0x0
There are 2 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x001000 0x00000000 0x00000000 0x0000c 0x0000c R E 0x1000
  LOAD           0x001010 0x00000010 0x00000010 0x00000 0x00320 RW  0x1000
```

```
 Section to Segment mapping:
  Segment Sections...
   00     .foo
   01     .bar .baz
```

Mixing NOLOAD and LOAD regions

Code:

```
cat > fb.c << \!
__attribute__((aligned(4))) int foo() { return 0; }
__attribute__((aligned(4))) int bar() { return 0; }
__attribute__((aligned(4))) int baz() { return 0; }
__attribute__((aligned(4))) int bad() { return 0; }
!

cat > noload.lcs << \!
PHDRS {
  A PT_LOAD;
  B PT_LOAD;
}

SECTIONS {
  .foo : { *(.text.foo) } :A
  .bar (NOLOAD) : { *(.text.bar) } :B
  .baz (NOLOAD) : { *(.text.baz) } :B
  .bad  : { *(.text.bad) } :B
}
!
```

This places text.bar and text.baz as NOLOAD and .text.bad as a loadable section but all of them in a LOAD segment.

These are the things that happens

- The text section gets turned into a BSS section (without any warning)
- The text.bad section gets assigned a proper address

Output:

```
$ llvm-readelf -l -W a.out

Elf file type is EXEC (Executable file)
Entry point 0x0
There are 2 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x001000 0x00000000 0x00000000 0x0000c 0x0000c R E 0x1000
  LOAD           0x001010 0x00000010 0x00000010 0x0000c 0x0002c R E 0x1000

 Section to Segment mapping:
  Segment Sections...
   00     .foo
   01     .bar .baz .bad
```

NOLOAD section in the middle of a PT_LOAD segment

```
cat > 1.c << \!
int foo() {
  return 0;
}
int mybss[100] = { 0 };

int bar() {
  return 0;
}
!

cat > script.t << \!
PHDRS {
 A PT_LOAD;
}
SECTIONS {
 .foo : {
   *(.text.foo)
 } :A
 .bss (NOLOAD) : {
   *(.data*)
   *(.bss*)
 } :A
 .bar : {
  *(.text.bar)
 } :A
}
!
```

The example places foo and bar in .foo and .bar output sections respectively. The linker script also reserves a .bss region of type NOLOAD, which is followed by a loadable section .bar. All of the sections are located in a loadable segment by name ‘A’.

The layout is adjusted so that, the section .bar is placed after .bss in the file. The offset for .bar is calculated as the offset of bss and the size of bss.

Output:

```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x001000 0x00000000 0x00000000 0x001ac 0x001ac RWE 0x1000

 Section to Segment mapping:
  Segment Sections...
   00     .f .bss .bar
```

Warning

Without the NOLOAD directive, Linker would have produced this error message

Mixing BSS and non-BSS sections in segment. non-BSS ‘.bar’ is after BSS ‘.bss’ in linker script

NOLOAD sections start of the segment and PT_NULL segment

Code:

```
int foo() {
  return 0;
```

```
}
__attribute__((aligned(16384))) int bss1[100] = { 0 };
__attribute__((aligned(16384))) int bss2[100] = { 0 };
__attribute__((aligned(16384))) int bss3[100] = { 0 };
__attribute__((aligned(16384))) int abss1[100] = { 0 };
__attribute__((aligned(16384))) int abss2[100] = { 0 };
__attribute__((aligned(16384))) int abss3[100] = { 0 };

int bar() {
  return 0;
}
```

LinkerScript:

```
PHDRS {
 A PT_NULL;
}
SECTIONS {
 .abss1 (NOLOAD) : {
   *(.bss.abss1)
 } :A
 .abss2 (NOLOAD) : {
   *(.bss.abss2)
 } :A
 .abss3 (NOLOAD) : {
   *(.bss.abss3)
 } :A
 .f (NOLOAD) : {
   *(.text.foo)
 } :A
 .bss1 (NOLOAD) : {
   *(.bss.bss1)
 } :A
 .bss2 (NOLOAD) : {
   *(.bss.bss2)
 } :A
 .bss3 (NOLOAD) : {
   *(.bss.bss3)
 } :A
 .bar (NOLOAD) : {
  *(.text.bar)
 } :A
}
```

Commands:

```
$clang -c 1.c -ffunction-sections -fdata-sections -G0
$ld.eld 1.o -T script.t
```

Output:

```
$ readelf -l -W a.out

Elf file type is EXEC (Executable file)
Entry point 0x0
There is 1 program header, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x000000 0x00000000 0x00000000 0x00000 0x1419c RWE 0x1000

 Section to Segment mapping:
```

```
  Segment Sections...
   00     .abss1 .abss2 .abss3 .f .bss1 .bss2 .bss3 .bar
```

Note

Note: the initial offset assigned to the section. This results in a bug with GNU linker.

NOLOAD sections start of the segment

Code:

```
int foo() {
  return 0;
}
__attribute__((aligned(16384))) int bss1[100] = { 0 };
__attribute__((aligned(16384))) int bss2[100] = { 0 };
__attribute__((aligned(16384))) int bss3[100] = { 0 };
__attribute__((aligned(16384))) int abss1[100] = { 0 };
__attribute__((aligned(16384))) int abss2[100] = { 0 };
__attribute__((aligned(16384))) int abss3[100] = { 0 };

int bar() {
  return 0;
}
```

LinkerScript:

```
PHDRS {
 A PT_LOAD;
}
SECTIONS {
 .abss1 (NOLOAD) : {
   *(.bss.abss1)
 } :A
 .abss2 (NOLOAD) : {
   *(.bss.abss2)
 } :A
 .abss3 (NOLOAD) : {
   *(.bss.abss3)
 } :A
 .f : {
   *(.text.foo)
 } :A
 .bss1 (NOLOAD) : {
   *(.bss.bss1)
 } :A
 .bss2 (NOLOAD) : {
   *(.bss.bss2)
 } :A
 .bss3 (NOLOAD) : {
   *(.bss.bss3)
 } :A
 .bar : {
  *(.text.bar)
 } :A
}
```

Commands:

```
$clang -c 1.c -ffunction-sections -fdata-sections -G0
$ld.eld 1.o -T script.t
```

Output:

```
$ readelf -l -W a.out

Elf file type is EXEC (Executable file)
Entry point 0x0
There is 1 program header, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x004000 0x00000000 0x00000000 0x1419c 0x1419c RWE 0x1000

 Section to Segment mapping:
  Segment Sections...
   00     .abss1 .abss2 .abss3 .f .bss1 .bss2 .bss3 .bar
```

How the offset of the section is calculated

- The offset of the first loadable section starts at an offset = previous offset + (current section address - first section in the segment)
- The offset of the first lodable section starts at an offset = previous offset + (current section address - previous section which occupies file space)

NOLOAD sections start of the segment with first load section starting at a virtual address

Code:

```
  int foo() {
  return 0;
}
__attribute__((aligned(16384))) int bss1[100] = { 0 };
__attribute__((aligned(16384))) int bss2[100] = { 0 };
__attribute__((aligned(16384))) int bss3[100] = { 0 };
__attribute__((aligned(16384))) int abss1[100] = { 0 };
__attribute__((aligned(16384))) int abss2[100] = { 0 };
__attribute__((aligned(16384))) int abss3[100] = { 0 };

int bar() {
  return 0;
}
```

LinkerScript:

```
 PHDRS {
A PT_LOAD;
 }
 SECTIONS {
  .abss1 (NOLOAD) : {
    *(.bss.abss1)
  } :A
  .abss2 (NOLOAD) : {
    *(.bss.abss2)
  } :A
  .abss3 (NOLOAD) : {
    *(.bss.abss3)
  } :A
  . = 0x80000000;
  .f : {
```

```
    *(.text.foo)
  } :A
  .bss1 (NOLOAD) : {
    *(.bss.bss1)
  } :A
  .bss2 (NOLOAD) : {
    *(.bss.bss2)
  } :A
  .bss3 (NOLOAD) : {
    *(.bss.bss3)
  } :A
  .bar : {
   *(.text.bar)
  } :A
 }
```

Commands:

```
$clang -c 1.c -ffunction-sections -fdata-sections -G0
$ld.eld 1.o -T script.t
```

Output:

```
$ readelf -l -W a.out

Elf file type is EXEC (Executable file)
Entry point 0x0
There is 1 program header, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x004000 0x00000000 0x00000000 0x8000c19c 0x8000c19c RWE 0x1000

 Section to Segment mapping:
  Segment Sections...
   00     .abss1 .abss2 .abss3 .f .bss1 .bss2 .bss3 .ba
```

NOLOAD sections placed at beginning of segment

When NOLOAD sections are placed at the beginning of the segment, they are not assigned to any segment.

The sections are assigned NOBITS section property.

```
cat > script.t << \!
PHDRS {
  A PT_LOAD;
}

SECTIONS {
  .foo (NOLOAD) : {
            *(.text.foo)
            . = . + 0x4000;
          }
  .bar (NOLOAD) : { *(.text.bar) }
  .baz : { *(.text.baz) } :A
  /DISCARD/ : { *(.eh_frame) *(.ARM.exidx*) }
}
!

cat > 1.c << \!
int foo() { return 0; }
int bar() { return 0; }
```

```
int baz() { return 0; }
!

$clang -c 1.c -ffunction-sections -fdata-sections -G0
$ld.eld 1.o -T script.t

$ readelf -l -W a.out

Elf file type is EXEC (Executable file)
Entry point 0x0
There is 1 program header, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x001020 0x00004020 0x00004020 0x0000c 0x0000c R E 0x1000

 Section to Segment mapping:
  Segment Sections...
   00     .baz
```

LOAD sections following NOLOAD sections

If there is a LOAD section and the the previous section was a NOLOAD section, the load section is not assigned program headers automatically.

```
cat > script.t << \!
PHDRS {
  A PT_LOAD;
}

SECTIONS {
  .foo (NOLOAD) : {
            *(.text.foo)
            . = . + 0x4000;
         }
  .bar (NOLOAD) : { *(.text.bar) }
  .baz : { *(.text.baz) }
  /DISCARD/ : { *(.eh_frame) *(.ARM.exidx*) }
}
!

cat > 1.c << \!
int foo() { return 0; }
int bar() { return 0; }
int baz() { return
!

$clang -c 1.c -ffunction-sections
$ld.eld 1.o -T script.t
```

Warning

Section .baz does not have segment assignment in linker script.

Error

Loadable section .baz not in any load segment

Error

Linking had errors.

NOLOAD sections at the beginning of the LOAD segment

NOLOAD sections when assigned to a LOAD segment gets loaded. The section is marked NOBITS.

```
cat > script.t << \!
PHDRS {
  A PT_LOAD;
}

SECTIONS {
  .foo (NOLOAD) : {
            *(.text.foo)
            . = . + 0x4000;
          } :A
  .bar (NOLOAD) : { *(.text.bar) } :A
  .baz : { *(.text.baz) } :A
  /DISCARD/ : { *(.eh_frame) *(.ARM.exidx*) }
}
!

cat > 1.c << \!
int foo() { return 0; }
int bar() { return 0; }
int baz() { return 0; }
!
```

Commands:

```
$clang -c 1.c -ffunction-sections
$ld.eld 1.o -T script.t
```

Output:

```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .foo              NOBITS          00000000 001000 00400c 00  AX  0   0 16
  [ 2] .bar              NOBITS          00004010 001000 00000c 00  AX  0   0 16
  [ 3] .baz              PROGBITS        00004020 005020 00000c 00  AX  0   0 16
  [ 4] .comment          PROGBITS        00000000 00502c 000106 01  MS  0   0  1
  [ 5] .shstrtab         STRTAB          00000000 005132 000033 00      0   0  1
  [ 6] .symtab           SYMTAB          00000000 005168 0000a0 10      7   6  4
  [ 7] .strtab           STRTAB          00000000 005208 00002f 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  p (processor specific)

Program Headers:
```

```
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x001000 0x00000000 0x00000000 0x0402c 0x0402c R E 0x1000

Section to Segment mapping:
 Segment Sections...
  00     .foo .bar .baz
```

User guide

- ELD User Guide (PDF)

Indices and tables

- **genindex**
- **modindex**
- **search**
- Build Dashboard

Index

Symbols

--no-threads	command line option
--no-trampolines	command line option
--no-undefined	command line option
--no-verify	command line option
--no-warn-mismatch	command line option
--no-warn-shared-textrel	command line option
--no-whole-archive	command line option
--noinhibit-exec	command line option
--omagic	command line option
--opt-record-file	command line option
--opt-remarks-filename	command line option
--opt-remarks-format	command line option
--opt-remarks-hotness-threshold	command line option
--opt-remarks-passes	command line option
--opt-remarks-with-hotness	command line option
--patch-base	command line option
--patch-enable	command line option
--plugin-config	command line option
--plugin-opt	command line option [1]
--pop-state	command line option
--print-gc-sections	command line option
--print-memory-usage	command line option
--print-timing-stats	command line option
--progress-bar	command line option
--push-state	command line option
--record-command-line	command line option
--relax	command line option [1]
--relax-xqci	command line option
--repository-version	command line option
--rosegment	command line option
--save-temps	command line option
--script-options	command line option
--sframe-hdr	command line option
--soname	command line option
--sort-common	command line option
--sort-section	command line option
--start-group	command line option
--start-lib	command line option
--start-lib-thin	command line option
--strip-all	command line option
--strip-debug	command line option
--summary	command line option
--symdef	command line option
--symdef-file	command line option
--symdef-style	command line option
--thin-archive-rule-matching-compatibility	command line option
--thinlto-jobs	command line option
--thread-count	command line option
--threads	command line option
--time-region	command line option
--trace-linker-script	command line option
--trace-lto	command line option
--unique-output-sections	command line option
--unresolved-symbols	command line option
--use-mov-veneer	command line option
--use-old-style-trampoline-name	command line option
--use-shlib-undefines	command line option
--verbose	command line option
--version	command line option
--warn-common	command line option
--warn-limit	command line option
--warn-mismatch	command line option
--warn-once	command line option
--warn-shared-textrel	command line option
--whole-archive	command line option
-allow-incompatible-section-mix	command line option
-allow-shlib-undefined	command line option
-b	command line option
-Bdynamic	command line option
-Bgroup	command line option
-Bsymbolic	command line option
-Bsymbolic-functions	command line option
-build-id	command line option
-color	command line option
-compact	command line option
-copy-farcalls-from-file	command line option
-cref	command line option
-d	command line option
-default-script	command line option
-defsym	command line option
-disable-verify	command line option
-display_hotness	command line option
-dump-mapping-file	command line option
-dump-response-file	command line option
-dwodir	command line option
-dynamic	command line option
-dynamic-linker	command line option
-dynamic-list	command line option
-e	command line option
-EL	command line option
-emit-relocs	command line option
-emit-relocs-llvm	command line option
-emit-timing-stats-in-output	command line option
-enable-threads	command line option
-end-group	command line option
-exclude-lto-filelist	command line option
-execute-only	command line option
-export-dynamic	command line option

-export-dynamic-symbol	command line option
-extern-list	command line option
-fini	command line option
-fix-cortex-a53-843419	command line option
-flto	command line option
-flto-options	command line option
-flto-use-as	command line option
-force-dynamic	command line option
-fPIC	command line option
-fropi	command line option
-frwpi	command line option
-g	command line option
-gc-sections	command line option
-hash-style	command line option
-help	command line option
-help-hidden	command line option
-include-lto-filelist	command line option
-init	command line option
-L	command line option
-l	command line option
-lto-sample-profile	command line option
-M	command line option
-m	command line option
-mabi	command line option
-Map	command line option
-MapDetail	command line option
-mapping-file	command line option
-MapStyle	command line option
-march	command line option
-mcpu	command line option
-mllvm	command line option
-mtriple	command line option
-no-dynamic-linker	command line option
-no-emit-relocs	command line option
-no-gc-sections	command line option
-no-pie	command line option
-no-reuse-trampolines-file	command line option
-no-threads	command line option
-no-undefined	command line option
-no-whole-archive	command line option
-nostdlib	command line option
-o	command line option
-opt-record-file	command line option
-orphan-handling	command line option
-pie	command line option
-plugin	command line option
-plugin-activity-file	command line option
-plugin-opt	command line option [1] [2]
-portable	command line option
-print-gc-sections	command line option
-print-timing-stats	command line option
-Qy	command line option
-R	command line option
-r	command line option
-relax	command line option
-reproduce	command line option
-reproduce-compressed	command line option
-reproduce-on-fail	command line option
-rosegment	command line option
-rpath	command line option
-rpath-link	command line option
-save-temps	command line option [1]
-section-start	command line option
-shared	command line option
-soname	command line option
-sort-common	command line option
-start-group	command line option
-static	command line option
-symdef	command line option
-sysroot	command line option
-T	command line option
-t	command line option
-target2	command line option
-Tbss	command line option
-Tdata	command line option
-thin-archive-rule-matching-compatibility	command line option
-threads	command line option
-trace	command line option
-trace-merge-strings	command line option
-trace-reloc	command line option
-trace-section	command line option
-trampoline-map	command line option
-Ttext	command line option
-Ttext-segment	command line option
-u	command line option
-unique-output-sections	command line option
-verbose	command line option
-verify-options	command line option
-version-script	command line option
-W<warn-type>	command line option
-whole-archive	command line option
-wrap	command line option
-Y	command line option
-y	command line option
-z	command line option

C

command line option

–no-fatal-internal-errors
–no-fatal-warnings
–no-fix-cortex-a8
–no-gc-sections
–no-merge-strings
–no-omagic
–no-record-command-line
–no-relax
–no-relax-c
–no-relax-gp
–no-relax-tlsdesc
–no-relax-xqci
–no-relax-zero
–no-threads
–no-trampolines
–no-undefined
–no-verify
–no-warn-mismatch
–no-warn-shared-textrel
–no-whole-archive
–noinhibit-exec
–omagic
–opt-record-file
–opt-remarks-filename
–opt-remarks-format
–opt-remarks-hotness-threshold
–opt-remarks-passes
–opt-remarks-with-hotness
–patch-base
–patch-enable
–plugin-config
–plugin-opt [1]
–pop-state
–print-gc-sections
–print-memory-usage
–print-timing-stats
–progress-bar
–push-state
–record-command-line
–relax [1]
–relax-xqci
–repository-version
–rosegment
–save-temps
–script-options
–sframe-hdr
–soname
–sort-common
–sort-section
–start-group
–start-lib
–start-lib-thin
–strip-all
–strip-debug
–summary
–symdef
–symdef-file
–symdef-style
–thin-archive-rule-matching-compatibility
–thinlto-jobs
–thread-count
–threads
–time-region
–trace-linker-script
–trace-lto
–unique-output-sections
–unresolved-symbols
–use-mov-veneer
–use-old-style-trampoline-name
–use-shlib-undefines
–verbose
–version
–warn-common
–warn-limit
–warn-mismatch
–warn-once
–warn-shared-textrel
–whole-archive
-allow-incompatible-section-mix
-allow-shlib-undefined
-b
-Bdynamic

-Bgroup
-Bsymbolic
-Bsymbolic-functions
-build-id
-color
-compact
-copy-farcalls-from-file
-cref
-d
-default-script
-defsym
-disable-verify
-display_hotness
-dump-mapping-file
-dump-response-file
-dwodir
-dynamic
-dynamic-linker
-dynamic-list
-e
-EL
-emit-relocs
-emit-relocs-llvm
-emit-timing-stats-in-output
-enable-threads
-end-group
-exclude-lto-filelist
-execute-only
-export-dynamic
-export-dynamic-symbol
-extern-list
-fini
-fix-cortex-a53-843419
-flto
-flto-options
-flto-use-as
-force-dynamic
-fPIC
-fropi
-frwpi
-g
-gc-sections
-hash-style
-help
-help-hidden
-include-lto-filelist
-init
-L
-l
-lto-sample-profile
-m
-M
-mabi
-Map
-MapDetail
-mapping-file
-MapStyle
-march
-mcpu
-mllvm
-mtriple
-no-dynamic-linker
-no-emit-relocs
-no-gc-sections
-no-pie
-no-reuse-trampolines-file
-no-threads
-no-undefined
-no-whole-archive
-nostdlib
-o
-opt-record-file
-orphan-handling
-pie
-plugin
-plugin-activity-file
-plugin-opt [1] [2]
-portable
-print-gc-sections
-print-timing-stats
-Qy
-r

-R

-relax

-reproduce

-reproduce-compressed

-reproduce-on-fail

-rosegment

-rpath

-rpath-link

-save-temps [1]

-section-start

-shared

-soname

-sort-common

-start-group

-static

-symdef

-sysroot

-t

-T

-target2

-Tbss

-Tdata

-thin-archive-rule-matching-compatibility

-threads

-trace

-trace-merge-strings

-trace-reloc

-trace-section

-trampoline-map

-Ttext

-Ttext-segment

-u

-unique-output-sections

-verbose

-verify-options

-version-script

-W<warn-type>

-whole-archive

-wrap

-y

-Y

-z