Qualcomm Technologies, Inc.

Qualcomm® Hexagon™ Utilities

User Guide

80-N2040-1584 Rev. D

November 3, 2020

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

Contents

Figures

Tables

1 Introduction

This document describes the Qualcomm® Hexagon™ utilities, which are a set of software tools used to create and manage object code. They are used with compilers, debuggers, and profilers to support software development for the Hexagon processor.

While the Hexagon utilities are designed to be feature-compatible with the GNU binutils, a few differences exist. This document describes the differences.

1.1 Conventions

`Courier` font is used for computer text and code samples, for example, `hexagon_<function_name>()`.

The following notation is used to define command syntax:

- Square brackets enclose optional items, for example, `[label]`.
- **Bold** indicates literal symbols, for example, `[comment]`.
- The vertical bar character, `|`, indicates a choice of items.
- Parentheses enclose a choice of items, for example, `(add|del)`.
- An ellipsis, `. . .` , follows items that can appear more than once.

1.2 Technical assistance

For assistance or clarification on information in this document, submit a case to Qualcomm Technologies, Inc. (QTI) at https://createpoint.qti.qualcomm.com/.

If you do not have access to CreatePoint, register for access or send email to qualcomm.support@qti.qualcomm.com.

2 Overview

The Hexagon utilities include the following tools:

Table 2-1 Hexagon utilities

Utility	Command	Description
Assembler	`hexagon-llvm-mc`	Translate assembly source files into object files
Linker	`hexagon-link`	Combine object files into executable
Archiver	`hexagon-ar`	Create, modify, and extract files from archives
Object file symbol lister	`hexagon-nm`	List symbols from object files
Object file copier	`hexagon-llvm-objcopy`	Copy and translate object files
Object file viewer	`hexagon-llvm-objdump`	Display contents of object files
Archive indexer	`hexagon-ranlib`	Generate index to archive contents
Object file section size lister	`hexagon-size`	List section sizes and total size
Object file string lister	`hexagon-llvm-strings`	List strings from object files
Object file stripper	`hexagon-strip`	Remove symbols from object file
C++ filter	`hexagon-c++filt`	Filter used to demangle encoded C++ symbols
Address converter	`hexagon-addr2line`	Convert addresses to file and line
DWARF interpreter	`hexagon-addr2srce`	Reads DWARF-based metrics from ELF files
ELF file viewer	`hexagon-llvm-readelf`	Display contents of ELF format files

2.1 Processor versions

The Hexagon utilities support versions V5x and V6x of the Hexagon processor.

- The assembler defines command options (`-mv5`, `-mv55`, and so on) that specify the processor version for which it will generate output files.
- The linker supports the same command options to specify the version of the object file it will link. Attempts to link object files with different processor versions will result in an error message.
- The other utilities automatically determine the processor version of an input object file from information stored in the file.

For more information on these and other command options, see Chapter 3 and Chapter 4.

Not all processor versions are supported in a specific Hexagon tools release. Table 2-2 lists the versions supported in each release.

NOTE: You must include the appropriate version option when using the assembler command.

Table 2-2 Supported processor versions

Tools release	Processor versions
7.2.x	V5, V55, V56V60, V60 HVX, V61
7.3.x	
8.0.x	V5, V55, V56, V60, V60 HVX, V61, V62, V62 HVX
8.1.x	V5, V55, V56, V60, V60 HVX, V61, V62, V62 HVX, V65, V65 HVX
8.2.x	V5, V55, V56, V60, V60 HVX, V61, V62, V62 HVX, V65, V65 HVX, V66, V66 HVX
8.3.x	V55, V56, V60, V60 VX, V61, V62, V62 HVX, V65, V65 HVX, V66, V66 HVX, V67, V67 Small Core (V67t)
8.4.x	V62, V62 HVX, V65, V65 HVX, V66, V66 HVX, V67, V67 Small Core (V67t), V68, V68 HVX

NOTE: The Hexagon utilities do not define command options specifically for V56, but they do support V56 if you use the V55 command options. For more information on the versions, see the *Qualcomm Hexagon Programmer's Reference Manual*.

2.2 Using the utilities

The Hexagon utilities are virtually identical to the corresponding Hexagon GNU binutils. Except for the Hexagon prefix, their command names are mostly identical, and they support many of the GNU command options.

2.3 System requirements

The Hexagon utilities are part of the software development tools for the Hexagon processor, which run on the Windows and Linux operating systems.

3 Assembler

The Hexagon assembler translates Hexagon assembly language into object code. Object files are stored in ELF format.

The Hexagon assembler is mostly compatible with the syntax and directives supported in the GNU assembler, and it supports all features such as macros, conditional text, and include files.

3.1 Command

To start the assembler from the command line:

`hexagon-llvm-mc -filetype=`*`obj`* `[`*`option`*`...] [`*`input_file`*`...]`

Except for the different command name and the required option, `-filetype=`*`obj`*, the assembler is invoked like the GNU assembler.

3.2 Options

Option names can be truncated if they uniquely identify the option.

Assembler information (Section 3.2.1)

```
-help
-help_hidden
-version
```

Input and output files (Section 3.2.2)

```
-filetype =(asm|null|obj)
-main-file-name=string
-o filename
```

Assembly translation (Section 3.2.3)

```
-as-lex
-assemble
-disable-cfi
-I=pathname
-L
-mno-compound
-mno-fixup
-mno-pairing
-n
```

Code generation (Section 3.2.4)

```
-code-model=(default|small|kernel|medium|large)
-g
-mc-no-exec-stack
-mc-relax-all
-relocation-model=(default|static|pic|dynamic-no-pic)
```

Assembler messages (Section 3.2.5)

```
-fatal-assembler-warnings
-mfalign-warn
-mfalign-warn-instruction
-mfalign-warn-packet
```

Assembly output (Section 3.2.6)

```
-disassemble
-hdis
-mdis
-output-asm-variant=uint
-show-encoding
-show-inst
-show-inst-operands
-stats
```

Target (Section 3.2.7)

```
-mv5   |  -mv55  | -mv60  |  -mv61   |  -mv62  |  -mv65  |  -mv66
-mv67  |  -mv67t  |  -mv68
-arch=string
-mattr=a1,+a2,-a3
-triple=string
```

3.2.1 Assembler information

`-help`

Display the common assembler options.

`-help_hidden`

Display all assembler options.

`-version`

Display the assembler release version.

3.2.2 Input and output files

`-filetype =(asm|null|obj)`

Output file type:

`asm`	Assembly file (`*.s`) (Default)
`null`	No output file (used for timing purposes)
`obj`	Hexagon object file (`*.o`)

`-main-file-name=`*`string`*

Name of input file to be treated as primary program file.

`-o` *`filename`*

Output file name.

3.2.3 Assembly translation

`-as-lex`

Lexically scan tokens from the specified *.s file.

`-assemble`

Assemble the specified *.s file. (Default: enabled).

`-disable-cfi`

Ignore any `.cfi_*` directives in the assembly source code.

`-I=`*`pathname`*

Pathname used to search for assembly include files. (Default: current directory).

`-L`

Keep temporary symbols in generated symbol table.

`-mno-compound`

Disable searching for Hexagon compound instructions during assembly.

`-mno-fixup`

Disable fixing resolved relocations during assembly.

`-mno-pairing`

Disable searching for Hexagon duplex instructions during assembly.

`-n`

Do not assume assembly file starts in the text section.

3.2.4 Code generation

`-code-model=(default|small|kernel|medium|large)`

This option is accepted by the assembler, but it is ignored.

`-g`

Generate DWARF debug information for assembly source files.

`-mc-no-exec-stack`

This option is accepted by the assembler, but it is ignored.

`-mc-relax-all`

Relax all code fix-ups.

`-relocation-model=(default|static|pic|dynamic-no-pic)`

This option is accepted by the assembler, but it is ignored.

3.2.5 Assembler messages

`-fatal-assembler-warnings`

Treat warning messages as errors.

`-mfalign-warn`

Warn when there is both `.falign` packet and instruction activity.

NOTE: This message is effective only for the standalone `hexagon-llvm-mc` assembler or when `-fno-integrated-as` is passed to hexagon-clang.

`-mfalign-warn-instruction`

Warn when there is `.falign` instruction activity only.

NOTE: This message is effective only for the standalone `hexagon-llvm-mc` assembler or when `-fno-integrated-as` is passed to hexagon-clang.

`-mfalign-warn-packet`

Warn when there is `.falign` packet activity only.

NOTE: This message is effective only for the standalone `hexagon-llvm-mc` assembler or when `-fno-integrated-as` is passed to hexagon-clang.

3.2.6 Assembly output

`-disassemble`

Disassemble strings of hexadecimal bytes.

`-hdis`

Disassemble strings of hexadecimal bytes, printing immediate values in hexadecimal format.

`-mdis`

Disassemble strings of hexadecimal bytes, marking up the disassembled code.

`-output-asm-variant=uint`

Syntax variant to use for printing output. To list the possible values, run the assembler with the `-version` option.

`-show-encoding`

Display instruction encodings.

`-show-inst`

Display internal instruction representations.

`-show-inst-operands`

Display instruction operands as they are parsed.

`-stats`

Enable statistics output from program.

3.2.7 Target

```
-mv5
-mv55
-mv60
-mv61
-mv62
-mv65
-mv66
-mv67
-mv67t
-mv68
```

Specify the Hexagon processor version for which the assembler will generate code.

NOTE: For more information on these processor versions, see the *Qualcomm Hexagon Programmer's Reference Manual*.

NOTE: Not all of the `-mvX` options are supported in a specific Hexagon tools release. The default option setting is also release-specific. For more information, see Section 2.1.

`-arch=`*`string`*

Target processor architecture. To list the possible values, run the assembler with the `-version` option.

`-mattr=`*`a1,+a2,-a3`*

Target-specific attributes. To list the possible values, run the assembler with the `-mattr=help` option.

`-triple=`*`string`*

Target triple. To list the possible values, run the assembler with the `-version` option.

NOTE: For more information on processor versions, see the *Qualcomm Hexagon Programmer's Reference Manual*.

3.3 Basic syntax

This section describes the syntax of Hexagon assembly language.

3.3.1 Symbols

A symbol consists of the following characters:

- `A ... Z, a ... z`
- `0 ... 9`
- `_, .,` or `$`

Symbols must begin with a letter and are case-sensitive.

3.3.2 Whitespace

Whitespace is defined as one or more blank or tab characters. It is used in assembly code to separate symbols. The assembler interprets whitespace as a single space character.

NOTE: Whitespace does not appear in character constants.

3.3.3 Keywords

Keywords are symbols that are reserved by the assembler and cannot be redefined in the assembly code.

Registers, assembly language instructions, and assembly directive parameters are all defined as keywords.

3.3.4 Directives

Assembler directives are commands that are embedded in the assembly code. They are used to declare items such as constants and external symbols, and to control assembler features such as macros and include files.

Directives always begin with a period character (`.`) immediately followed by the directive name and (optionally) one or more parameters. The names are case-sensitive. The parameters are delimited by whitespace. For example:

```
.set const_val, 1
.include “myfile.s”
```

For more details on directives, see Section 3.6.

3.3.5 Numbers

Numbers can be expressed in several numeric formats. The formats are specified with the number's prefix character.

Table 3-1 Number formats

Format	Prefix	Example
Binary	0b or 0B	`0b1111`
Octal	0	`077`
Decimal	non-0 digit	`1028`
Hexadecimal	0x or 0X	`0Xffff`
Floating point	0e or 0E	`0e3.14159E-3`

Floating point numbers can include the following elements:

- A sign character (`+` or `-`)
- An integer part that consists of 0 or more decimal digits
- A fractional part that consists of a period (`.`) followed by 0 or more decimal digits
- An exponent part that consists of the following elements:
  - The prefix E or e
  - A sign character (`+` or `-`)
  - One or more decimal digits

An integer or fractional part is required, while all the other elements are optional.

3.3.6 Characters

String constants are delimited by a pair of double-quote characters (`"`). For example:

```
"mydata"
```

Special characters can be used in characters and strings by prefixing them with a backslash character (`\`). This example represents a single backslash, where the first backslash acts as the prefix character for the second:

```
\\
```

NOTE: When used in a numeric expression, the value of a character constant is its 8-bit ASCII code value.

Table 3-2 Escape characters to use in characters and strings

Code	Character
\b	Backspace
\f	Form feed
\n	Newline
\r	Return
\t	Tab
\\	Backslash
\"	Double quote
\NNN	Octal character value NNN specifies three octal digits
\xhex...	HEX character value hex... specifies N hexadecimal digits

3.3.7 Expressions

Expressions consist of one or more symbols (representing numeric or address values) that are operated on by one or more operators.

Expressions can be used wherever numeric or address values are allowed, including variables and instructions.

NOTE: An expression must evaluate to either an absolute value or an offset into a section (Section 3.4). Otherwise, it will be flagged with an error message.

3.3.8 Operators

Operators are characters that represent arithmetic functions such as addition or multiplication.

Prefix operators accept a single operand, which must be an absolute value. The result is absolute.

Table 3-3 Prefix operators

Operator	Description
-	Two's complement arithmetic negate
~	Bitwise logical NOT

Infix operators accept two operands: + and -. They can operate on relocatable values. For all the other infix operators, the operand values must be absolute and the result is absolute.

Table 3-4 Infix operators

Operator	Description	Precedence
*	Multiply	Highest
/	Divide	
%	Remainder	
< <<	Shift left	
>> >>	Shift right	
|	Bitwise inclusive OR	Intermediate
&	Bitwise AND	
^	Bitwise exclusive OR	
!	Bitwise OR NOTE	
-	Two's complement arithmetic negate	Lowest
~	Bitwise logical NOT	

The infix operators have differing precedence—operators with the highest precedence are evaluated first in an expression, and operators with equal precedence are evaluated left to right in an expression.

3.3.8.1 Relocatable operands

If an absolute operand is added to a relocatable operand, the result belongs to the section of the relocatable operand. Two relocatable operands cannot be added if they belong to different sections.

If an absolute operand is subtracted from a relocatable operand, the result belongs to the section of the relocatable operand. If both operands are relocatable and in the same section, the result of a subtract is absolute. Two relocatable operands cannot be subtracted if they belong to different sections.

3.3.9 Statements

A statement is considered to be equivalent to one line in the assembly source code. It is terminated with a newline character (\n).

```
statement \n
```

A single statement can span multiple lines in the assembly source code if you place a backslash character (\) at the end of each line. Place the newline character at the end of the last line.

```
statement \
continued statement \
last line of statement \n
```

NOTE: Every statement in an assembly source file (including the last statement in the file) must be terminated with a newline character.

3.3.10 Comments

Delimited comments begin with the /* character sequence and end with */. They can span multiple lines and cannot be nested. For example:

```
/* delimited comment */
```

Line comments begin with // and end at the newline character on the current line. For example:

```
code    // line comment \n
```

3.4 Sections

Assembly language programs are organized into collections of units known as *sections*. For example:

```
.section     .text
      // Hexagon instructions

.section     .data
      // variable declarations
```

Each section is defined to occupy a range of consecutive addresses. When the linker merges object files into a single executable image, it relocates the starting address of each section in the object files so the sections do not overlap in memory.

- A program contains a minimum of three sections:
  - `text` — Contains program code and is assigned to address 0
  - `data` — Is relocated to immediately follow the `text` section in memory
  - `bss` — Is relocated to follow the `data` section
- The `sdata` and `sbss` sections contain data (such as common symbols) that is assigned to the global data area.
- The `ebi_*`, `tcm_*`, and `smi_*` sections contain data that is assigned to system-specific memories with defined cache properties. For more information, see the *Qualcomm Hexagon Programmer's Reference Manual*.

Table 3-5 Assembly sections in an assembly program

Section name	Description

`.text`
`.data`
`.rodata`
	Program code and data.
	`.rodata` is used to store read-only data.

`.bss`	Uninitialized variables or common data. `.bss` does not occupy any space in an object file because the section data is always initialized to zero when the section is loaded into memory.
`.sdata`	Global `.data` section
`.sbss`	Global `.bss` section
`.ebi_code_cached`	Code ■ `ebi` = external bus interface ■ `tcm` = tightly-coupled memory ■ `smi` = stacked memory interface
`.tcm_code_cached`	
`.smi_code_cached`	
`.ebi_data_cached`	Data (cached)
`.tcm_data_cached`	
`.smi_data_cached`	
`.ebi_data_cached_wt`	Data (write-through cached)
`.tcm_data_cached_wt`	
`.smi_data_cached_wt`	

Table 3-5 Assembly sections in an assembly program (cont.)

Section name	Description
.ebi_data_uncached	Data (uncached)
.tcm_data_uncached	
.smi_data_uncached	

If sections are not explicitly declared in the source code, the assembler automatically assigns the program code and data to EBI memory.

The linker typically assigns all sections of the same type to occupy contiguous addresses in memory. This can be changed by creating a linker script (Section 4.4). Use the linker to assign the system-specific memory sections to specific memory areas (for more information, see Section 3.4.2).

NOTE: Code and data cannot be stored together in the same system-specific section.

3.4.1 User-defined sections

User-defined sections are defined when a program is to store code or data in a non-standard configuration. They are defined by specifying section names that differ from the predefined sections (Table 3-5). For example:

```
.section my_strings
```

A user-defined section declaration does not specify a section type (`code`, `data`, and so on). The assembler automatically determines the section type from the type of objects stored in the section.

To make the program more readable, it is common practice to prefix the name of a user-defined section with the section type. For example:

```
.section .rodata.my_strings
```

3.4.2 Subsections

To provide finer control over how items are assigned to memory within a section, the assembler supports subsections for organizing that assignment. For example:

```
.data 1                                    // subsection 0
.ascii "String assigned to sub-section 1"
.data 2                                    // subsection 1
.ascii "String assigned to sub-section 2"
```

The `.text` and `.data` assembler directives accept an argument in the range 0 to 8192. This argument specifies the subsection to which any subsequent declarations are assigned within the current text or data section. The assembler ensures that all items declared in the same subsection are assigned to adjacent addresses in the relevant section.

The default subsection is 0.

NOTE: Negative subsections are not supported.

3.5 Symbols

This section describes how symbols (Section 3.3.1) are used in Hexagon assembly language.

3.5.1 Symbol assignment

Symbols are assigned values with the `.set` directive. For example:

```
.set my_symbol, 78
```

3.5.2 Temporary symbols

Temporary symbols work like regular symbols in the program source code, but they are typically not saved in the object file. Thus, they are not visible when linking, debugging, or profiling.

Temporary symbols are prefixed with `.L`. For example:

```
.L_temp_label
```

The `-L` option causes the assembler to save temporary symbols in the object file as local symbols. If the linker is also directed to save local symbols, the resulting temporary symbols can be used in debugging and profiling.

NOTE: Temporary symbols differ from ELF local and global symbol types.

3.5.3 Labels

Labels are symbols immediately followed by a colon (`:`). For example:

```
my_label:
```

Because labels can cause problems when profiling, do not use them in embedded assembly functions. Instead, use local labels.

3.5.4 Local labels

Local labels can be specified as either symbolic labels or numeric labels.

Symbolic local labels are declared and used like regular labels, but they specify a temporary symbol as the label symbol. For example:

```
.L_loc:
```

Numeric local labels consist of an integer immediately followed by a colon (`:`). For example:

```
99:
```

Numeric local labels can reuse the same set of numeric label names (`0:`, `1:`, `2:`, ...) throughout a program.

To reference the preceding definition of a numeric label, write `Nb` (where `N` specifies the integer in the label). To reference a subsequent label definition, write `Nf`. For example:

```
99: nop;
    jump 99b;       // jump to preceding local label 99
99: nop;
```

A program can reference only the immediately preceding and following instances of a numeric label symbol.

The assembler transforms numeric local labels into unique symbolic local labels (as defined in the example). The generated name includes the numeric local label integer, an ASCII control character (Ctrl-B), and a second integer that indicates the instance of the numeric local label. For example:

```
.L99<ctrl-B>6
```

NOTE: Symbolic local label names can be saved in an object file (Section 3.5.2).

3.5.5 Dot symbol

When used as a symbol, the period character (`.`) specifies the current address for which the assembler is generating code. For example:

```
my_var: .word  .
```

The period following the `.word` directive (Section 3.6) specifies the value of the data allocated by `.word`. The value is the memory address of the data.

3.5.6 Symbol attributes

Symbol have the following attributes.

3.5.6.1 value

Symbol values are generally 32-bit numbers. The values can be explicitly assigned in the program code or implicitly assigned by the assembler.

- The value of a data symbol is the section-relative offset of the data item. This value is relocatable and is resolved by the linker.
- The value of an undefined symbol is 0. This value is resolved by the linker.
- The value of a common symbol is the number of bytes to reserve for the symbol in common memory. This value is resolved by the linker.

3.5.6.2 type

The symbol type indicates whether a symbol identifies a function or object. It is set with `.type` or `.stabs`.

3.5.6.3 size

The symbol size indicates the size (in bytes) of the item identified by the symbol. It is set with `.size` or `.stabs`.

3.5.6.4 scope

The symbol scope indicates whether a symbol is local or global. The scope can be made global with `.global`.

3.5.6.5 visibility

The symbol visibility indicates whether a symbol is protected, hidden, or internal.

- Protected symbols always resolve to symbol definitions in the current component, even if definitions exist in other components that might normally supersede the local definition.
- Hidden symbols are protected symbols that are not visible in other components.
- Internal symbols are hidden symbols that require extra processing.

The visibility is set with `.protected`, `.hidden`, or `.internal`.

3.5.6.6 descriptor

The symbol descriptor contains an arbitrary 16-bit value. It is set with `.stabs`.

3.5.7 Symbol modifiers

Symbol modifiers are used to transform the value of the referenced symbol. For example:

```
r0 = add (pc, ##bar@PCREL)
```

In this example, the `@PCREL` modifier causes the symbol bar to be evaluated as a PC-relative offset.

The following table lists the symbol modifiers and the Hexagon instructions in which they can be used.

Table 3-6 Symbol modifiers

Modifier	Description
`#HI (symbol)`	Bits [32:16] `r1.h = #HI (my_symbol)`
`#LO (symbol)`	Bits [15:0] `r1.l = #LO (my_symbol)`
`CONST32 (symbol)`	32-bit constant `r12 = CONST32 (#my_symbol)`
`CONST64 (value)`	64-bit constant from hexadecimal value `r13:12 = CONST64 (#0xffff)`
`symbol@PCREL`	PC-relative offset `r0 = add (pc, ##my_symbol@PCREL)`
`symbol@GOT`	Offset of the entry from the global offset table (GOT) `r2 = memw (r0 + ##my_symbol@GOT)`
`symbol@GOTREL`	Offset to the location from the GOT `r4 = my_symbol@GOTREL`
`symbol@PLT`	Offset to the procedure linkage table (PLT) entry from the current code location `call my_label@PLT`
`symbol@GDPLT`	Offset to the PLT entry for the function that returns the address of a thread-local storage (TLS) symbol from the current code location (using the GD method) `call my_label@GDPLT`
`symbol@GDGOT`	Offset of the first GOT entry for the TLS_index structure from the GOT (using the GD method) `r0 = add (r14, ##my_symbol@GDGOT)`
`symbol@LDPLT`	Offset to the PLT entry for the function that returns the address of a TLS symbol from the current code location (using the LD method) `call my_label@LDPLT`
`symbol@LDGOT`	Offset of the first GOT entry for the TLS_index structure from the GOT (using the LD method) `r0 = add (r14, ##my_symbol@LDGOT)`
`symbol@DTPREL`	Offset to the position from the base of the TLS template (using the LD method) `r23 = add (r0, ##my_symbol@DTPREL)`

Table 3-6 Symbol modifiers (cont.)

Modifier	Description
`symbol@IE`	Address of the GOT entry for the offset into the TLS area (using the IE method) `r12 = memw (##my_symbol@IE)`
`symbol@IEGOT`	Offset of the GOT entry for the offset into the TLS area from the GOT (using the IE method) `r13 = memw (r14 + ##my_symbol@IEGOT)`
`symbol@TPREL`	Offset to the position from the base of the TLS area (using the LE method) `r13 = add (r15, ##my_symbol@TPREL)`

NOTE: The symbol modifiers can be used only in specific instructions. For more information, see the *Qualcomm Hexagon Application Binary Interface Specification* (80-N2040-23).

3.6 Directives

Assembler directives are used to declare sections, constants, data, and symbols, and to control assembler features such as macros and include files. Directives are prefixed with a period character (.) and might be followed by one or more arguments.

The assembler recognizes the following directives.

Sections (Section 3.6.1)

```
.data [subsection]
.section name [,"flags" [,type]]
.text [subsection]
```

Symbols (Section 3.6.2)

```
.common symbol, length [,align] [,access]
.comm symbol, length [,align] [,access]
.equ symbol, expr
.equiv symbol, expr
.global symbol
.globl symbol
.lcommon symbol, length [,align] [,access]
.lcomm symbol, length [,align] [,access]
.set symbol, expr
.symver name, name2@nodename
.symver name, name2@@nodename
.symver name, name2@@@nodename
```

Attributes (Section 3.6.3)

```
.hidden symbol [,symbol]...
.internal symbol [,symbol]...
.protected symbol [,symbol]...
.size symbol, expr
.type symbol, type
```

Alignment (Section 3.6.4)

```
.align abs-expr [,abs-expr] [,abs-expr]
.balign[w|l] abs-expr [,abs-expr] [,abs-expr]
.falign
.p2align[w|l] abs-expr [,abs-expr] [,abs-expr]
.org new-lc [,fill]
```

Data (Section 3.6.5)

```
.ascii ["string" [,"string"]...]
.asciz ["string" [,"string"]...]
.byte [expr [,expr]...]
.2byte [expr [,expr]...]
.4byte [expr [,expr]...]
.double [flonum [,flonum]...]
.fill repeat, size, value
```

```
.float [flonum [,flonum]...]
.half [expr [,expr]...]
.hword [expr [,expr]...]
.int [expr [,expr]...]
.long [expr [,expr]...]
.quad [bignum [,bignum]...]
.short [expr [,expr]...]
.single [flonum [,flonum]...]
.skip size [,fill]
.space size [,fill]  |  .block size [,fill]
.string "string" [,"string"]...
.word [expr [,expr]...]
```

Conditionals (Section 3.6.6)

```
.else
.elseif
.endif
.if abs-expr
.ifdef symbol
.ifndef symbol
.ifnotdef symbol
```

Macros (Section 3.6.7)

```
.endm
.endr
.irp symbol, value [,value]...
.irpc symbol, string
.macro name
.macro name argument...
.purgem name
```

Include files (Section 3.6.8)

```
.include "filename"
```

Debug (Section 3.6.9)

```
.file fileno filename
.loc fileno lineno [column] [options]
.stabs string, type, other, desc, value
.sleb128 expr [,expr]...
.uleb128 expr [,expr]...
```

Assembler control (Section 3.6.10)

```
.rept [count]
```

3.6.1 Sections

`.data [subsection]`

Assign the following data declarations to the specified data subsection. The value must be absolute. The default value is 0.

`.section name [,"flags" [,type]]`

Specify a new section.

- The `flags` argument specifies one or more section properties: allocatable (`a`), writable (`w`), or executable (`x`). The default property depends on the section name; if it is not recognized, the section contains data but has none of the indicated properties.
- The `type` argument (`@progbits`, `@nobits`) specifies whether a section contains data. A section with no data only occupies space.

`.text [subsection]`

Assign the subsequent Hexagon instructions to the specified code subsection. The value must be absolute. The default value is 0.

3.6.2 Symbols

`.common symbol, length [,align] [,access]`
`.comm symbol, length [,align] [,access]`

Declare a common data item with the specified symbol name and data size.

- The `length` argument specifies the data length (in bytes).
- The `align` argument specifies the data alignment (in bytes). The alignment value must be an integral power of 2. The default value is the largest power of 2 less than or equal to the length value.
- The `access` argument specifies the size (in bytes) of the smallest memory access that will be made to the data item. The default value is the align value.

`.equ symbol, expr`

Set the symbol value.

`.equiv symbol, expr`

Same as `.equ` and `.set`, except that an error is generated if the symbol is already defined.

`.global symbol`
`.globl symbol`

Declare a symbol as global.

`.lcommon` *`symbol, length`* `[,`*`align`*`] [,`*`access`*`]`
`.lcomm` *`symbol, length`* `[,`*`align`*`] [,`*`access`*`]`

Declare a local common data item with the specified symbol name and data size.

- The `length` argument specifies the data length (in bytes).
- The `align` argument specifies the data alignment (in bytes). The alignment value must be an integral power of 2. The default value is the largest power of 2 less than or equal to the length value.
- The `access` argument specifies the size (in bytes) of the smallest memory access that will be made to the data item. The default value is the align value.

`.set` *`symbol, expr`*

Set a symbol to the specified value.

If the symbol was previously specified as external, it remains external.

If the symbol is global, the value that is stored in the object file is the most recent value stored into the symbol.

`.symver` *`name, name2@nodename`*
`.symver` *`name, name2@@nodename`*
`.symver` *`name, name2@@@nodename`*

Bind symbols to specific version nodes in a source file. Do this when developing programs that work with shared objects.

- The `name2` argument specifies the name used to reference the symbol externally.
- The `nodename` argument specifies the name of a node in the linker script that is used when building a shared library.
- The single `@` version of this directive behaves differently depending on whether the symbol is defined in the current source file.

  If it is defined, the symbol is aliased with the name, `name2@nodename`. This alias allows multiple versions of the same function to co-exist in a source file, while still allowing the compiler to be able to distinguish the function references.

  If the symbol is not defined, all references to it are changed to `name2@nodename`.
- The double `@@` version of this directive specifies that the symbol must be defined in the source file. It is the same as the single @ version, except that `name2@@nodename` is also used to resolve references to `name2`.
- The triple `@@@` version specifies that, when the symbol is not defined in the source file, it is treated as `name2@nodename`. Otherwise, it is renamed as `name2@@nodename`.

3.6.3 Attributes

`.hidden` *`symbol`* `[,`*`symbol`*`]...`

Set the symbol visibility of the specified symbols to `hidden`.

`.internal` *`symbol`* `[,`*`symbol`*`]...`

Set the symbol visibility of the specified symbols to `internal`.

`.protected` *`symbol`* `[,`*`symbol`*`]...`

Set the symbol visibility of the specified symbols to `protected`.

`.size` *`symbol`*`,` *`expr`*

Set the symbol size to the specified value (in bytes).

The value must be absolute, but it can be the result of label arithmetic.

`.type` *`symbol`*`,` *`type`*

Set the symbol type.

The type argument can be one of the following values.

Type	Description
`@function`	Function name
`@gnu_indirect_function`	Indirect function name
`@object`	Data object
`@tls_object`	Thread-local storage data object
`@common`	Common data object
`@notype`	Typeless symbol
`@gnu_unique_object`	Globally unique in process

3.6.4 Alignment

`.align` *`abs-expr`* `[,`*`abs-expr`*`]` `[,`*`abs-expr`*`]`

Pad the location counter in the current subsection until it reaches an integral multiple of the value in the first argument.

- The first `abs-expr` argument is the alignment request value (in bytes). It must be an integral multiple of 2 and cannot exceed 0x3ffff.
- The second `abs-expr` argument specifies the value to be stored in the padding data. If the argument is omitted, the default value is 0 (or in a text section, the default value is NOPs).
- The third `abs-expr` argument specifies the maximum number of bytes to be skipped. If the alignment requires skipping more bytes than the specified value, no alignment is performed.

NOTE: The second argument can be omitted by placing two commas between the first and third arguments.

`.balign[w|l] abs-expr [,abs-expr] [,abs-expr]`

Pad the location counter in the current subsection until it reaches an integral multiple of the value in the first argument.

- In `.balignw`, the padding value is a two-byte word value.
- In `.balignl`, the padding value is a four-byte longword value.
- The first `abs-expr` argument is the alignment request value (in bytes). It must be an integral multiple of 2 and cannot exceed 0x3ffff.
- The second `abs-expr` argument specifies the value to be stored in the padding bytes. If the argument is omitted, the default value of each padding byte is 0 (or in a text section, the default value is NOPs).
- The third `abs-expr` argument specifies the maximum number of bytes to be skipped. If the alignment requires skipping more bytes than the specified value, no alignment is performed.

NOTE: The second argument can be omitted by placing two commas between the first and third arguments.

`.falign`
`.falign [packets]`

Align instruction packets to not cross a cache line boundary (or for processor versions before Hexagon V60, a 16-byte boundary).

The packets argument specifies the number of packets the assembler should align to a cache line boundary. The default value is 1, and the upper limit is 2.

Padding is accomplished either by adding NOPs to any empty slots in previous instruction packets or by adding a new instruction packet containing NOPs.

This directive is used to reduce the amount of cache activity for time-critical code. The `.falign` directive is short for *fetch align*.

NOTE: A label cannot be assigned to a source line containing an `.falign` directive.

`.p2align[w|l] abs-expr [,abs-expr] [,abs-expr]`

Pad the location counter in the current subsection until it reaches an integral multiple of 2x, where x is the value of the first argument.

- In `.p2alignw`, the padding value is a two-byte word value.
- In `.p2alignl`, the padding value is a four-byte longword value.
- The first `abs-expr` argument effectively specifies the number of low-order zero bits that the location counter must contain after being advanced. The value must not exceed 17.
- The second `abs-expr` argument specifies the value to be stored in the padding bytes. If the argument is omitted, the default value of each padding byte is 0 (or in a text section, the default value is NOPs).
- The third `abs-expr` argument specifies the maximum number of bytes to be skipped. If the alignment requires skipping more bytes than the specified value, no alignment is performed.

NOTE: The second argument can be omitted by placing two commas between the first and third arguments.

`.org new-lc [,fill]`

Advance the location counter of the current section and allocate pad data as necessary.

- The `new-lc` argument must be either absolute or an expression that evaluates to the same section as the current subsection.
- The `fill` argument specifies the value assigned to the pad data. The default value is 0.

3.6.5 Data

`.ascii ["string" ["string"]...]`

Allocate string literals (with no terminating zero bytes) as character data at consecutive addresses.

`.asciz ["string" ["string"]...]`

Allocate string literals (with automatic terminating zero bytes) as character data at consecutive addresses.

`.byte [expr [,expr]...]`

Allocate the specified expressions as bytes stored at consecutive addresses.

`.2byte [expr [,expr]...]`

Allocate two-byte data items at consecutive addresses.

`.4byte [expr [,expr]...]`

Allocate four-byte data items at consecutive addresses.

`.double [flonum [,flonum]...]`

Allocate the specified `flonums` as 64-bit floating point numbers stored at consecutive addresses.

`.fill repeat, [,size] [,value]`

Allocate fill data at consecutive addresses.

- The `repeat` argument specifies the number of data items to allocate.
- The `size` argument specifies the size (in bytes) of each data item. The default value is 1.
- The optional `value` argument specifies the value assigned to each item. The default value is 0.

NOTE: The second argument can be omitted by placing two commas between the first and third arguments.

`.float [flonum [,flonum]...]`

Allocate the specified `flonums` as 32-bit floating point numbers stored at consecutive addresses.

`.half [expr [,expr]...]`

Allocate the specified expressions as halfwords stored at consecutive addresses.

`.hword [expr [,expr]...]`

Allocate the specified expressions as halfwords stored at consecutive addresses.

`.int [expr [,expr]...]`

Allocate the specified expressions as 32-bit integers stored at consecutive addresses.

`.long [expr [,expr]...]`

Allocate the specified expressions as 32-bit numbers stored in consecutive addresses.

Accepts zero or more expressions separated by commas.

NOTE: This `.long` option is equivalent to `.int`.

`.quad [bignum [,bignum]...]`

Allocate `bignums` as 64-bit integers stored at consecutive addresses.

`.short [expr [,expr]...]`

Allocate the specified expressions as 16-bit numbers stored at consecutive addresses.

`.single [flonum [,flonum]...]`

Allocate `flonums` as 32-bit floating point numbers stored at consecutive addresses.

`.skip size [,fill]`

Allocate bytes at consecutive addresses, using the specified fill value. The default fill value is 0.

`.space size [,fill]`
`.block size [,fill]`

Allocate bytes at consecutive addresses, using the specified fill value. The default fill value is 0.

`.string "string" [,"string"]...`

Allocate string literals as character data stored at consecutive byte addresses.

Each string is terminated by a zero byte.

`.word [expr [,expr]...]`

Allocate expressions as 32-bit words stored at consecutive addresses.

3.6.6 Conditionals

`.else`
`.elseif` *abs-expr*
`.endif`

Conditional assembly directives.

`.if` *abs-expr*
`.ifdef` *symbol*
`.ifndef` *symbol*
`.ifnotdef` *symbol*

Conditional assembly directives.

The condition can be the value of an expression (`.if`), whether a symbol is already defined (`.ifdef`), or whether a symbol is not defined (`.ifndef` and `.ifnotdef`).

3.6.7 Macros

`.endm`

End of the macro definition.

`.endr`

Mark the end of a sequence of source lines processed by the `.rept` directive.

`.irp` *symbol, value* [,*value*]...

Generate a macro-like sequence of similar statements, assigning a different value to the symbol parameter in each statement template.

The statement template is delimited by the `.irp` and `.endr` directives, with the symbol appearing in the template as a parameter prefixed by the backslash character (`\`). For example:

```
.irp    arg,1,2,3
move    x\arg,sp@-
.endr
```

This declaration generates the following assembly statements:

```
move    x1,sp@-
move    x2,sp@-
move    x3,sp@-
```

`.irpc` *symbol, string*

Generate a macro-like sequence of similar statements, assigning a different character from `string` to the `symbol` parameter in each statement template.

The statement template is delimited by the `.irpc` and `.endr` directives, with the symbol appearing in the template as a parameter prefixed by the backslash character (`\`). For example:

```
.irpc   arg,123
move    x\arg,sp@-
.endr
```

This declaration generates the following assembly statements:

```
move    x1,sp@-
move    x2,sp@-
move    x3,sp@-
```

`.macro` *`name`*
`.macro` *`name argument ...`*

Declare a macro.

Macro arguments in the macro declaration must be separated by commas or spaces.

The macro definition is delimited by the `.macro` and `.endm` directives, with the macro parameters appearing in the macro definition as the macro argument name prefixed by the backslash character (`\`).

Declare macro arguments with the following options:

Argument	Description
`arg=default_value`	Default value if argument not specified in macro call.
`arg:req`	Argument must always be assigned non-blank value in macro calls.
`arg:vararg`	Argument assigned all remaining arguments in macro call.

In macro calls, specify arguments either by name (`add p1=17, p2=9`) or by order in the argument list (`add 17,9`).

`.purgem` *`name`*

Delete the macro definition for the specified macro.

Once deleted, the name can be used for a new macro declaration.

3.6.8 Include files

`.include "`*`filename`*`"`

Include the contents of the specified file into the program.

3.6.9 Debug

`.file` *`fileno filename`*

Assign file names to the file name table `.debug_line`.

- The `fileno` argument specifies an index into the file name table. The index value must be unique with respect to any other instances of the `file` directive in the program.
- The `filename` argument is a C string literal.

This directive is used when generating DWARF debug information.

`.loc` *`fileno lineno`* [*`column`*] [*`options...`*]

Add rows to the `.debug_line` line number matrix.

- The `fileno` argument specifies an entry index in the file name table.
- The `lineno` and column arguments specify a file position.
- The `options` argument specifies one or more instances of the following values: `basic_block`, `prologue_end`, `epilogue_end`, `is_stmt` *`value`*, or `isa` *`value`*.

`.stabs` *`name, type, other, desc, value`*

Define symbols that are used by symbolic debuggers.

The directive arguments are used to set the following symbol attributes: `name`, `value`, `type`, `descriptor`, and `other`. These symbols are used for debugging; they cannot be referenced in the assembly source code.

`.sleb128` *`expr`* [`,`*`expr`*]`...`

Allocate the specified expressions as sleb128-format numbers stored at consecutive addresses.

sleb128 is an acronym for signed little-endian base 128. This format is used when generating DWARF debug information.

`.uleb128` *`expr`* [`,`*`expr`*]`...`

Allocate expressions as uleb128-format numbers stored at consecutive addresses.

uleb128 is an acronym for unsigned little-endian base 128. This format is used when generating DWARF debug information.

3.6.10 Assembler control

`.rept` [*`count`*]

Generate a macro-like sequence of statements, where count indicates the number of times the statement sequence is to be repeated in the code. The default value is 0.

The statement sequence is delimited by the `.rept` and `.endr` directives.

3.7 Packet splitting guidance for Small Cores

Hexagon Small Cores have fewer slots available for packetized instructions. Because many instruction packets in hand-written assembly code are invalid for the Small Core, much of the existing assembly source code is unusable on Small Cores without significant redesign. An invalid packet might have too many instructions, or the instructions might not be placed in the correct slots.

NOTE: For an explanation of available slots, see the "Resource constraints" section in the *Qualcomm Hexagon Programmer's Reference Manual*.

Following are steps you can take to ease the porting of assembly source code, written for V5 through V66, to successfully assemble code for V67t (Small Core). These examples are applicable to any Hexagon Small Core, and they are designed to be used with Hexagon 8.3 or later tools.

3.7.1 No packet-splitting option

Using `-mv67t` without a packet-split option, the assembler:

- Uses the default `-mpacket-split-hw` option
- Does not split any packets
- Displays no errors or warnings about `native-small-core-invalid` packets

Using this switch tells the assembler to disregard `native-small-core-invalid` packets, and that any packet splitting will be done by the Hexagon processor at runtime.

3.7.2 Hardware split

Using `-mv67t -mno-packet-split-hw` on the hexagon-clang command line tells the assembler:

- To report all packets that are not compatible with Small Core
- To not attempt to split any packets

This approach is appropriate for Small Core targets that are not expected to have a packet-splitting function in the hardware.

Example

```
hexagon-clang -mv67t -O2 -G0 -g -mno-packet-split-hw -c nosplit.S -o
nosplit.o
nosplit.S:23:1: error: invalid instruction packet: slot error
}
^
nosplit.S:23:1: error: invalid instruction packet: out of slots
}
```

3.7.3 Software split

Using `-mv67t -mpacket-split-sw` on the hexagon-clang command line tells the assembler:

- To split all the invalid packets that it can
- To flag as errors those packets it cannot split

Example

```
hexagon-clang -mv67t -O2 -G0 -g -mpacket-split-sw -c nosplit.S -o nosplit.o
nosplit.S:23:1: error: failed to split this packet
}
```

3.7.4 Atomicity semantics

However you choose to alter the packets in existing Hexagon assembly source code to comply with Small Core constraints, the resulting packets will probably operate differently from the original code at the atomic level.

For example, if a single packet reads a register and clears a bit in a memory-mapped input/output register, splitting these operations into two packets now allows for an interrupt to arrive between these two packets.

The revised (Small Core) code must account for these new possibilities.

3.7.5 Instruction serialization

To preserve the functionality of packets being split, the assembler internally creates a serialized representation of the instructions in the packet. This representation is of the dependencies among the parts of instructions that would execute serially on the Hexagon processor. The rules for these dependencies are described in the *Qualcomm Hexagon Programmer's Reference Manual*.

3.7.6 Memory ordering

For maximum fidelity, the assembler's packet splitter orders multiple memory operations such that their effects match the behavior of the target Hexagon processor.

If you know in advance that multiple memory operations cannot have an alias or that their effects do not require this ordering, you can disable this feature:

- Globally with the `-mpacket-mem-no-order` flag
- Locally to a packet with the `:mem_no_order` attribute

By disabling this feature, the assembler's packet splitter will not map these dependencies and will be less likely to encounter cyclical dependencies.

3.7.7 Examples

The `nosplit.S` source file includes one packet with four instructions:

```
{
  r0=memw(r6)
  r3=r5
  r5=r7
  memw(r7)=r0
}
```

When the packet splitter considers this packet according to the serialization rules of the target architecture, it finds an unresolvable dependency cycle among the instructions. This packet cannot be split by the assembler.

Assemble nosplit.S with default arguments and then with -mpacket-split-sw

In this example, the assembler with default arguments throws no warning or error on this packet when building for `-mv67t`. But using `-mpacket-split-sw` throws an error on this packet:

```
hexagon-clang -mv67t -O2 -G0 -g -c nosplit.S -o nosplit_test.o
No error or warning about the four-instruction packet
-----------------------------------
hexagon-clang -mv67t -O2 -G0 -g -mpacket-split-sw -c nosplit.S -o
nosplit.o
nosplit.S:23:1: error: failed to split this packet
}
^
```

Write new assembly source code to manage packet splitting inherently

To design the source file so it builds for standard or Small Core Hexagon processors without using the `-mpacket-split-sw` switch, add `#if __HEXAGON_PHYSICAL_SLOTS__ == 3 } { #endif` conditionals where you want packets to be split. This method makes it clear how packets must be split, focusing attention on the effects of these splits.

`__HEXAGON_PHYSICAL_SLOTS__` is a default hexagon-clang macro that is automatically set to the number of slots available in the designated processor. Select it by using the `-mv_` switch:

- For `-mv67`, `__HEXAGON_PHYSICAL_SLOTS__ = 4`
- For `-mv67t`, `__HEXAGON_PH YSICAL_SLOTS__ = 3`

3.7.7.1 Packet comparison

This example shows a comparison of packets that the assembler can split with `-mpacket-split-sw`, and packets that are manually split by inserting preprocessor directives.

```
split_soft.S
// packet splitting for Small Core

.text

.global small_split
.type small_split, @function

small_split:

// load and store in the same packet
load_store_asm:
{ r6=r0
  r1=memw(r0)
  r2=r7
  memw(r4)=r3
}

// load and store in the same packet
load_store_man:
{ r6=r0
  r1=memw(r0)
#if __HEXAGON_PHYSICAL_SLOTS__ == 3
} {
#endif
  r2=r7
  memw(r4)=r3
}


// hw loop
loop_split_asm:
{
  loop0(base_a,#7)
  r9:8=memd(r24+#8)
  r15:14=memd(r24++#16)
}
base_a:
{
  memd(r2++#16)=r15:14
  r1:0=memd(r24++#16)
}:endloop0


// hw loop
loop_split_man:
{
  loop0(base_m,#7)
  r9:8=memd(r24+#8)

#if __HEXAGON_PHYSICAL_SLOTS__ == 3
```

```
} {
#endif
  r15:14=memd(r24++#16)
}
base_m:
{
  memd(r2++#16)=r15:14
#if __HEXAGON_PHYSICAL_SLOTS__ == 3
} {
#endif
  r1:0=memd(r24++#16)
}:endloop0

// two instructions in execute slot
exec_split_asm:
{
    usr = r28 // slot X only
    r5 ^= lsr(r1,#31)// slot X only
    r28 = r5
}


exec_split_man:
{
    usr = r28 // slot X only
#if __HEXAGON_PHYSICAL_SLOTS__ == 3
} {
#endif
    r5 ^= lsr(r1,#31) // slot X only
    r28 = r5
}


.size small_split, .-small_split
```

4 Linker

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

The linker supports most, but not all, features in the Hexagon GNU linker.

4.1 Command

To start the linker from the command line:

hexagon-link [*option...*] [*input_file...*]

Where [*option...*] is one or more of the following:

```
@file
--allow-multiple-definition
--as-needed  |  --no-as-needed
-Bdynamic    |  -dy
-Bshareable  |  -shared
-Bstatic     |  -dn  |  -non_shared  |  -static
-Bsymbolic
-build-id=style
-cref
-d
--defsym symbol=expression
-discard-all
-discard-locals
--dynamic-linker=file
--dynamic-list=file
-eh-frame-hdr
--entry=entry     |  -e entry
--export-dynamic  |  -E
-extern-list=file
-fatal-warnings   |  -no-fatal-warnings
-filetype=(obj|dso|exe)
-fini=name
--force-dynamic
-fPIC
-g
--gc-sections
-gpsize size  |  -G size
-hash-size=size
-hash-style=(sysv|gnu|both
--help
-init=name
```

```
-larchive
--library=namespec          |  -lnamespec
--library-path=searchdir    |  -Lsearchdir
-m=emulation
-Map=mapfile
-MapDetail <value>
-march=version      |  -mcpu=version
-merge-strings
-mtriple=(hexagon-unknown-elf|hexagon-unknown-linux)
-mv5    |  -mv55
-mv60   |  -mv61    |  -mv62   |  -mv65   |  -mv66
-mv67   |  -mv67t   |  -mv68
--no-threads
-no-trampolines
-no-undefined
-noinhibit-exec
-nostdlib
--output=output   |  -o output
-p
--print-gc-sections
--print-map       |  -M
--relocatable     |  -r
-relocation-model=(default|static|pic|dynamic-no-pic)
-rpath-pathname
-rpath-link=pathname
--script=file     |  -T file
--section-start section=org
-soname=name      |  -h name
--start-group=archive ... --end-group   |  -( archive ... -)
--strip-all
--strip-debug     |  -S
-symdef <file>
-sysroot=pathname
-trace=symbol=<symbolname>
--trace=type      |  -t=type
--undefined=symbol   |  -u symbol
--unresolved-symbols=method
-use-memory
-version
--version-script=file
--whole-archive   |  --no-whole-archive
--wrap=symbol
-z keyword
```

Except for the command name, the linker is invoked like the GNU linker, and it supports the most commonly-used options in the GNU linker.

Arguments specified on the command line with no option switch are assumed to be object files. If the linker does not recognize these files as object files, it treats them as linker script files (Section 4.4).

NOTE: The `@file` option allows linker options and input files to be specified in a text file rather than directly on the command line.

4.2 Options

Option names can be truncated if they uniquely identify the option.

@file

Include linker command arguments from the specified text file.

--allow-multiple-definition

Allow multiple definitions of a symbol to exist in the files being linked.

Only the first definition is used; the additional ones are ignored. Typically, the linker flags multiple symbol definitions with an error.

--as-needed
--no-as-needed

These options are accepted for compatibility with the linker, but they are ignored.

-Bdynamic
-dy

Link against dynamic libraries.

This option can be used multiple times. It affects the library searching performed by subsequent occurrences of the -l option.

-Bshareable
-shared

Create a shared library.

-Bstatic
-dn
-non_shared
-static

Link against static libraries.

-Bsymbolic

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

-build-id=style

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

-cref

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

-d

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

`--defsym` *`symbol=expression`*

Create a global symbol in the output file, containing the absolute address given by `expression`. This option can be used multiple times.

The expression is limited to a hexadecimal constant or existing symbol name, with the optional addition or subtraction of a second hexadecimal constant or symbol.

`-discard-all`

Delete all local symbols from the output file.

`-discard-locals`

Delete all temporary local symbols from the output file.

`--dynamic-linker=`*`file`*

Specify a dynamic linker.

`--dynamic-list=`*`file`*

Add symbols to the dynamic symbol table of a dynamic executable.

The specified text file contains one or more lists of symbol names, with each list having the following form:

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

This option can be used multiple times.

The dynamic symbol table is used by the dynamic loader to resolve external symbols associated with shared libraries.

NOTE: All symbols in the dynamic symbol table must already be defined in the linker input object files. Otherwise, the linker will generate an error message.

`-eh-frame-hdr`

In the output file, create the ELF note section, `.note.gnu.build-id`, and the ELF segment header, `PT_GNU_EH_FRAME`.

`--entry=entry`
`-e entry`

Use the specified symbol as the program entry point.

`--export-dynamic`
`-E`

Export all dynamic symbols.

By default, the only dynamic symbols exported are those referenced by dynamic objects that are explicitly specified during linking. Dynamic symbols can be selectively exported with `--dynamic-list`.

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

`-extern-list=`*`file`*

Add symbols to the external symbol table of the output file.

The specified text file contains one or more lists of symbol names, with each list having the following form:

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

This option can be used multiple times.

The external symbol table is used to force additional object files to be linked into the program. The specified symbols are assumed to be defined in archive libraries.

Symbols in the external symbol table are *garbage collected* during linking.

If the specified symbol is already defined in one of the linker input files, the linker uses that definition and does not add the symbol to the external symbol table; the symbol and its dependencies will not be garbage-collected.

NOTE: This option is equivalent to using multiple `-u` options.

`-fatal-warnings`
`-no-fatal-warnings`

Convert warnings into fatal errors.

`-no-fatal-warnings` restores the default behavior.

`-filetype=(obj|dso|exe)`

Output file type (not all types are supported on all targets).

- `obj` Generate a relocatable object file (*.o)
- `dso` Generate a dynamic shared object file (*.so)
- `exe` Generate an executable object file (*.exe)

`-fini=`*`name`*

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

`--force-dynamic`

Force the output file to include dynamic sections.

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

`-fPIC`

Set the relocation model to PIC. This option is equivalent to `-relocation-model=pic`.

`-g`

Generate debug information.

`--gc-sections`

Delete all unused input sections from the output file (garbage collection).

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

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

`-gpsize` *`size`*
`-G` *`size`*

Specify the maximum size (in bytes) of data items allocated in the global `bss` section. The default value is 8.

`-hash-size=`*`size`*

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

`-hash-style=(sysv|gnu|both`

Specify the hash table type used in the linker.

`sysv`	Classic ELF `.hash` section
`gnu`	New-style GNU `.gnu.hash` section
`both`	Both ELF and GNU hash tables

`--help`

Display the list of linker command options.

`-init=`*`name`*

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

`-l`*`archive`*

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

`--library=`*`namespec`*
`-l`*`namespec`*

Add the archive or object file specified by `namespec` to the list of files to link. This option can be used multiple times.

If `namespec` is in the form of `:filename`, the linker searches the library path for a file called `filename`. Otherwise, it searches the library path for a file called `libnamespec.a`.

The linker can also search for files on systems that support shared libraries. The linker searches a directory for a library called `libnamespec.so` before searching for one called `libnamespec.a` (by convention, the SO extension indicates a shared library). However, this behavior does not apply to `:filename`, which always specifies a file called `filename`.

The `--Bdynamic` or `-dy` switch can be used to switch the linker to search shared libraries.

The `-Bstatic` switch can be used to switch the linker to search for archive libraries.

`--library-path=`*`searchdir`*
`-L`*`searchdir`*

Add the specified pathname to the list of paths used to search for libraries and scripts. This option can be used multiple times.

`-m=`*`emulation`*

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

`-Map=`*`mapfile`*

Generate a link map file with the specified file name.

`-MapDetail <`*`value`*`>`

Provide path conversion or string details in the map file.

The *`value`* argument can be one of the following options.

- *`absolute-path`*

  This option converts relative paths to absolute paths. When you pass the input files, they are displayed the same way they are passed.

  This option shows absolute paths of all other relative paths in the map file.

- *`show-strings`*

  If a symbol points to a string section type, that string is displayed in the map file when *`show-strings`* is enabled.

`-march=`*`version`*
`-mcpu=`*`version`*

Specify the Hexagon processor version of the linker output file. The default value is the version specified in the first input file.

For more information, see the *Qualcomm Hexagon Programmer's Reference Manual*.

NOTE: Not all of the -march and `-mcpu` option settings are supported in a specific Hexagon tools release. For more information, see Section 2.1.

`-merge-strings`

Remove duplicate instances of character strings from the output file.

`-mtriple=(hexagon-unknown-elf|hexagon-unknown-linux)`

Specify the target triple (processor, operating system, ABI) of the linker output.

`-mv5`

Equivalent to `-march hexagonv5`.

`-mv55`

Equivalent to `-march hexagonv55`.

`-mv60`

Equivalent to `-march hexagonv60`.

`-mv61`

Equivalent to `-march hexagonv61`.

`-mv62`

Equivalent to `-march hexagonv62`.

`-mv65`

Equivalent to `-march hexagonv65`.

`-mv66`

Equivalent to `-march hexagonv66`.

`-mv67`

Equivalent to `-march hexagonv67`.

`-mv67t`

Equivalent to `-march hexagonv67t` (Small Core).

`-mv68`

Equivalent to `-march hexagonv68`.

`--no-threads`

Disable threads at link time.

`-no-trampolines`

Do not add trampolines to the linked code.

`-no-undefined`

Do not allow unresolved references in the linked code.

`-noinhibit-exec`

Do not delete the output file after a linker error.

`-nostdlib`

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

`--output=`*`output`*
`-o` *`output`*

Specify the linker output file. The default name is `a.out`.

`-p`

This option has no effect. It is provided for compatibility with the Arm linker.

`--print-gc-sections`

Display all sections that were removed by garbage collection during linking.

Use this option in conjunction with `--gc-sections`.

`--print-map`
`-M`

Generate a link map file that provides information on how the object files were linked. For more information, see Section 4.3.

`--relocatable`
`-r`

Create a relocatable object. The default object is an absolute file.

`-relocation-model=(default|static|pic|dynamic-no-pic)`

Specify a relocation model.

`default`	Target default relocation model
`static`	Non-relocatable code
`pic`	Fully relocatable, position-independent code
`dynamic-no-pic`	Relocatable external references, non-relocatable code

`-rpath-`*`pathname`*

Add the specified pathname to the search path for the runtime library. For a usage example, see Section 4.2.1.1.

`-rpath-link=`*`pathname`*

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

`--script=`*`file`*
`-T` *`file`*

Specify the linker script file (Section 4.4).

`--section-start` *`section=org`*

Set the start address of the specified section. The `org` value must be a hexadecimal integer.

This option can be repeated multiple times.

`-soname=`*`name`*
`-h` *`name`*

Set the internal name of a shared library by setting `DT_SONAME`.

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

Resolve circular symbol references in the specified files and libraries.

Because the linker must repeatedly search the libraries to resolve circular references, this option significantly affects the linker performance. Use it only when circular symbol references are present.

Specify files and libraries by placing them between the `-(` and `-)` options or the `--start-group` and `--end-group` options.

`--strip-all`

Do not include any symbol information in the output file.

`--strip-debug`
`-S`

Do not include debugger symbols in the output file.

`-symdef <file>`

Specify a file that the linker emits after processing all the input files when it is ready to output the ELF file.

The symdef file contains all global symbols and their addresses. The file can later be used in a subsequent link, and the linker uses the symbols specified in the file as absolute symbols.

`-sysroot=pathname`

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

`-trace=symbol=<symbolname>`

Make the linker trace the symbol name from all input files.

`--trace=type`
`-t=type`

Display trace information that indicates how the files are being linked.

The following types of information can be displayed.

`command-line`	Display linker options specified on the command line
`file`	Display files specified for linking
`garbage-collection`	Display garbage collection performed during linking
`plugin`	Display various plug-in load/unload and execution operations
`symbols`	Display program symbols processed by the linker, and indicate how they are resolved
`sym-reloc`	Display symbol relocation performed by the linker
`trampolines`	Display details on trampolines created by the linker
`wrap-symbols`	Display symbols that rename information to support the `--wrap` option

This option can be repeated multiple times.

`--undefined=symbol`
`-u symbol`

Add the specified symbol to the external symbol table of the output file. This is typically done to force additional object files to be linked into a program.

NOTE: Symbols in the external symbol table are garbage-collected during linking.

If the specified symbol is already defined in one of the linker input files, the linker uses that definition and does not add the symbol to the external symbol table. The symbol and its dependencies will not be garbage collected.

This option can be repeated multiple times.

`--unresolved-symbols=method`

Determine how to handle unresolved symbols. There are four possible values for `method`:

`ignore-all`

Do not report any unresolved symbols.

`report-all`

Report all unresolved symbols. (Default)

`ignore-in-object-files`

Report unresolved symbols that are contained in shared libraries, but ignore them if they come from regular object files.

`ignore-in-shared-libs`

Report unresolved symbols that come from regular object files, but ignore them if they come from shared libraries.

This value can be useful when creating a dynamic binary and it is known that all the shared libraries that it should be referencing are included on the linker command line.

`-use-memory`

Use the memory buffer for linker operations.

`-version`

Display the linker release version.

`--version-script=file`

Control symbols to be exported in a dynamic symbol table.

`--whole-archive`
`--no-whole-archive`

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

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

`--wrap=`*`symbol`*

Create a wrapper for the specified symbol.

Resolve any undefined references to the symbol as references to `__wrap_symbol`, and resolve any undefined references to the symbol `__real_symbol` as references to symbol.

This option can be repeated multiple times.

`-z` *`keyword`*

Extended options for the linker, where *`keyword`* is one of the following sub-options. For usage examples, see Section 4.2.1.2.

`clade-file=`*`file`*

Set the CLADE file name.

`combreloc`

Merge dynamic relocations into one section and sort them.

`nocombreloc`

Do not merge dynamic relocations into one section.

`global`

Make symbols in a dynamic shared object available for subsequently loaded objects.

`defs`

Report unresolved symbols in object files.

`muldefs`

Allow multiple definitions.

`initfirst`

This option is ignored.

`nocopyreloc`

Do not create copy relocations.

`nodefaultlib`

This option is ignored.

`relro`

Create a RELRO program header.

`norelro`

Do not create a RELRO program header.

`lazy`

Mark an object as lazy runtime binding.

When generating an executable or shared library, this option tells the dynamic linker to defer a function call resolution to the point when the function is called (lazy binding) rather than at load time. Lazy binding is the default setting.

`now`

Mark an object as immediate runtime binding.

When generating an executable or shared library, this option tells the dynamic linker to bind references immediately.

`origin`

This option is ignored.

`text`

Treat `DT_TEXTREL` in a shared object as an error.

`execstack`

Mark an executable as requiring an executable stack.

`noexecstack`

Mark an executable as not requiring an executable stack.

`common-page-size=`*`value`*

Set the emulation common page size to a specified value.

`max-page-size=SIZE`

Set maximum page size to `SIZE`.

`nodelete`

Mark a dynamic shared object as non-deletable at runtime.

`compactdyn`

Reduce the number of entries in the dynamic table.

4.2.1 Examples

4.2.1.1 Use -rpath option

Use the `-rpath=`*`pathname`* option when linking an ELF executable with shared objects. This option provides the linker with the information that the loader requires to resolve some of the dynamic dependencies at runtime, and it instructs the linker to store that information in the dynamic section of the executable.

All `-rpath` arguments are concatenated and passed to the runtime linker, which uses them to locate shared objects at runtime.

The `foo.c`, `bar.c`, `foobar.c`, and `main.c` files are used for these examples.

In `foo.c`:

```
#include <stdio.h>

void foo(void)
{
    puts(__func__);
}
```

In `bar.c`:

```
#include <stdio.h>

void bar(void)
{
    puts(__func__);
}
```

In `foobar.c`:

```
extern void foo(void);
extern void bar(void);

void foobar(void)
{
    foo();
    bar();
}
```

In `main.c` :

```
extern void foobar(void);

int main(void)
{
    foobar();
    return 0;
}
```

Example 1:

```
hexagon-clang -c -fPIC foo.c
hexagon-link -shared -o libfoo.so foo.o

hexagon-clang -c -fPIC bar.c
hexagon-link -shared -o libbar.so bar.o

hexagon-clang -c -fPIC foobar.c
hexagon-link -shared -o libfoobar.so foobar.o -dynamic -lfoo -lbar

hexagon-readelf -d libfoobar.so
```

The output of `hexagon-readelf -d libfoobar.so` contains the following lines:

```
Dynamic section at offset 0x1000 contains 15 entries:
  Tag        Type          Name/Value
  0x00000001 (NEEDED)      Shared library: [libfoo.so]
  0x00000001 (NEEDED)      Shared library: [libbar.so]
```

Example 2:

```
hexagon-clang -c main.c
hexagon-link -o out main.o -dynamic -lfoobar -rpath=<rpath-pathname>

hexagon-readelf -d out
```

The output of `hexagon-readelf -d out` shows that the executable contains the provided `runpath`:

```
Dynamic section at offset 0x2000 contains 15 entries:
  Tag        Type          Name/Value
  0x00000001 (NEEDED)      Shared library: [libfoobar.so]
  0x0000001d (RUNPATH)     Library runpath: [<rpath-pathname>]
```

4.2.1.2 Use -z sub-options

This section provides examples of using the `-z keyword` sub-options.

The `shlib.c` test file is used for the following examples:

```
#include <assert.h>
int a = 10;
int b = 20;
int c = 30;
static int a1 = 40;
static int b1 = 50;
static int c1 = 60;
static int *d1 = &c1;
int *x;
static int *d2 = &c;
static int localfn() {
  return 0;
}

int (*hexrelativefn)() = &localfn;

int fn1() {
  printf("%d\n",b);
  printf("%d\n",b1);
  assert(b == 20);
  assert(b1 == 50);
  return 0;
}

int fn2() {
  x = &c;
  printf("%d\n",*x);
  printf("%d\n",c1);
  assert(*x == 30);
  assert(*d2 == 30);
  assert(c == 30);
  assert(c1 == 60);
  return 0;
}

int fn() {
  printf("hello world\n");
  printf("%d\n",a);
  printf("%d\n",a1);
  printf("%d\n",*d1);
  assert(a == 10);
  assert(a1 == 40);
  assert(*d1 == 60);
  (*hexrelativefn)();
  fn1();
  fn2();
  return 0;
}
```

Examples

`-z clade-file=`*`file`*

This option is used to set the CLADE file name. It only applicable for the `hexagon-link` command.

For a CLADE file named `dicts.bin`:

```
hexagon-link -z clade-file=dicts.bin 1.o 2.o -o out
```

`-z combreloc`

This option merges dynamic relocations into one section and sorts them. The relocations are sorted and placed on the top of the output of `readelf -r <shared-object>`.

Example:

```
hexagon-clang -fcommon -mv65 -O2 -mv65 -fPIC -c shlib.c -o shlib.o
hexagon-link --emit-relocs -mcpu=hexagonv65 initS.o shlib.o
finiS.o -o libshlib.so -dy -shared -z combreloc
```

The output of `hexagon-readelf -r libshlib.so` contains the following lines. The items are sorted by the `combreloc` sub-option:

```
Relocation section '.rela.dyn' at offset 0x2dc contains 8 entries:
 Offset     Info    Type             Sym. Value  Symbol's Name + Addend
0000109c  00000023 R_HEX_RELATIVE                6ac
000010ac  00000306 R_HEX_32           000004e0    __cxa_finalize_stub + 0
000010c0  00000421 R_HEX_GLOB_DAT     00000000    __cxa_finalize + 0
000010c8  00000a21 R_HEX_GLOB_DAT     0000109c    hexrelativefn + 0
000010c4  00000c21 R_HEX_GLOB_DAT     00001090    a + 0
000010b4  00000d21 R_HEX_GLOB_DAT     00001094    b + 0
000010b8  00000e21 R_HEX_GLOB_DAT     00001098    c + 0
000010bc  00001121 R_HEX_GLOB_DAT     00001100    x + 0
```

`-z nocombreloc`

This option prevents dynamic relocations from being merged into one section. The dynamic relocations are not sorted in the output of `readelf -r <shared-object>`.

Example:

```
hexagon-clang -fcommon -mv65 -O2 -mv65 -fPIC -c shlib.c -o shlib.o
hexagon-link --emit-relocs -mcpu=hexagonv65 initS.o shlib.o
finiS.o -o libshlib.so -dy -shared -z nocombreloc
```

The output of `hexagon-readelf -r libshlib.so` contains the following lines. The items are not sorted due to use of the `nocombreloc` sub-option:

```
Relocation section '.rela.dyn' at offset 0x2dc contains 8 entries:
 Offset     Info    Type             Sym. Value  Symbol's Name + Addend
000010ac  00000306 R_HEX_32           000004e0    __cxa_finalize_stub + 0
000010b4  00000d21 R_HEX_GLOB_DAT     00001094    b + 0
000010b8  00000421 R_HEX_GLOB_DAT     00000000    __cxa_finalize + 0
000010bc  00000e21 R_HEX_GLOB_DAT     00001098    c + 0
000010c0  00001121 R_HEX_GLOB_DAT     00001100    x + 0
000010c4  00000c21 R_HEX_GLOB_DAT     00001090    a + 0
000010c8  00000a21 R_HEX_GLOB_DAT     0000109c    hexrelativefn + 0
0000109c  00000023 R_HEX_RELATIVE                6ac
```

`-z global`

This option makes symbols in the dynamic shared object available for subsequently loaded objects.

Example:

```
hexagon-clang -fcommon -mv65 -O2 -mv65 -fPIC -c shlib.c -o shlib.o
hexagon-link --emit-relocs -mcpu=hexagonv65 initS.o shlib.o
finiS.o -o libshlib.so -dy -shared -z global
```

The output of `readelf -d libshlib.so` has the following line:

```
0x6ffffffb (FLAGS_1) GLOBAL.
```

`-z defs`

This option reports unresolved symbols in object files.

Example in `defs.c`:

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

hexagon-clang -fcommon -mv65 -O2 -mv65 -fPIC -c defs.c -o defs.o
hexagon-link -shared defs.o -o defs.so
```

The link in the example produces *not error.*

```
hexagon-link -shared defs.o -o defs.so -z defs
Error: defs.o(.text+0x0): undefined reference to `bar'
Fatal: Linking had errors.
```

`-z muldefs`

This option allows for multiple definitions, thus suppressing multiple definition errors.

Example in `muldef.c`:

```
int foo = 0;
int main() { return foo; }

muldefCopy.c:
int foo = 0;
int main() { return foo; }

hexagon-clang -c muldef.c
hexagon-clang -c muldefCopy.c

hexagon-link muldef.o muldefCopy.o
Error: multiple definition of symbol `foo' in file muldef.o and
muldefCopy.o
Error: multiple definition of symbol `main' in file muldef.o and
muldefCopy.o
Fatal: Linking had errors.
```

Use `-z muldefs` in the link to suppress the errors:

```
hexagon-link muldef.o muldefCopy.o -z muldefs
<no-error>
```

`-z nocopyreloc`

This option tells the linker to not create copy relocations. The following example includes four test files.

In `1.c`:

```
extern int foo;
int bar() {
  return foo;
}
```

In `2.c`:

```
extern int foo;
int car() {
  return foo;
}
```

In `3.c`:

```
extern int foo;
int baz() { return foo; }
```

In `4.c`:

```
int foo = 10;
```

The following link line:

```
hexagon-link 1.o 2.o 3.o -Bdynamic lib4.so -o out -z nocopyreloc
```

Results in the following error:

```
"Cannot copy symbol foo"
```

`-z relro`

This option creates a RELRO program header.

Example in `relro.c`:

```
__attribute__((constructor)) int cfoo() { return 0; }
__attribute__((destructor)) int dbar() { return 0; }
int val = 10;
// Access val from GOT.
__attribute__((section(".text.foo"))) int foo() { return val; }
__attribute__((section(".text.bar"))) int bar() { return foo(); }

hexagon-clang -c relro.c -fPIC -fno-use-init-array
hexagon-link relro.o -shared -o relro.so -z relro -z now
```

The output of `readelf -l -W relro.so` has the following item in the program headers:

```
GNU_RELRO   0x001000 0x00001000 0x00001000 0x000b8 0x000b8 RW  0x8
```

`-z norelro`

This options prevents a RELRO program header from being created.

Example in `relro.c`:

```
__attribute__((constructor)) int cfoo() { return 0; }
__attribute__((destructor)) int dbar() { return 0; }
int val = 10;
```

```
// Access val from GOT.
__attribute__((section(".text.foo"))) int foo() { return val; }
__attribute__((section(".text.bar"))) int bar() { return foo(); }

hexagon-clang -c relro.c -fPIC -fno-use-init-array
hexagon-link relro.o -shared -o relro.so -z norelro -z now
```

The output of `readelf -l -W relro.so` does not have a header with `GNU_RELRO`.

`-z lazy`

This option marks an object lazy runtime binding.

Example in `relro.c`:

```
__attribute__((constructor)) int cfoo() { return 0; }
__attribute__((destructor)) int dbar() { return 0; }
int val = 10;
// Access val from GOT.
__attribute__((section(".text.foo"))) int foo() { return val; }
__attribute__((section(".text.bar"))) int bar() { return foo(); }

hexagon-clang -c relro.c -fPIC -fno-use-init-array
hexagon-link relro.o -shared -o relro.so -z relro -z lazy
```

The output of `readelf -l -W relro.so` has the following lines:

```
Program Headers:
  Type         Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg  Align
  LOAD         0x000000 0x00000000 0x00000000 0x0025c 0x0025c R E  0x1000
  LOAD         0x001000 0x00001000 0x00001000 0x010b0 0x010b0 RW   0x1000
  DYNAMIC      0x001008 0x00001008 0x00001008 0x00080 0x00080 RW   0x4
  GNU_RELRO    0x001000 0x00001000 0x00001000 0x0008c 0x0008c RW   0x4
```

`-z now`

This option marks an object non-lazy runtime binding.

Example in `relro.c`:

```
__attribute__((constructor)) int cfoo() { return 0; }
__attribute__((destructor)) int dbar() { return 0; }
int val = 10;
// Access val from GOT.
__attribute__((section(".text.foo"))) int foo() { return val; }
__attribute__((section(".text.bar"))) int bar() { return foo(); }

hexagon-clang -c relro.c -fPIC -fno-use-init-array
hexagon-link relro.o -shared -o relro.so -z relro -z now
```

The output of `readelf -l -W relro.so` has the following lines:

```
Program Headers:
 Type          Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg  Align
 LOAD          0x000000 0x00000000 0x00000000 0x0022c 0x0022c R E  0x1000
 LOAD          0x001000 0x00001000 0x00001000 0x010bc 0x010bc RW   0x1000
 DYNAMIC       0x001008 0x00001008 0x00001008 0x00090 0x00090 RW   0x4
 GNU_RELRO     0x001000 0x00001000 0x00001000 0x000b8 0x000b8 RW   0x8
```

`-z text`

This option treats `DT_TEXTREL` in a shared object as an error.

`-z execstack`

This option marks an executable as requiring an executable stack.

Example in `1.c`:

```
int main()
{
   return 0;

hexagon-clang -target aarch64 -fPIC 1.c -c -o 1.o
hexagon-link -march aarch64 -Bshareable 1.o -o 1.so -z execstack
```

The output of `readelf -a -W 1.so` has the following item that includes Execute (`E`):

```
GNU_STACK      0x000000 0x0000000000000000 0x0000000000000000
0x000000 0x000000 RWE 0
```

`-z noexecstack`

This option marks an executable as not requiring an executable stack.

Example in `1.c`:

```
int main()
{
   return 0;
}
hexagon-clang -target aarch64 -fPIC 1.c -c -o 1.o
hexagon-link -march aarch64 -Bshareable 1.o -o 1.so -z noexecstack
```

The output of `readelf -a -W 1.so` has the following item that does not include `E`:

```
GNU_STACK      0x000000 0x0000000000000000 0x0000000000000000
0x000000 0x000000 RW  0
```

`-z common-page-size=`*`value`*

This option sets the emulation common page size to the specified value. This example uses two files.

In `max.s`:

```
.section ".data", "wa"
.long 0x12345678
```

In `max.t`:

```
SECTIONS
{
  . = 0x4017dc;
  . = DATA_SEGMENT_ALIGN (CONSTANT (MAXPAGESIZE), CONSTANT
(COMMONPAGESIZE));
  .data : { *(.data) }
  . = DATA_SEGMENT_END (.);
}
```

```
hexagon-clang -c max.s
hexagon-link max.o -T max.t -z max-page-size=0x2000 -z common-page-
size= 0x1000
```

The output of `readelf -S -W a.out` has entries like the following:

```
.data         PROGBITS       00403000 001000 000004 00  WA  0   0  1
.comment      PROGBITS       00000000 001004 00002c 00  MS  0   0  1
```

`-z max-page-size=`*`SIZE`*

This option sets the maximum page size to `SIZE`. This example uses two files.

In `max.s`:

```
.section ".data", "wa"
.long 0x12345678
```

In `max.t`:

```
SECTIONS
{
  . = 0x4017dc;
  . = DATA_SEGMENT_ALIGN (CONSTANT (MAXPAGESIZE), CONSTANT
(COMMONPAGESIZE));
  .data : { *(.data) }
  . = DATA_SEGMENT_END (.);
}
```

```
hexagon-clang -c max.s
hexagon-link max.o -T max.t -z max-page-size=0x500 -z common-page-
size= 0x1000
```

The output of `readelf -S -W a.out` has lines that show the maximum page size value that overrides the common page size:

```
.data          PROGBITS     00401800 000500 000004 00  WA  0   0  1
.comment       PROGBITS     00000000 000504 00002c 00  MS  0   0  1
```

`-z nodelete`

This option marks a dynamic shared object as non-deletable at runtime.

Example in `nodelete.c`:

```
int foo() {
  return 0;
}
hexagon-clang nodelete.c -c -fPIC
hexagon-link -shared nodelete.o -z nodelete -z now --enable-new-
dtags -o nodelete.so
```

The output of `readelf -d nodelete.so` has the following line:

```
0x6ffffffb (FLAGS_1)                    Flags: NOW NODELETE
```

`-z compactdyn`

This option reduces the number of entries in the dynamic table.

For example, `PLTGOT` and `DEBUG` are not added if this option is present in the command line.

4.3 Link maps

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

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

A link map is generated as a text file. The file can be specified on the command line with either the `-Map` or `--print-map` option. For more information, see Section 4.2.

Link maps can have two output formats:

- Conventional GNU map file format
- YAML map file format

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

Table 4-1 Link map sections

Section	Section entry format
Archive	`archive_file (symbol_define_object_file)` `  symbol_reference_object_file (symbol)`
Common symbols	`symbol size archive_file (symbol_define_object_file)`
Linker script	`linker_script_command`

Memory map	`output_section addr size`
	`  input_section addr size object_file`
	`  symbol addr`
	`  ...`
	`  ...`

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

4.3.1 Example link map file

The following code example shows a typical link map file.

NOTE: The map file has been shortened for readability.

```
Archive member included because of file (symbol)
/pkg/qct/software/hexagon/releases/tools/7.2/Tools/bin/../target/hexagon/lib/v60/
libstandalone.a (low_thread.o)
/pkg/qct/software/hexagon/releases/tools/7.2/Tools/bin/../target/hexagon/lib/v60/
crt0_standalone.o (thread_stop)
/pkg/qct/software/hexagon/releases/tools/7.2/Tools/bin/../target/hexagon/lib/v60/
libstandalone.a (sys_get_cmdline.o)
/pkg/qct/software/hexagon/releases/tools/7.2/Tools/bin/../target/hexagon/lib/v60/
crt0.o (sys_get_cmdline)

...

Allocating common symbols
Common symbol     size    file

__eh_nodes        0x4     /pkg/qct/software/hexagon/releases/tools/7.2/Tools/bin/
                           target/hexagon/lib/v60/libc.a (xregister.o)

...

Linker Script and memory map
START GROUP
END GROUP
LOAD /pkg/qct/software/hexagon/releases/tools/7.2/Tools/bin/../target/hexagon/lib/
v60/crt0_standalone.o[v60]
LOAD /pkg/qct/software/hexagon/releases/tools/7.2/Tools/bin/../target/hexagon/lib/
v60/crt0.o[v60]
LOAD /pkg/qct/software/hexagon/releases/tools/7.2/Tools/bin/../target/hexagon/lib/
v60/init.o[v60]

...

              0x0    0x0

.start        0x0    0x3dc0
(.start)
.start        0x0    0x3b88   /pkg/qct/software/hexagon/releases/tools/7.2/Tools/
                               bin/../target/hexagon/lib/v60/crt0_standalone.o
    0x5b8         .CheckWB
    0x724         .FinishOverrides
    0x104         .Init

...

.start        0x3ba0 0x220    /pkg/qct/software/hexagon/releases/tools/7.2/Tools/
                               bin/../target/hexagon/lib/v60/crt0.o
    0x3d34         .PreArgs
    0x3ce0         .PreBSS
    0x3d34         .PreBSSSkip
    0x3c78         .SkipUpdate
```

```
    0x3ba0          .start
    0x0             _start
    0x3bb8          hexagon_pre_main
    0x3d34          hexagon_start_main
    0x3bb8          qdsp6_pre_main
    0x3d34          qdsp6_start_main
    0x0             start

.init    0x4000          0x54
(.init)
.init    0x4000          0x54    /pkg/qct/software/hexagon/releases/tools/7.2/Tools/
                                  bin/../target/hexagon/lib/v60/init.o
         0x4000          .init
         0x4000          _init

.text    0x5000         0x20f0

...
```

4.3.2 Archive section

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

Each entry in the archive section contains the following items:

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

In the following example, the `thread_stop` symbol is referenced in `crt0_standalone.o` and defined in `low_thread.o`, which in turn is stored in the `libstandalone.a` archive file:

```
/pkg/qct/software/hexagon/releases/tools/7.2/Tools/bin/../target/
hexagon/lib/v60/libstandalone.a (low_thread.o)
/pkg/qct/software/hexagon/releases/tools/7.2/Tools/bin/../target/
hexagon/lib/v60/crt0_standalone.o (thread_stop)
```

4.3.3 Common symbols section

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

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

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

In the following example, the common symbol `__eh_nodes` has size 0x4 and is defined in `xregister.o`, which in turn is stored in the `libc.a` archive file:

```
__eh_nodes          0x4
/pkg/qct/software/hexagon/releases/tools/7.2/Tools/bin/target/hexagon/
lib/v60/libc.a (xregister.o)
```

4.3.4 Linker script section

The linker script section of a link map lists the complete linker script that was specified for the link. For more information on linker scripts see Section 4.4.

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

The linker script section also has all the linker scripts used, listed in a recursive manner.

In the following example, the `a.t` script is a linker script passed to the linker. It has an `INCLUDE` command to include `ld.t`:

```
Linker scripts used (including INCLUDE command)
/tmp/a.t
        ld.t
```

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

```
START GROUP
END GROUP
LOAD /pkg/qct/software/hexagon/releases/tools/7.2/Tools/bin/../target/
hexagon/lib/v60/crt0_standalone.o[v60]
LOAD /pkg/qct/software/hexagon/releases/tools/7.2/Tools/bin/../target/
hexagon/lib/v60/crt0.o[v60]
LOAD /pkg/qct/software/hexagon/releases/tools/7.2/Tools/bin/../target/
hexagon/lib/v60/init.o[v60]
...
```

4.3.5 Memory map section

The memory map section of a link map lists how symbols and assembly language sections (Section 3.4) are assigned to memory in the output file.

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

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

In the following example, the output section `.start` has the start address 0x0 and size 0x3dc0. Two input sections (also named `.start`) are mapped to this output section: the first has start address 0x0 and size 0x3b88, and the second has start address 0x3ba0 and size 0x220. Following each section descriptor is a list of the symbols in the section:

```
.start      0x0   0x3dc0
(.start)
.start      0x0   0x3b88
/pkg/qct/software/hexagon/releases/tools/7.2/Tools/bin/../target/
hexagon/lib/v60/crt0_standalone.o
    0x5b8      .CheckWB
    0x724      .FinishOverrides
    0x104      .Init
    ...

.start      0x3ba0   0x220
/pkg/qct/software/hexagon/releases/tools/7.2/Tools/bin/../target/
hexagon/lib/v60/crt0.o
    0x3d34      .PreArgs
    0x3ce0      .PreBSS
```

4.3.6 Changes to GNU format maps

4.3.6.1 LTO-related changes

With the incorporation of the LTO feature in the tools, the Map has two main sections:

- Pre-LTO map section
- Post-LTO map section

Pre-LTO maps only have an archive section and a linker script section. For example:

```
Pre-LTO map records:
Archive member included because of file (symbol)
/prj/dsp/qdsp6/release/internal/branch-8.0/linux64/latest/Tools/bin/../
target/hexagon/lib/v60/G0/libstandalone.a(pte.o)
/prj/dsp/qdsp6/release/internal/branch-8.0/linux64/latest/Tools/bin/../
target/hexagon/lib/v60/G0/crt0_standalone.o (UPTE_START)
/prj/dsp/qdsp6/release/internal/branch-8.0/linux64/latest/Tools/bin/../
target/hexagon/lib/v60/G0/libstandalone.a(low_thread.o)
/prj/dsp/qdsp6/release/internal/branch-8.0/linux64/latest/Tools/bin/../
target/hexagon/lib/v60/G0/crt0_standalone.o (thread_stop)
/prj/dsp/qdsp6/release/internal/branch-8.0/linux64/latest/Tools/bin/../
target/hexagon/lib/v60/G0/libstandalone.a(sys_get_cmdline.o)
```

The linker script and memory map sections for the pre-LTO section have the file type marked with the files loaded. For an ELF file type, the architecture version is printed, making it easier to inspect. For example:

```
Linker Script and memory map:
LOAD /prj/dsp/qdsp6/release/internal/branch-8.0/linux64/latest/Tools/
bin/../target/hexagon/lib/v60/G0/crt0_standalone.o[v60]
LOAD /prj/dsp/qdsp6/release/internal/branch-8.0/linux64/latest/Tools/
bin/../target/hexagon/lib/v60/G0/crt0.o[v60]
LOAD /prj/dsp/qdsp6/release/internal/branch-8.0/linux64/latest/Tools/
bin/../target/hexagon/lib/v60/G0/init.o[v60]
LOAD /tmp/a-323fe1.o [Bitcode]
```

The sections in the post-LTO section are similar to the old map file with one exception. Files marked as SKIPPED in the map files are not processed again by the linker since they have been processed before invoking the LTO code generator. For example:

```
Linker Script and memory map:
LOAD /tmp/lto-llvm-0-324c9e.o[v60]
SKIPPED /tmp/a.t (ELF)
SKIPPED /prj/dsp/qdsp6/release/internal/branch-8.0/linux64/latest/Tools/
bin/../target/hexagon/lib/v60/G0/crt0_standalone.o (ELF)
SKIPPED /prj/dsp/qdsp6/release/internal/branch-8.0/linux64/latest/Tools/
bin/../target/hexagon/lib/v60/G0/crt0.o (ELF)
```

4.3.6.2 Miscellaneous information in map file

Additional information is not printed in the GNU format map file.

Following are examples of useful tool chain information and inferred settings for the linker from command line options:

```
# Linker from QuIC LLVM Hexagon Clang version: hexagon-clang-80-9530
# Linker based on LLVM version: 3.9
# CPU Architecture Version: hexagonv60
# Target triple environment for the link: unknown
# Maximum GP size: 0
# Link type: Static
# LTO Flag: Enabled
# LTO Options: codegen
# ABI Page Size: 0x1000
```

The memory map section now has extra information for output and input sections.

Output sections display their offset and LMA with the VMA:

```
.text   0x5000  0x60cc # Offset: 0x5000, LMA: 0x5000
```

Input sections display the section permissions and section alignment with the input file from which the section came:

```
.text   0x50c0  0x10    /tmp/lto-llvm-0-324c9e.o
#SHT_PROGBITS,SHF_ALLOC|SHF_EXECINSTR,16
```

4.3.6.3 YAML format for map files

YAML maps have information that is a hybrid of the GNU maps and are structured for easier parsing.

```
Header:
Architecture:    Hexagon
Emulation:       v60
AddressSize:     32bit
VersionInformation:
CompilerVersion: hexagon-clang-80-9530
LinkerVersion:   3.9
```

The `ArchiveRecords` list is similar to GNU maps in that it has a list of archive members pulled and the referring object that caused this inclusion of the member.

```
ArchiveRecords:
  - Origin:          '/prj/dsp/qdsp6/release/internal/branch-8.0/linux64/
latest/ Tools/bin/../target/hexagon/lib/v60/G0/libstandalone.a(pte.o)'
    Referred:        '/prj/dsp/qdsp6/release/internal/branch-8.0/linux64/
latest/Tools/bin/../target/hexagon/lib/v60/G0/crt0_standalone.o
(UPTE_START)'
  - Origin:          '/prj/dsp/qdsp6/release/internal/branch-8.0/linux64/
latest/Tools/bin/../target/hexagon/lib/v60/G0/libstandalone.a(low_thread.o)'
    Referred:        '/prj/dsp/qdsp6/release/internal/branch-8.0/linux64/
latest/Tools/bin/../target/hexagon/lib/v60/G0/crt0_standalone.o
(thread_stop)'
  - Origin:          '/prj/dsp/qdsp6/release/internal/branch-8.0/linux64/
latest/Tools/bin/../target/hexagon/lib/v60/G0/libstandalone.a(sys_get_cmdlin
e)'
```

The `Commons` list is the list of common symbols that are allocated.

```
Commons:
  - Name:            __eh_nodes
    Size:            0x00000004
    InputPath:       /prj/dsp/qdsp6/release/internal/branch-
8.0/linux64/latest/Tools/bin/../target/hexagon/lib/v60/G0/libc.a
    InputName:       xregister.o
```

The `Inputs` list shows the inputs that are loaded for this link instance.

```
Inputs:
  - 'LOAD /prj/dsp/qdsp6/release/internal/branch-8.0/linux64/latest/
Tools/bin/../target/hexagon/lib/v60/G0/crt0_standalone.o[v60]'
  - 'LOAD /prj/dsp/qdsp6/release/internal/branch-8.0/linux64/latest/
Tools/bin/../target/hexagon/lib/v60/G0/crt0.o[v60]'
  - 'LOAD /prj/dsp/qdsp6/release/internal/branch-8.0/linux64/latest/
Tools/bin/../target/hexagon/lib/v60/G0/init.o[v60]'
  - 'LOAD /tmp/a-835971.o[v60]'
```

The image description in YAML is more symbol-centric. The description of the output sections is a structure that contains the name, permissions, type, and memory information. The contents of the sections are then listed as structures describing the name of the symbol, type, size, and value.

Example

```
- Name:            .start
  Type:            [ SHT_PROGBITS ]
  Permissions:     [ SHF_ALLOC, SHF_WRITE, SHF_EXECINSTR ]
  Address:         0x0000000000000000
  Offset:          0x0000000000001000
  Size:            0x0000000000003EC0

  Contents:
    - Name:            .start
      Type:            [ SHT_PROGBITS ]
      Permissions:     [ SHF_ALLOC, SHF_WRITE, SHF_EXECINSTR ]
      LinkerScript:    'KEEP (*(.start ))'
      Offset:          0x0000000000000000
      Size:            0x0000000000001C88
      Origin:          /prj/dsp/qdsp6/release/internal/branch-8.0/linux64/
latest/Tools/bin/../target/hexagon/lib/v60/G0/crt0_standalone.o
      Alignment:       64
      Symbols:
        - Symbol:          _start
          Type:            [ STT_FUNC ]
          Scope:           0
          Size:            0x00000018
          Value:           0x0000000000000000
        - Symbol:          .start
          Type:            [ STT_OBJECT, STT_FUNC, STT_SECT ]
          Scope:           2
          Size:            0x00000000
          Value:           0x0000000000000000
        - Symbol:          start
```

```
  Type:            [ STT_FUNC ]
  Scope:           0
  Size:            0x00000018
  Value:           0x0000000000000000
- Symbol:          R0_mailbox
  Type:            [  ]
  Scope:           2
  Size:            0x00000000
  Value:           0x0000000000000020
- Symbol:          zebu_mailboxes
  Type:            [  ]
  Scope:           2
  Size:            0x00000000
  Value:           0x0000000000000020
- Symbol:          R1_mailbox
  Type:            [  ]
  Scope:           2
  Size:            0x00000000
  Value:           0x0000000000000024
```

4.4 Linker scripts

Linker scripts are used to provide a detailed specification of how the files are to be linked. The scripts offer greater control over linking than is available using just the linker command options (Section 4.2).

Linker scripts are optional. In most cases, the default behavior of the linker is sufficient (with input sections merged according to their section names, and segments ordered by sections that share similar permissions).

Linker scripts control the following properties:

- Output file format
- Program entry point
- Library search paths
- Section memory placement
- Section removal
- Section runtime properties

A linker script consists of a sequence of commands stored in a text file (the commands are described in Section 4.4.4). The script file can be specified on the command line either with the `-T` option or by specifying the file as one of the input files. The linker can distinguish between script files and object files, and it handles each file accordingly.

Table 4-2 Commands supported in linker script files

Command	Description
`ALIGN`	Align location counter to specified boundary
`ASSERT`	Linker script assertion
`CONSTRUCTORS`	Accepted for compatibility, but ignored
`ENTRY`	Program execution entry point
`EXCLUDE_FILE`	Exclude file from input section list
`FILL`	Output section fill
`INCLUDE`	Include linker script file
`KEEP`	Prevent input section from being garbage collected
`OUTPUT_ARCH`	Target processor
`OUTPUT_FORMAT`	Output properties (file format, endianness)
`PHDRS`	Object file program header
`PROVIDE`	Conditional symbol definition
`SEARCH_DIR`	Library search path
`SECTIONS`	Section mapping and memory placement
`SEGMENT_START`	Return base address of output section

NOTE: Linker scripts are like the scripts used with the GNU linker. However, no compatibility is guaranteed; only the features and syntax described in this document are supported.

4.4.1 Script example

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

```
OUTPUT_FORMAT (elf32-littlehexagon)
OUTPUT_ARCH (hexagon)
ENTRY (main)
SEARCH_DIR("./")

SECTIONS
{
  .text.mcldfn (0x2000) : { *(.text.mcldfn*) }
  . = ALIGN(0x1000);

  PROVIDE(__etext = .);
  __text_start = . + 0x1000 - 0x1000;
  .text : { *(.text.*) }


  .data : { *(.data.*) }

  . = SEGMENT_START(".bss", 0x80000);
  .bss : { *(.bss.*) }

  .init : { KEEP (*(.init)) }
  __bss_start = .;
  __bss_end = .;

}
```

The script contains several commands that specify various link properties, and a SECTIONS command that specifies two things:

- How input sections are mapped to output sections
- Where output sections are located in memory

Wild-card characters in the SECTIONS command indicate that multiple input sections are mapped to a single output section.

A period (.) is used to indicate the current location counter. However, be aware that it is assigned several different values in the SECTIONS command.

`.text.mcldfn`

In the example script file, all input sections whose names begin with `.text.mcldfn` are mapped to an output section named `.text.mcldfn`, and the merged output section is located at the virtual memory address 0x2000.

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

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

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

`.text`

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

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

`.data`

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

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

`.bss`

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

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

`.init`

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

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

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

4.4.2 Script syntax

Following is the basic syntax for link scripts.

`Symbols`

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

`Comments`

Comments can appear in linker scripts:

```
/* comment */
```

`Strings`

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

```
OUTPUT_FORMAT ("elf32-littlehexagon")
OUTPUT_ARCH (hexagon)
```

`Expressions`

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

`Location counter`

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

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

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

`Symbol assignment`

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

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

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

4.4.3 Input sections

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 wild-card characters (*, ?, and so on.) 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

Table 4-3 Examples of input section specifications

Specification	Description
`modem/lte/lt.lib:lt_init.o(.text.*)`	Specify one or more `.text.` sections from a specific object file (`lt_init.o`) in a specific archive (`lt.lib`)
`modem/lte/lt.lib:(.text.*)`	Specify one or more `.text.` sections from any object file in the specified archive (`lt.lib`)
`modem/lte/*:(.text.*)`	Specify one or more `.text.` sections from any archive in the specified directory (`modem/lte`)

Table 4-3 Examples of input section specifications (cont.)

Specification	Description
`*(.text.*)`	Specify one or more `.text.` sections from any archive or object file in the entire file system
`*lt_init.o:(.text.*)`	Specify no sections at all (because an object file is specified where an archive should be, this is an invalid specification)

4.4.4 Script commands

`ALIGN (value)`

Return the value of the location counter after it has been aligned to a specified address boundary.

This directive is used in expressions. It increases the current location counter value until it is an integral multiple of the specified numeric value, and returns the increased value.

The actual location counter is not affected.

`ASSERT(expression, string)`

Add an assertion to the linker script.

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

`CONSTRUCTORS`

Script command that is accepted for compatibility, but it is ignored.

`ENTRY (symbol)`

Specify the program execution entry point, which is the first instruction that is executed after a program is loaded.

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

`EXCLUDE_FILE (file ...)`

Reduce the number of items matched by a wild card in an input section (`SECTIONS`). Excluded items can be files, archives, or archive members.

In a section-mapping statement, wild-card characters can be used to specify multiple files or archives. For example:

```
*(.text)
```

The `EXCLUDE_FILE` command can be specified in an input section as if it were a file name, but its effect is to exclude all specified items from the input section. For example, the following input section description includes all files ending with `.text` except for `myfile.text`:

```
*(EXCLUDE_FILE(myfile.text) .text .data)
```

NOTE: If an exclusion is used in a list of section names, it applies only to the immediately following section name in the list (`.text` in the example). For several examples of using exclusions, see Section 4.5.

`FILL (value)`

Assign the specified value to any memory locations in an output section that were not assigned values during linking. Only the memory locations following the command are affected.

`FILL` can be specified multiple times to assign different fill values to different parts of the section.

NOTE: This command can be used only in an input section description (`SECTIONS`).

`INCLUDE (file)`

Include the contents of the specified text file at the current location in the linker script.

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

Include files can be nested.

`KEEP (input_sections)`

Specify input sections that are never to be removed from memory if garbage collection is enabled.

This command is used in section-mapping statements. The input section specification is wrapped as an argument of the `KEEP` directive. For example, `KEEP (*(.init))`.

Garbage control of sections is controlled with the `--gc-sections` option.

`OUTPUT_ARCH ("hexagon")`

Specify the target processor architecture.

This command accepts only the `hexagon` parameter because the target processor is always the Hexagon processor.

`OUTPUT_FORMAT ("elf32-littlehexagon")`

Specify the output file properties.

This command accepts only the `elf32-littlehexagon` parameter because the output file format is always 32-bit ELF format, and the Hexagon processor byte order is always little-endian.

`PHDRS`

Set information in the program header of an ELF output file.

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

Where:

- `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

NOTE: No loadable section should be set to `PT_NULL`.

- FILEHDR and PHDRS are not supported; they are parsed but otherwise ignored by the linker.
- FLAGS sets the p_flags field in the program header. Supported values:
  - PF_X=1 (Execute)
  - PF_W=2 (Write)
  - PF_R=4 (Read)
  - PF_MASKOS=0x0FF00000 (Bit masking for operating system-specific semantics)
  - PF_MASKPROC=0xF0000000 (Bit masking for processor-specific semantics)

  The value of FLAGS must be an integer. For example, FLAGS(5) sets p_flags to PF_R | PF_X, and FLAGS(0x03000000) sets the operating system-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 flags using a bitwise or.

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

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

The PHDRS command overrides the default program header settings of the linker.

PROVIDE (*symbol = expression*);

Conditionally define a symbol in the linker script.

The symbol is defined with the specified value only if any unresolved references exist for the symbol during linking.

This command must be terminated with a semicolon (;).

SEARCH_DIR (*path*)

Add 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.

SECTIONS

Tell the linker how to map input sections to output sections, and where output sections are located in memory.

```
SECTIONS {

   sections-command
   sections-command
   sections-command
   sections-command

}
```

The sections-command can be one of the following:

- An ENTRY command
- A symbol assignment
- An output section description

The command contains one or more section-commands, and each can be either a section mapping statement or a symbol assignment statement:

- A section-mapping statement specifies one or more input sections in one or more library files (.text.mcldfn), and maps those sections to the specified output section. It can also use the optional addr attribute to specify the virtual memory address of the output section.
- Symbol assignment statements are used to set the location counter (Section 4.4.2). The counter serves as the default address in subsequent section-mapping statements that do not explicitly specify an address.

In a section-mapping statement, wild-card characters can be used to specify multiple library files or input sections. For example, {*(.text.*)}.

Use the -M linker command line option to view the section mapping produced by a linker script.

NOTE: The SECTIONS command must appear once and only once in a linker script; all the other script commands are optional.

Output section description

Each output section command can be one of the following:

- A symbol assignment
- An input section description

The output section has the following format:

```
outputsectionname (overridetype [,permissions]) [address]
[expression] [plugin_type] [(type)] : [AT(lma)]
[ALIGN(section_alignment)] [SUBALIGN(subsection_alignment)]
[constraint]
{
    output-section-command
    output-section-command
    …
} [>region] [AT>lma_region] [:phdr :phdr ...] [=fill]
```

Where:

- outputsectionname specifies the name of the output section. The output section name, /DISCARD/, is special. It is used as a way to tell the linker to discard specific input sections.
- overridetype overrides the output section type by using overrides.

  Only the PROGBITS value is supported. When PROGBITS is used, the linker overrides the type default setting to PROGBITS.

For Hexagon environments, the following types are supported to allow the section to be compressed using the CLADE plug-in:

- COMPRESS
- COMPRESS_RW
- COMPRESS_RO

- permissions overrides the permissions of the output section. Supported values:
  - RW
  - RWX
  - RX
  - R
- address is an expression for specifying the virtual memory address of the output section. This field is optional. The linker sets the virtual address to the current value of the location counter
- expression specifies that an address can be an arbitrary expression.
- plugin_type specifies a plug-in to be run on that output section. Supported types:
  - PLUGIN_CONTROL_FILESZ
  - PLUGIN_CONTROL_MEMSZ
  - PLUGIN_ITER_CHUNKS
- type specifies the type for an output section.

  Each output section can have a type. Only the NOLOAD type is supported with the Qualcomm Linker (QCLD).
- AT(lma) specifies a physical address for a section.

  Every section has a virtual address (VMA) and a physical address (LMA). The linker typically sets the LMA to be equal to the VMA. You can change that by using the AT keyword. The lma expression that follows the AT keyword specifies the physical address of the section.
- ALIGN(section_alignment) aligns the output section start address. The start address is increased until it is an integral multiple of the expression specified in section_align.
- SUBALIGN(subsection_alignment) aligns the start address of an input section within an output section. The input section start address is increased until it is an integral multiple of the expression specified in subsection_align.
- constraint specifies the output section constraint. Supported values:
  - ONLY_IF_RO – Create the output section only if all of its input sections are read only.
  - ONLY_IF_RW – Create the output section only if all of its input sections are read-write.
- region and AT>lma_region are currently not supported.

- `:phdr` assigns the section to a previously-defined program segment.

  If a section is assigned to one or more segments, all subsequently allocated sections are assigned to those segments as well, unless they explicitly specify a `:phdr` attribute.

- `=fill` assigns the value of the specified expression to all unspecified memory areas of the section.

`SEGMENT_START (`*`section`*`, `*`default`*`)`

Return the base address of the specified output section. If the section is not defined, the specified address value is returned instead.

This command is used in expressions. The section is specified as a character string, and the default value as an address value. For example, `SEGMENT_START(".data", 0x1000)`.

4.4.5 Examples of linker scripts

This section presents several example linker scripts that demonstrate the following:

- How to use exclusions to specify input files for linking
- How to build static and dynamic executables

4.4.5.1 Exclude file in archive

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

NOTE: Any sections whose names match the exclusion are still included in the link, unless they are stored in the excluded archive.

The following example uses four relocatable files: `a1.o`, `a2.o`, `a3.o`, `a4.o`. Each file contains functions named `foo_N` and `bar_N`, where `N` indicates the digit in the file name. The linker script excludes `foo_2` but not `bar_2` from `.text1` (because `EXCLUDE_FILE` applies only to the immediately following section name):

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

hexagon-clang -ffunction-sections -c a1.c a2.c a3.c a4.c -mv5
hexagon-ar cr lib23.a a2.o a3.o
hexagon-ar cr lib4.a a4.o
hexagon-link -T script.t -o mcld.out a1.o --whole-archive lib23.a lib4.a
--no-whole-archive
```

Section headers (starting at offset 0x22a0)

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

Symbol table

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

4.4.5.2 Exclude all files in archive

To exclude all files in an archive from being linked, specify the archive to be excluded as the parameter of the `EXCLUDE_FILE (file ...)` command.

NOTE: Any sections whose names match the exclusion are still included in the link, unless they are stored in the excluded archive.

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

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

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


hexagon-clang -ffunction-sections -c a1.c a2.c a3.c a4.c -mv5
hexagon-ar cr lib23.a a2.o a3.o
hexagon-ar cr lib4.a a4.o
hexagon-link -T script.t -o mcld.out a1.o --whole-archive lib23.a lib4.a
--no-whole-archive
```

Section headers (starting at offset 0x22a0)

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

Symbol table

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

4.4.5.3 Exclude multiple files

To exclude multiple files from being linked, specify the files as parameters of the `EXCLUDE_FILE (file ...)` command.

NOTE: `EXCLUDE_FILE` accepts multiple file name parameters.

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

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

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

```
hexagon-clang -ffunction-sections -c a1.c a2.c a3.c a4.c -mv5
hexagon-ar cr lib23.a a2.o a3.o
hexagon-ar cr lib4.a a4.o
hexagon-link -T script.t -o mcld.out a1.o --whole-archive lib23.a lib4.a
--no-whole-archive
```

Section headers (starting at offset 0x2260)

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

Symbol table

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

4.4.5.4 Exclude archive and non-archive files

To exclude both archive and a non-archive files from being linked, specify the files as parameters of the `EXCLUDE_FILE (file ...)` command. This command searches both inside and outside archives for files to exclude.

The following example uses four relocatable files: `a1.o`, `a2.o`, `a3.o`, `a4.o`. Each file contains functions named `foo_N` and `bar_N`, where `N` indicates the digit in the file name. The linker script excludes `foo_1/2` but not `bar_1/2` from `.text1` (because `EXCLUDE_FILE` applies only to the immediately following section name):

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

hexagon-clang -ffunction-sections -c a1.c a2.c a3.c a4.c -mv5
hexagon-ar cr lib23.a a2.o a3.o
hexagon-ar cr lib4.a a4.o
hexagon-link -T script.t -o mcld.out a1.o --whole-archive lib23.a lib4.a
--no-whole-archive
```

Section headers (starting at offset 0x2260)

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

Symbol table

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

```
 14: 00000010    12 FUNC    GLOBAL DEFAULT    1 foo_3
 15: 00000030    12 FUNC    GLOBAL DEFAULT    1 foo_4
```

4.4.5.5 Conflicting wild cards

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

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

The following example uses four relocatable files: `a1.o`, `a2.o`, `a3.o`, `a4.o`. Each file contains functions named `foo_N` and `bar_N`, where `N` indicates the digit in the file name. The linker script does not exclude `foo_2` from `.text1` (because `EXCLUDE_FILE` has no effect in this case):

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


hexagon-clang -ffunction-sections -c a1.c a2.c a3.c a4.c -mv5
hexagon-ar cr lib23.a a2.o a3.o
hexagon-ar cr lib4.a a4.o
hexagon-link -T script.t -o mcld.out a1.o --whole-archive lib23.a lib4.a
--no-whole-archive
```

Section headers (starting at offset 0x22a0)

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

Symbol table

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

4.4.5.6 Build static executable

The following code example presents the linker script file for a static executable.

```
/* Script for -z combreloc: combine and sort reloc sections */
OUTPUT_FORMAT("elf32-littlehexagon", "elf32-bighexagon",
              "elf32-littlehexagon")
OUTPUT_ARCH(hexagon)
ENTRY(start)
SEARCH_DIR("/opt/codesourcery/hexagon/lib");
SECTIONS
{
  /* Code starts. */
  .start          :
  {
    KEEP (*(.start))
  } =0x00c0007f
  . = ALIGN(4K);
  .init           :
  {
    KEEP (*(.init))
  } =0x00c0007f
  .text           :
  {
    *(.text.unlikely .text.*_unlikely)
    *(.text.hot .text.hot.* .gnu.linkonce.t.hot.*)
    *(.text .stub .text.* .gnu.linkonce.t.*)
    /* .gnu.warning sections are handled specially by elf32.em.  */
    *(.gnu.warning)
  } =0x00c0007f
  .fini           :
  {
    KEEP (*(.fini))
  } =0x00c0007f
  PROVIDE (__etext = .);
  PROVIDE (_etext = .);
  PROVIDE (etext = .);
  . = ALIGN(4K);
  /* Constants start. */
  .rodata         :
        {
          *(.rodata.hot .rodata.hot.* .gnu.linkonce.r.hot.*)
          *(.rodata .rodata.* .gnu.linkonce.r.*)
        }
  .eh_frame_hdr   :  { *(.eh_frame_hdr) }
  .eh_frame       :  ONLY_IF_RO { KEEP (*(.eh_frame)) }
  .gcc_except_table   :  ONLY_IF_RO { *(.gcc_except_table
.gcc_except_table.*) }
  /* Data start. */
  /* Adjust the address for the data segment.  We want to adjust up to
     the same address within the page on the next page up.  */
  . = ALIGN (CONSTANT (MAXPAGESIZE)) - ((CONSTANT (MAXPAGESIZE) - .) &
            (CONSTANT (MAXPAGESIZE) - 1)); . = DATA_SEGMENT_ALIGN
            (CONSTANT (MAXPAGESIZE), CONSTANT (COMMONPAGESIZE));
```

```
  /* Exception handling  */
  .eh_frame       :  ONLY_IF_RW { KEEP (*(.eh_frame)) }
  .gcc_except_table   :  ONLY_IF_RW { *(.gcc_except_table
.gcc_except_table.*) }
  .preinit_array     :
  {
    PROVIDE_HIDDEN (__preinit_array_start = .);
    KEEP (*(.preinit_array))
    PROVIDE_HIDDEN (__preinit_array_end = .);
  }
  .init_array     :
  {
     PROVIDE_HIDDEN (__init_array_start = .);
     KEEP (*(SORT(.init_array.*)))
     KEEP (*(.init_array))
     PROVIDE_HIDDEN (__init_array_end = .);
  }
  .fini_array     :
  {
    PROVIDE_HIDDEN (__fini_array_start = .);
    KEEP (*(.fini_array))
    KEEP (*(SORT(.fini_array.*)))
    PROVIDE_HIDDEN (__fini_array_end = .);
  }
  . = ALIGN(4K);
  .ctors          :
  {
    /* gcc uses crtbegin.o to find the start of
       the constructors, so we make sure it is
       first.  Because this is a wildcard, it
       doesn't matter if the user does not
       actually link against crtbegin.o; the
       linker won't look for a file to match a
       wildcard.  The wildcard also means that it
       doesn't matter which directory crtbegin.o
       is in.  */
    KEEP (*crtbegin.o(.ctors))
    KEEP (*crtbegin?.o(.ctors))
    /* We don't want to include the .ctor section from
       the crtend.o file until after the sorted ctors.
       The .ctor section from the crtend file contains the
       end of ctors marker and it must be last */
    KEEP (*(EXCLUDE_FILE (*crtend.o *crtend?.o fini.o) .ctors))
    KEEP (*(SORT(.ctors.*)))
    KEEP (*(.ctors))
  }
  .dtors          :
  {
    KEEP (*crtbegin.o(.dtors))
    KEEP (*crtbegin?.o(.dtors))
    KEEP (*(EXCLUDE_FILE (*crtend.o *crtend?.o fini.o) .dtors))
    KEEP (*(SORT(.dtors.*)))
    KEEP (*(.dtors))
  }
```

```
.data           :
{
  *(.data.hot .data.hot.* .gnu.linkonce.d.hot.*)
  *(.data .data.* .gnu.linkonce.d.*)
  SORT(CONSTRUCTORS)
}
_edata = .; PROVIDE (edata = .);
. = ALIGN (64);
__bss_start = .;
.bss            :
{
 *(.dynbss)
 *(.bss.hot .bss.hot.* .gnu.linkonce.b.hot.*)
 *(.bss .bss.* .gnu.linkonce.b.*)
 *(COMMON)

 /* Align here to ensure that the .bss section occupies space up to
    _end.  Align after .bss to ensure correct alignment even if the
    .bss section disappears because there are no input sections. */
 . = ALIGN (. != 0 ? 64 : 1);
}
. = ALIGN (64);
_end = .;
/* Small data start. */
. = ALIGN(4K);
. = ALIGN (64);
.sdata          :
{
  PROVIDE (_SDA_BASE_ = .);
  *(.sdata.1 .sdata.1.* .gnu.linkonce.s.1.*)
  *(.sbss.1 .sbss.1.* .gnu.linkonce.sb.1.*)
  *(.scommon.1 .scommon.1.*)
  *(.sdata.2 .sdata.2.* .gnu.linkonce.s.2.*)
  *(.sbss.2 .sbss.2.* .gnu.linkonce.sb.2.*)
  *(.scommon.2 .scommon.2.*)
  *(.sdata.4 .sdata.4.* .gnu.linkonce.s.4.*)
  *(.sbss.4 .sbss.4.* .gnu.linkonce.sb.4.*)
  *(.scommon.4 .scommon.4.*)
  *(.lit[a4] .lit[a4].* .gnu.linkonce.l[a4].*)
  *(.sdata.8 .sdata.8.* .gnu.linkonce.s.8.*)
  *(.sbss.8 .sbss.8.* .gnu.linkonce.sb.8.*)
  *(.scommon.8 .scommon.8.*)
  *(.lit8 .lit8.* .gnu.linkonce.l8.*)
  *(.sdata.hot .sdata.hot.* .gnu.linkonce.s.hot.*)
  *(.sdata .sdata.* .gnu.linkonce.s.*)
}
.sbss           :
{
  PROVIDE (__sbss_start = .);
  PROVIDE (___sbss_start = .);
  *(.dynsbss)
  *(.sbss.hot .sbss.hot.* .gnu.linkonce.sb.hot.*)
  *(.sbss .sbss.* .gnu.linkonce.sb.*)
  *(.scommon .scommon.*)
  . = ALIGN (. != 0 ? 64 : 1);
  PROVIDE (__sbss_end = .);
```

```
    PROVIDE (___sbss_end = .);
  }
  . = ALIGN (64);
  PROVIDE (end = .);
  . = DATA_SEGMENT_END (.);
  /* Stabs debugging sections.  */
  .stab          0 :  { *(.stab) }
  .stabstr       0 :  { *(.stabstr) }
  .stab.excl     0 :  { *(.stab.excl) }
  .stab.exclstr  0 :  { *(.stab.exclstr) }
  .stab.index    0 :  { *(.stab.index) }
  .stab.indexstr 0 :  { *(.stab.indexstr) }
  .comment       0 :  { *(.comment) }
  /* DWARF debug sections.
     Symbols in the DWARF debugging sections are relative to the beginning
     of the section so we begin them at 0.  */
  /* DWARF 1 */
  .debug          0 :  { *(.debug) }
  .line           0 :  { *(.line) }
  /* GNU DWARF 1 extensions */
  .debug_srcinfo  0 :  { *(.debug_srcinfo) }
  .debug_sfnames  0 :  { *(.debug_sfnames) }
  /* DWARF 1.1 and DWARF 2 */
  .debug_aranges  0 :  { *(.debug_aranges) }
  .debug_pubnames 0 :  { *(.debug_pubnames) }
  /* DWARF 2 */
  .debug_info     0 :  { *(.debug_info .gnu.linkonce.wi.*) }
  .debug_abbrev   0 :  { *(.debug_abbrev) }
  .debug_line     0 :  { *(.debug_line) }
  .debug_frame    0 :  { *(.debug_frame) }
  .debug_str      0 :  { *(.debug_str) }
  .debug_loc      0 :  { *(.debug_loc) }
  .debug_macinfo  0 :  { *(.debug_macinfo) }
  /* SGI/MIPS DWARF 2 extensions */
  .debug_weaknames 0 :  { *(.debug_weaknames) }
  .debug_funcnames 0 :  { *(.debug_funcnames) }
  .debug_typenames 0 :  { *(.debug_typenames) }
  .debug_varnames  0 :  { *(.debug_varnames) }
  /* DWARF 3 */
  .debug_pubtypes 0 :  { *(.debug_pubtypes) }
  .debug_ranges   0 :  { *(.debug_ranges) }
  .gnu.attributes 0 :  { KEEP (*(.gnu.attributes)) }
  /DISCARD/       :  { *(.note.GNU-stack) *(.gnu_debuglink) *(.gnu.lto_*) }
}
```

4.4.5.7 Build dynamic executable

The following code example presents the linker script file for a dynamic executable.

```
/* Script for -z combreloc: combine and sort reloc sections */
OUTPUT_FORMAT("elf32-littlehexagon", "elf32-littlehexagon",
              "elf32-littlehexagon")
ENTRY(start)
SEARCH_DIR("/opt/codesourcery/hexagon/lib");
SECTIONS
{
  /* Start EBI memory. */
  .interp         :
                    { *(.interp) }
  .note.gnu.build-id :  { *(.note.gnu.build-id) }
  .hash           :  { *(.hash) }
  .gnu.hash       :  { *(.gnu.hash) }
  .dynsym         :  { *(.dynsym) }
  .dynstr         :  { *(.dynstr) }
  .rela.dyn       :
    {
      *(.rela.init)
      *(.rela.text .rela.text.* .rela.gnu.linkonce.t.*)
      *(.rela.fini)
      *(.rela.rodata .rela.rodata.* .rela.gnu.linkonce.r.*)
      *(.rela.data .rela.data.* .rela.gnu.linkonce.d.*)
      *(.rela.tdata .rela.tdata.* .rela.gnu.linkonce.td.*)
      *(.rela.tbss .rela.tbss.* .rela.gnu.linkonce.tb.*)
      *(.rela.ctors)
      *(.rela.dtors)
      *(.rela.got)
      *(.rela.sdata .rela.lit[a48] .rela.sdata.* .rela.lit[a48].*
        .rela.gnu. linkonce.s.* .rela.gnu.linkonce.l[a48].*)
      *(.rela.sbss .rela.sbss.* .rela.gnu.linkonce.sb.*)
      *(.rela.sdata2 .rela.sdata2.* .rela.gnu.linkonce.s2.*)
      *(.rela.sbss2 .rela.sbss2.* .rela.gnu.linkonce.sb2.*)
      *(.rela.bss .rela.bss.* .rela.gnu.linkonce.b.*)
      PROVIDE_HIDDEN (__rela_iplt_start = .);
      *(.rela.iplt)
      PROVIDE_HIDDEN (__rela_iplt_end = .);
    }
  .rela.plt       :
    {
      *(.rela.plt)
    }
  /* Code starts. */
  . = ALIGN (DEFINED (TEXTALIGN) ? (TEXTALIGN * 1K) : CONSTANT
(MAXPAGESIZE));
  .start          :
  {
    KEEP (*(.start))
  } =0x00c0007f
  . = ALIGN(4K);
  .init           :
  {
```

```
    KEEP (*(.init))
  } =0x00c0007f
  .plt            :  { *(.plt) }
  . = ALIGN (. + CONSTANT (COMMONPAGESIZE), CONSTANT (COMMONPAGESIZE));
  .text           :
  {
    *(.text.unlikely .text.*_unlikely)
    *(.text.hot .text.hot.* .gnu.linkonce.t.hot.*)
    *(.text .stub .text.* .gnu.linkonce.t.*)
    /* .gnu.warning sections are handled specially by elf32.em.  */
    *(.gnu.warning)
  } =0x00c0007f
  .fini           :
  {
    KEEP (*(.fini))
  } =0x00c0007f
  PROVIDE (__etext = .);
  PROVIDE (_etext = .);
  PROVIDE (etext = .);
/* Constants start. */
  . = ALIGN (4K);
  .rodata         :
        {
          *(.rodata.hot .rodata.hot.* .gnu.linkonce.r.hot.*)
          *(.rodata .rodata.* .gnu.linkonce.r.*)
        }
  .eh_frame_hdr   :  { *(.eh_frame_hdr) }
  .eh_frame       :  ONLY_IF_RO { KEEP (*(.eh_frame)) }
  .gcc_except_table   :  ONLY_IF_RO { *(.gcc_except_table
.gcc_except_table.*) }
  /* Data start. */
  . = ALIGN (4K);
  /* Exception handling  */
  .eh_frame       :  ONLY_IF_RW { KEEP (*(.eh_frame)) }
  .gcc_except_table   :  ONLY_IF_RW { *(.gcc_except_table
.gcc_except_table.*) }
  .preinit_array     :
  {
    PROVIDE_HIDDEN (__preinit_array_start = .);
    KEEP (*(.preinit_array))
    PROVIDE_HIDDEN (__preinit_array_end = .);
  }
  .init_array     :
  {
     PROVIDE_HIDDEN (__init_array_start = .);
     KEEP (*(SORT(.init_array.*)))
     KEEP (*(.init_array))
     PROVIDE_HIDDEN (__init_array_end = .);
  }
  .fini_array     :
  {
    PROVIDE_HIDDEN (__fini_array_start = .);
    KEEP (*(.fini_array))
    KEEP (*(SORT(.fini_array.*)))
    PROVIDE_HIDDEN (__fini_array_end = .);
  }
```

```
.ctors          :
{
  /* gcc uses crtbegin.o to find the start of
     the constructors, so we make sure it is
     first.  Because this is a wildcard, it
     doesn't matter if the user does not
     actually link against crtbegin.o; the
     linker won't look for a file to match a
     wildcard.  The wildcard also means that it
     doesn't matter which directory crtbegin.o
     is in.  */
  KEEP (*crtbegin.o(.ctors))
  KEEP (*crtbegin?.o(.ctors))

  /* We don't want to include the .ctor section from
     the crtend.o file until after the sorted ctors.
     The .ctor section from the crtend file contains the
     end of ctors marker and it must be last */
  KEEP (*(EXCLUDE_FILE (*crtend.o *crtend?.o fini.o) .ctors))
  KEEP (*(SORT(.ctors.*)))
  KEEP (*(.ctors))
}
.dtors          :
{
  KEEP (*crtbegin.o(.dtors))
  KEEP (*crtbegin?.o(.dtors))
  KEEP (*(EXCLUDE_FILE (*crtend.o *crtend?.o fini.o) .dtors))
  KEEP (*(SORT(.dtors.*)))
  KEEP (*(.dtors))
}
_DYNAMIC = .;
.dynamic        :  { *(.dynamic) }
.got            :  { *(.got) *(.igot) }
. = DATA_SEGMENT_RELRO_END (16, .);
.got.plt        :  { *(.got.plt)  *(.igot.plt) }
.data           :
{
  *(.data.hot .data.hot.* .gnu.linkonce.d.hot.*)
  *(.data .data.* .gnu.linkonce.d.*)
  SORT(CONSTRUCTORS)
}
.data1          :  { *(.data1) }
_edata = .; PROVIDE (edata = .);
/* Small data start. */
. = ALIGN (4K);
. = ALIGN (64);
.sdata          :
{
  PROVIDE (_SDA_BASE_ = .);
  *(.sdata.1.8 .sdata.1.8.* .gnu.linkonce.s.1.8.*)
  *(.sbss.1.8 .sbss.1.8.* .gnu.linkonce.sb.1.8.*)
  *(.scommon.1.8 .scommon.1.8.*)
  *(.sdata.1.4 .sdata.1.4.* .gnu.linkonce.s.1.4.*)
  *(.sbss.1.4 .sbss.1.4.* .gnu.linkonce.sb.1.4.*)
  *(.scommon.1.4 .scommon.1.4.*)
  *(.sdata.1.2 .sdata.1.2.* .gnu.linkonce.s.1.2.*)
```

```
  *(.sbss.1.2 .sbss.1.2.* .gnu.linkonce.sb.1.2.*)
  *(.scommon.1.2 .scommon.1.2.*)
  *(.sdata.1.1 .sdata.1.1.* .gnu.linkonce.s.1.1.*)
  *(.sbss.1.1 .sbss.1.1.* .gnu.linkonce.sb.1.1.*)
  *(.scommon.1.1 .scommon.1.1.*)
  *(.sdata.1 .sdata.1.* .gnu.linkonce.s.1.*)
  *(.sbss.1 .sbss.1.* .gnu.linkonce.sb.1.*)
  *(.scommon.1 .scommon.1.*)
  *(.sdata.2.8 .sdata.2.8.* .gnu.linkonce.s.2.8.*)
  *(.sbss.2.8 .sbss.2.8.* .gnu.linkonce.sb.2.8.*)
  *(.scommon.2.8 .scommon.2.8.*)
  *(.sdata.2.4 .sdata.2.4.* .gnu.linkonce.s.2.4.*)
  *(.sbss.2.4 .sbss.2.4.* .gnu.linkonce.sb.2.4.*)
  *(.scommon.2.4 .scommon.2.4.*)
  *(.sdata.2.2 .sdata.2.2.* .gnu.linkonce.s.2.2.*)
  *(.sbss.2.2 .sbss.2.2.* .gnu.linkonce.sb.2.2.*)
  *(.scommon.2.2 .scommon.2.2.*)
  *(.sdata.2 .sdata.2.* .gnu.linkonce.s.2.*)
  *(.sbss.2 .sbss.2.* .gnu.linkonce.sb.2.*)
  *(.scommon.2 .scommon.2.*)
  *(.sdata.4.8 .sdata.4.8.* .gnu.linkonce.s.4.8.*)
  *(.sbss.4.8 .sbss.4.8.* .gnu.linkonce.sb.4.8.*)
  *(.scommon.4.8 .scommon.4.8.*)
  *(.sdata.4.4 .sdata.4.4.* .gnu.linkonce.s.4.4.*)
  *(.sbss.4.4 .sbss.4.4.* .gnu.linkonce.sb.4.4.*)
  *(.scommon.4.4 .scommon.4.4.*)
  *(.sdata.4 .sdata.4.* .gnu.linkonce.s.4.*)
  *(.sbss.4 .sbss.4.* .gnu.linkonce.sb.4.*)
  *(.scommon.4 .scommon.4.*)
  *(.lit[a4] .lit[a4].* .gnu.linkonce.l4.*)
  *(.sdata.8.8 .sdata.8.8.* .gnu.linkonce.s.8.8.*)
  *(.sbss.8.8 .sbss.8.8.* .gnu.linkonce.sb.8.8.*)
  *(.scommon.8.8 .scommon.8.8.*)
  *(.sdata.8 .sdata.8.* .gnu.linkonce.s.8.*)
  *(.sbss.8 .sbss.8.* .gnu.linkonce.sb.8.*)
  *(.scommon.8 .scommon.8.*)
  *(.lit8 .lit8.* .gnu.linkonce.l8.*)
  *(.sdata.hot .sdata.hot.* .gnu.linkonce.s.hot.*)
  *(.sdata .sdata.* .gnu.linkonce.s.*)
}
.got            :  { *(.got) *(.igot) }
. = ALIGN (64);
.sbss           :
{
  PROVIDE (__sbss_start = .);
  PROVIDE (___sbss_start = .);
  *(.dynsbss)
  *(.sbss.hot .sbss.hot.* .gnu.linkonce.sb.hot.*)
  *(.sbss .sbss.* .gnu.linkonce.sb.*)
  *(.scommon .scommon.*)
  . = ALIGN (. != 0 ? 64 : 1);
  PROVIDE (__sbss_end = .);
  PROVIDE (___sbss_end = .);
}
. = ALIGN (64);
__bss_start = .;
```

```
  .bss            :
  {
   *(.dynbss)
   *(.bss.hot .bss.hot.* .gnu.linkonce.b.hot.*)
   *(.bss .bss.* .gnu.linkonce.b.*)
   *(COMMON)
   /* Align here to ensure that the .bss section occupies space up to
      _end.  Align after .bss to ensure correct alignment even if the
      .bss section disappears because there are no input sections. */
   . = ALIGN (. != 0 ? 64 : 1);
  }
  . = ALIGN (64);
  _end = .;
  . = ALIGN (64);
  PROVIDE (end = .);
  . = DATA_SEGMENT_END (.);
  .comment       0 :  { *(.comment) }
  /* DWARF debug sections.
     Symbols in the DWARF debugging sections are relative to the beginning
     of the section so we begin them at 0.  */
  /* DWARF 1 */
  .debug          0 :  { *(.debug) }
  .line           0 :  { *(.line) }
  /* GNU DWARF 1 extensions */
  .debug_srcinfo  0 :  { *(.debug_srcinfo) }
  .debug_sfnames  0 :  { *(.debug_sfnames) }
  /* DWARF 1.1 and DWARF 2 */
  .debug_aranges  0 :  { *(.debug_aranges) }
  .debug_pubnames 0 :  { *(.debug_pubnames) }
  /* DWARF 2 */
  .debug_info     0 :  { *(.debug_info .gnu.linkonce.wi.*) }
  .debug_abbrev   0 :  { *(.debug_abbrev) }
  .debug_line     0 :  { *(.debug_line) }
  .debug_frame    0 :  { *(.debug_frame) }
  .debug_str      0 :  { *(.debug_str) }
  .debug_loc      0 :  { *(.debug_loc) }
  .debug_macinfo  0 :  { *(.debug_macinfo) }
  /* SGI/MIPS DWARF 2 extensions */
  .debug_weaknames 0 :  { *(.debug_weaknames) }
  .debug_funcnames 0 :  { *(.debug_funcnames) }
  .debug_typenames 0 :  { *(.debug_typenames) }
  .debug_varnames  0 :  { *(.debug_varnames) }
  /* DWARF 3 */
  .debug_pubtypes 0 :  { *(.debug_pubtypes) }
  .debug_ranges   0 :  { *(.debug_ranges) }
  .gnu.attributes 0 :  { KEEP (*(.gnu.attributes)) }
  /DISCARD/       :  { *(.note.GNU-stack) *(.gnu_debuglink) *(.gnu.lto_*) }
}
```

4.5 Link-time optimization

Link-time optimization (LTO) is a compilation mode in which an intermediated language is written to object files and the optimizer is invoked during the linking stage. This approach has the following advantages:

- The scope of inter-procedural analysis can encompass the entire program that is visible at link-time.
- Compared to the file-by-file approach, where every file is optimized independently, LTO can leverage knowledge of the entire program to apply various inter-procedural optimizations (function inlining, dead code stripping, and so on).

The following graphic illustrates an overview of where LTO fits in the compilation process.

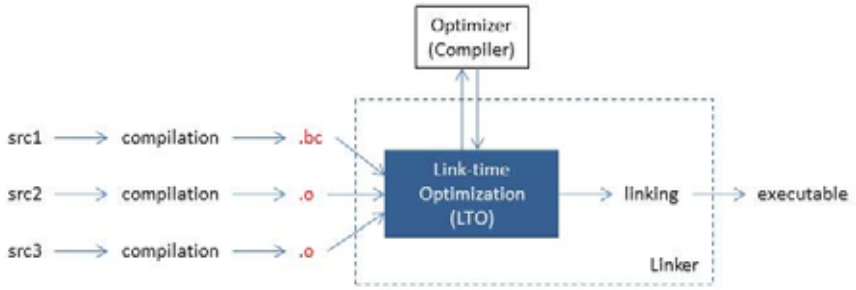

Figure 4-1 LTO in the compilation process

4.5.1 Example

The following example illustrates the advantages of the LTO integrated approach and clean interface. This example requires a system linker that supports LTO through the interface described in this document. In this example:

- Input source file `a.c` is compiled into LLVM bitcode form.
- Input source file `main.c` is compiled into native object code.

```
--- a.h ---
extern int foo1(void);
extern void foo2(void);
extern void foo4(void);
--- a.c ---
#include "a.h"

static signed int i = 0;
void foo2(void){
  i = -1;
}

static int foo3(){
  foo4();
  return 10;
}
```

```
int foo1(void){
  int data = 0;

  if(i<0)
    data = foo3();
  data = data + 42;
  return data;
}

--- main.c ---
#include <stdio.h>
#include "a.h"

void foo4(void){
  printf("Hi\n");
}

int main(){
  return foo1();
}
To compile, run:
% clang -emit-llvm -c a.c -o a.o  # <-- a.o is the LLVM bitcode file
% clang -c main.c -o main.o      # <-- main.o is the native object file
% clang a.o main.o -o main       # <-- standard link command without
                                       modifications
```

In this example:

- The linker recognizes that `foo2()` is an externally visible symbol defined in the LLVM bitcode file. The linker completes its usual symbol resolution pass and finds that `foo2()` is not used anywhere. This information is used by the LLVM optimizer and it removes `foo2()`.
- As soon as `foo2()` is removed, the optimizer recognizes that condition `i < 0` is always false, which means `foo3()` is never used. Hence, the optimizer also removes `foo3()`.
- This, in turn, enables the linker to remove `foo4()`.

This example illustrates the advantage of tight integration with the linker. Here, the optimizer cannot remove `foo3()` without the linker's input.

4.5.2 Multiphase communication between libLTO and linker

The linker collects information about symbol definitions and uses in various link objects, which is more accurate than any information collected by other tools during typical build cycles. The linker collects this information by looking at the definitions and uses of the symbols in native .o files and using symbol visibility information. The linker also uses user-supplied information, such as a list of exported symbols.

The LLVM optimizer collects control flow information and data flow information, and it knows much more about program structure from the optimizer's point of view. The goal is to take advantage of tight integration between the linker and the optimizer by sharing this information during various linking phases.

4.6 Develop linker plug-ins

The QCLD allows one or more linker plug-ins 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 plug-ins. They are built as shared libraries that allow the linker to dynamically load at link time. The plug-ins use a fixed API that allows you to communicate with the linker.

Plug-ins are currently supported on all architectures where the functionality is available.

The Hexagon Tools 8.2 release introduced the functionality for writing plug-ins and integrating the functionality at link time.

4.6.1 Plug-in usage

Most developers use linker scripts, and the linker plug-in 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.

4.6.2 Linker wrapper

Build a plug-in 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 plug-in. The linker wrapper is named `libLW.so` on Linux platforms and `LW.dll` on Windows platforms.

4.6.3 User plug-in types

The intent of each user-developed plug-in is associated with an appropriate interface. You can implement the plug-in algorithm by deriving it from one such interface.

You can use more than one interface in a single plug-in. Plug-in 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.

Each interface type has an associated header file that you can include in the plug-in:

Interface type	Header file
Section iterator plug-in type	`SectionIteratorPlugin.h`
Chunk iterator plug-in type	`ChunkIteratorPlugin.h`
Control memory size plug-in type	`ControlMemorySizePlugin.h`
Control file size plug-in type	`ControlFileSizePlugin.h`

4.6.4 LinkerScript changes

Linker plug-ins can be enabled using linker scripts. The linker script keyword uses a fixed syntax:

`<PluginType>("LibraryName", "PluginName" [, "PluginOptions"])`

Options:

`PluginType`

Corresponds to one of the available interface types.

`LibraryName`

Corresponds to the dynamic library that contains the plug-in 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` extension on Linux/Windows, respectively

`PluginName`

Specifies the name of the plug-in. 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.

4.6.5 User plug-in work flow

Use the following steps to create one or more linker plug-ins.

1. Determine the appropriate interfaces.
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. Write a `RegisterAll()` function to register the plug-ins.
7. Write a `getPlugin()` function to return the appropriate plug-in when the linker queries with `PluginName()`.

4.6.6 Linker work flow

All user plug-ins that are loaded by the linker must be initialized properly before the plug-in can communicate with the linker and perform the steps.

4.6.6.1 Linker operation

The linker performs the following sequence of operations with respect to plug-ins:

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()` function in the library, which registers all the plug-ins that are contained in the library.
4. Queries the library using the `getPlugin()` function with `PluginName`.

   The library returns the appropriate object for the linker to use to run the plug-in 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.

4.6.6.2 Plug-in tracing

Trace the linker work flow with the following option:

```
-trace==plugin
```

4.6.7 Plug-in 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`.

4.6.8 LinkerWrapper commands

The plug-in uses the LinkerWrapper to communicate and exchange information with the linker. Following are the LinkerWrapper commands.

NOTE: Use the `PluginADT.h` file to learn about the APIs documented in the header file.

`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.

4.6.9 Linker plug-in interface

Following is the plug-in 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(){}
};
```

4.6.10 Plug-in interfaces

The following interfaces are available in the release toolchain.

4.6.10.1 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;
};
```

4.6.10.2 SectionMatcher interface

Use the linker script keyword, `PLUGIN_SECTION_MATCHER`.

This interface allows you to process every input section, and it allows the plug-in 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;
};
```

4.6.10.3 ChunkIterator interface

Use the linker script keyword, `PLUGIN_ITER_CHUNKS`.

This interface allows you to process every input chunk in an output section using `ProcessChunk()`.

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;
};
```

4.6.10.4 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 plug-in 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;
};
```

4.6.10.5 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;
};
```

4.6.10.6 OutputSectionIterator interface

Use the linker script keyword, `PLUGIN_OUTPUT_SECTION_ITER`.

The interface is defined as follows:

```
class OutputSectionIteratorPlugin : public Plugin {  public:
/* OutputSection that the linker will call the client  with */
virtual void processOutputSection(OutputSection O)= 0;
};
```

The interface allows the plug-in 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 plug-in can query for the size of the OutputSection.

4.6.11 Map file

The Map file records any changes made by the plug-in for the layout. This is a good place to determine how the layout was affected due to the plug-in algorithm.

4.6.12 Example plug-ins

4.6.12.1 Match island sections and get uses

LinkerScript

```
PHDRS {
P PT_LOAD;
A PT_LOAD;
}
PLUGIN_ITER_SECTIONS ("matchislandsections", "MATCHANDFINDUSES") SECTIONS {
  .island : {
    *(.text.island*)
  }:P
  .bar :{
    *(.text.bar)
  }:A
}
```

Example user code for linking

```
int foo(){return 0; }
int bar (){return foo(); }
_attribute_((section (".text.island_baz")))int baz(){return foo()+ bar();
}
```

Plug-in code

```
#include “SectioniteratorPlugin.h”
#include <iostream>
#include <string>
#include <cstring>
#include <unordered_map>
#include <algorithm>
#include <queue>
#include <set>

using namespace QCLD;

class DLL_A_EXPORT FindUsesPlugin : public SectionIteratorPlugin {
public:
  FindUsesPlugin() : SectionIteratorPlugin ("MATCHANDFINDUSES"){}
  void Init(std::string Options)override {}
  void processSection(QCLD::Section S)override {
    if (S.matchPattern(".text.island*"))
      m_Sections.push_back(S);
  }

  Status Run(bool Trace) override {
    for (auto &S : m_Sections)
      printSectionUses(S);
    return Plugin::Status::SUCCESS;
  }

  void printSectionUses(QCLD::Section S){
    std::queue<QCLD::Use> Uses;
    for (auto &U : Linker->getUses(S))
      Uses.push(U);
    std::set<Chunk, Chunk::Compare> SectionUses;
    while (!Uses.empty()){
      Use &U = Uses.front();
      Uses.pop();
      Chunk ChunkForUse = U.getChunk{};
      SectionUses.insert(ChunkForUse);
      for (auto &V : Linker->getUses(ChunkForUse))
        Uses.push(V);
    }
    std::cout << "Uses for section " << S.getName() << "\n";
    for (auto &C : SectionUses )
      std::cout << C.getName() << "\n";
  }

  void Destroy() override {}

  uint32_t GetLastError() override { return 0; }
  std::string GetlastErrorAsString() override { return "SUCCESS"; }
  std::string GetName() override { return "MATCHANDFINDUSES "; }

private:
  std::vector<QCLD::Section> m_Sections;

};
```

```
std::unordered_map<std::string, Plugin *> Plugins;
extern "C" {
bool DLL_A_EXPORT RegisterAll(){
    Plugins["MATCHANDFINDUSES"]= new FindUsesPlugin();
    return true;
  }

  Plugin DLL_A_EXPORT *getPlugin(const char *T) {
    return Plugins[std::string(T)];
  }
}
```

4.6.12.2 Reorder chunks

Linker script

```
PHDRS {
  P PT_LOAD;
  A PT_LOAD;
  B PT_LOAD;
}
SECTIONS {
.foo PLUGIN_ITER_CHUNKS("order","ORDERBLOCKS", "G0") : {
    *(.text.foo)
    *(.text.baz)
  }:P
  .bar :{
    *(.text.bar)
  }:A
  .plugin : {
    *(.pluginfoo )
  }:B
}
```

Input file for linking

```
int foo() { return bar(); }
int bar() { return foo(); }
int baz() { return foo()+ bar(); }
```

Plug-in code

```
#include "ChunkIteratorPlugin.h"
#include <string>
#include <cstring>
#include <unordered_map>
#include <algorithm>

using namespace QCLD;

class DLL_A_EXPORT OrderChunksPlugin : public ChunkIteratorPlugin {
```

```
public:

  OrderChunksPlugin() : ChunkIteratorPlugin("ORDERBLOCKS"), Chunks(){}

  void Init(std::string Options) override {}

  void processChunk(Chunk C) override {Chunks.push_back(C); }

  Status Run(bool Trace) override {
    std::sort(Chunks.begin(), Chunks.end(),
              [](const Chunk &A, const Chunk &B) { return A.getName ()<
              B.getName(); });
    return Plugin::Status::SUCCESS;
    }

  std::vector<Chunk> getChunks() override { return Chunks; }

  void Destroy() override {}

  uint32_t GetLastError() override { return 0; }

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

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

private:

  std::vector<Chunk> Chunks;

};

std::unordered_map<std::string, Plugin *> Plugins;

extern "C" {
bool DLL_A_EXPORT RegisterAll() {
  Plugins["ORDERBLOCKS"]= new OrderChunksPlugin ();
  return true;
  }
  Plugin DLL_A_EXPORT *getPlugin(const char *T) {
    return Plugins [std :string(T)];
  }
}
```

4.7 Bundled plug-ins

The plug-ins described in this section are included with the linker as part of the toolchain release and can be used without installing them separately.

4.7.1 Section budgeting

Section budgeting enables you to allocate a size budget for specific output sections. If an output section exceeds its size budget, section budgeting moves input sections into a user-defined overflow section based on a priority ordering. The priority is determined using profile counts if the compiler's profile-driven section placement functionality is being used.

- Sections with the lowest PGO counts (if section placement is being used) are moved first.
- If the section is still not within budget after all sections with PGO counts have been moved to the overflow section, sections are moved in the reverse order in which they appear in the output section (last in, first out).

4.7.1.1 Configuration

The section budgeting plug-in must be enabled in the linker script using the following directive at the top level (outside any `SECTIONS { . . . }` block or similar block):

```
PLUGIN_ITER_SECTIONS("SectionBudgetingV2", "SectionBudgetingV2",
"budget.cfg")
```

Alternatively, the configuration file can also be specified via the `SECTION_BUDGETING_CFG` environment variable. If an environment variable is used, the file name in the linker script can be left blank, as in the following example:

```
PLUGIN_ITER_SECTIONS("SectionBudgetingV2", "SectionBudgetingV2", "")
```

The `budget.cfg` file specifies the budgets and overflow sections:

```
.text_foo:
    size: 22528
    policy: no-overflow

.rodata_foo:
    size: 7168
    policy: no-overflow

.text_sw_hot:
    size: 229376
    overflow: .text_sw
```

NOTE: The indentation (using an equal amount of spaces) of size: and policy: is significant. This is a YAML file, so you can use # to comment out lines. The budget is specified in bytes.

The configuration file must be located in the directory from which the linker is called (typically `build/ms/`).

The GLOBAL configuration specifies an optional report file (report-file:) name to contain summary of taken actions, and an optional detailed report file (section-contents-file:) name to contain a full list of objects allocated to every section.

4.7.2 Profile-based section sorting plug-in

This plug-in allows you to rearrange the layout of functions in an output section based on call graph and run-time behavior (via profile guided feedback) in a way that increases spatial access locality and decreases frequent call distance. This may also reduce the number of trampolines.

The default section layout chosen by the linker generally follows the order of files on the command line as well as linker script rules (if specified). However, the C/C++ standards do not guarantee a specific order of functions in the final executable. The section sorting plug-in takes advantage of this fact by detecting strongly connected components in the function call graph (over-imposed with call frequencies) and arranging them in closely located clusters with the cluster size driven by instruction cache block parameters. This increases the probability of cache hits for frequent callers/callees. The clusters themselves are arranged in random order.

For example, as shown in the graphic, any arrangement of these functions other than ABCD in memory will result in a larger overall call distance if call frequency is considered.

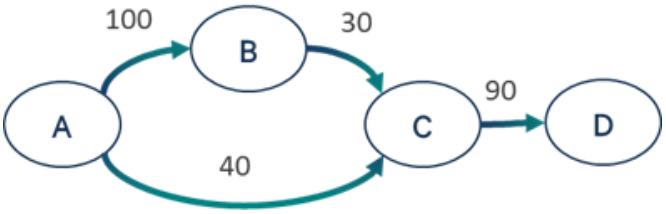

Figure 4-2 Section sorting plug-in

4.7.2.1 Configuration

To enable the section sorting plug-in, use the following syntax in the linker script:

```
.mytext PLUGIN_ITER_CHUNKS("SortPGO","SORTPGO","LLVMProfile.prof")
{
 *dir1*:(.text .text.*)
 *dir2*:(.text .text.*)
}
```

4.8 Control symbol exports

Use the `--dynamic-list`, `-E`, and `--version-script` options to control the symbols that are added to the `.dynsym` section of the dynamic linker.

4.8.1 Use --dynamic-list and -E options

The `--dynamic-list` and `-E` options only affect an executable that was created by linking with a shared object.

- The `-E` option exports all symbols. The linker does not regard any `--dynamic-list` if it is used with this option.
- The `--dynamic-list` option exports symbols that are in a file that is passed with this option.

Typically, the linker exports all symbols that are not defined but are resolved by the shared library at runtime in the `.dynsym` table. These exported symbols are added together with the symbols that are in the dynamic list.

Example

In `main.c`:

```
extern int foo();
int bar() { return 1; }
int baz() { return 2; }
int main() { return bar() + baz() + foo() ; }
```

In `foo.c`:

```
int foo() { return 3; }
int anotherfoo() { return 4;}
static int foo_local() { return foo(); }
int shlibbar() { return foo() + anotherfoo(); }
```

In `list`:

```
{
  bar;
};
```

NOTE: In the dynamic list, each symbol name ends with a semicolon, as does the list itself.

Usage:

```
hexagon-clang -fpic -c foo.c -o foo.o
hexagon-clang -c main.c
hexagon-link -shared -o libfoo.so foo.o
hexagon-link -dy --dynamic-list list main.o libfoo.so -o main
hexagon-readelf -D -s main
```

Symbol table for the image:

```
Num Buc:    Value  Size  Type  Bind   Vis      Ndx  Name
  2   0: 000000d0    12  FUNC  GLOBAL DEFAULT    7  bar
  1   0: 00000000     0  FUNC  GLOBAL DEFAULT  UND  foo
```

NOTE: As shown, `foo` was exported even though it was not in the list, because it was required to fully resolve main at runtime.

Usage:

```
hexagon-link -dy -E --dynamic-list list main.o libfoo.so
hexagon-readelf -D -s main
```

Symbol table for the image:

```
Num Buc:    Value  Size  Type  Bind   Vis      Ndx  Name
  3   1: 00000118    48  FUNC  GLOBAL DEFAULT    7  main
  4   2: 0000010c    12  FUNC  GLOBAL DEFAULT    7  baz
  2   4: 00000100    12  FUNC  GLOBAL DEFAULT    7  bar
  1   4: 00000000     0  FUNC  GLOBAL DEFAULT  UND  foo
```

The `--dynamic-list` option has no effect on linking a shared object. To build an executable, the `--dynamic-list` option has the same effect with `--force-dynamic` as with the `-dy` option.

The `--dynamic-list` option has no effect on garbage collection in the linker. The symbols that are not to be collected by garbage collection optimization must be explicitly listed in the linker script or added to the script using the `--extern-list` option.

4.8.2 Use --version-script option

The `--version-script` option is more useful when creating shared objects (although it can be used in creating executables) to control symbol exports in the `.dynsym` table.

The version script has two scope specifiers: `global` and `local`. The symbols listed under each specifier are used as guides to restrict the symbol export. This script is only used to restrict, and not increase, a symbol's scope. Symbols can also be specified with glob patterns for matching symbols. The version script also takes effect when creating program independent executables that have a `-shared` switch passed to them.

Example

```
version_script
{
  global:
  foo*;
  local:
  *bar*;
};
```

Usage:

```
hexagon-link -shared --version-script=version_script -o libfoo.so foo.o
hexagon-readelf -D -s libfoo.so
```

Symbol table for the image:

```
Num Buc:    Value  Size  Type  Bind   Vis     Ndx  Name
  1   0: 00000160    12  FUNC  GLOBAL DEFAULT   6  foo
  2   2: 00000188    12  FUNC  GLOBAL DEFAULT   6  anotherfoo
```

Observations:

- `foo_local` was not exported despite matching a pattern in the global scope because it is a local symbol.
- Both `bar` and `shlibbar` were not exported because they are restricted to the local scope, as specified in the version script.

NOTE: Use the version script to control symbol versions using version nodes and a dependent node specification in the script. Currently, these features are not supported. Only anonymous version nodes for symbols are supported.

5 Archiver

The archiver utility creates and maintains groups of files combined into an archive. When an archive is created, new files can be added to it and existing files can be extracted, deleted, or replaced.

Files are named in the archive by their last file name component, so if a file referenced by a path containing a / is archived, it is named by the last component of the path. Similarly, when matching paths listed on the command line against file names stored in the archive, only the last component of the path is compared.

The typical use of the archiver is to create and maintain libraries that are suitable for use with the linker, although the archiver is not restricted to this purpose. It can create and manage an archive symbol table that is used to speed up linking. If a symbol table is present in an archive, it is kept up to date by subsequent operations on the archive.

5.1 Command

To start the archiver from the command line:

```
hexagon-ar <genericiers><operation><operation modifier with positional
member names (if any)> archive members
```

To accomplish specific archiving tasks, enter one of the following commands from the command line:

```
hexagon-ar -d[Tvz] archive file...
hexagon-ar -m[Ta|b|i] member archive file...
hexagon-ar -p[Tv] archive file...
hexagon-ar -q[TcDFs|Svz] archive file...
hexagon-ar -r[TcDFs|Suvza|b|i] member archive file...
hexagon-ar -s[Dz] archive
hexagon-ar -t[Tv] archive file...
hexagon-ar -x [CTouv] archive file...
hexagon-ar -M
hexagon-ar -V
```

NOTE: All the options must be clubbed together as shown in this example (space separation is not allowed). The only options where spaces are permitted are the member names specification for `-i`, `-a`, and `-b` positional options. See Section 5.4 for more examples.

5.2 Options

Operations (Section 5.2.1)

```
-d, -m, -p, -q, -r, -t, -x
```

Generic modifiers (Section 5.2.2)

```
-c, -s, -S, -T, -v, -V
@file
```

Operation-specific modifiers (Section 5.2.3)

```
-a member
-b member
-i archive
-D, -o, -u
```

Other operations (Section 5.2.4)

```
-format flavor
-M
```

5.2.1 Operations

`-d`

Delete the archive members specified as `file` arguments on the command line from the archive specified on the command line.

If the archive has a symbol table, the table is updated to reflect the new contents of the archive.

`-m`

Move archive members specified as `file` arguments on the command line within the archive specified on the command line.

If a position was specified by `-a`, `-b`, or `-i`, the members are moved to before or after the specified position.

If no position was specified, the specified members are moved to the end of the archive.

If the archive has a symbol table, the table is updated to reflect the new state of the archive.

`-p`

Write to the standard output the contents of the archive members specified as `file` arguments on the command line.

If no members are specified, the contents of all files in the archive are written in the order they appear in the archive.

`-q`

Append the files specified as `file` arguments on the command line to the archive specified on the command line, without checking if the files already exist in the archive. The archive symbol table is updated as necessary.

If the file specified by the argument archive does not already exist, a new archive is created.

`-r`

Replace (add) the files specified as `file` arguments on the command line in the archive specified on the command line, creating a new archive if necessary.

Replacing existing members does not change the order of members within the archive. If one of the specified files does not exist, any existing archive members that match the file name are not changed. New files are added to the end of the archive unless one of the positioning options `-a`, `-b`, or `-i` is specified.

If the archive has a symbol table, the table is updated to reflect the new state of the archive.

`-t`

List the files specified as `file` arguments on the command line in the order they appear in the archive specified as the archive argument, with one file listed per line.

If no files are specified, all the files in the `archive` are listed.

`-x`

Extract to the current directory all archive members specified as `file` arguments on the command line. If no members are specified, extract all members of the archive.

If the file corresponding to an extracted member does not exist, it is created. If the file corresponding to an extracted member does exist, its owner and group are not changed, while its contents are overwritten, and its permissions are set to those entered in the archive.

The access and modification time of the file are set to the time of extraction unless -o is specified.

5.2.2 Generic modifiers

`-c`

Suppress the informational message printed when a new archive is created using `-r` or `-q`.

`-s`

Add an archive symbol table to the archive specified as the `archive` argument on the command line. Invoking the archiver with `-s` alone is equivalent to invoking the archive indexer utility (Chapter 9).

`-S`

Do not generate an archive symbol table.

`-T`

When naming archive members, use only the first 15 characters of the archive member name or command line `file` argument.

`-v`

Provide verbose output.

When used with `-d`, `-m`, `-q`, or `-x`, the archiver generates a file-by-file description of the archive modification being performed, which consists of three white-space separated fields: the option letter, a dash (`-`), and the file name.

When used with `-r`, the archiver displays the description in the same way, but the initial letter is an `a` if the file is added to the archive, or an `r` if the file replaces a file already in the archive.

When used with `-p`, the name of the file enclosed in `<` and `>` characters is written to the standard output preceded by a single newline character and followed by two newline characters. The contents of the named file follow the file name.

When used with `-t`, the archiver displays the following fields:

- File permissions
- Decimal user and group IDs separated by a slash (`/`)
- File size in bytes
- File modification time in format `%b %e %H:%M %Y`
- Name of the file

`-V`

Print the archiver version number and then exit.

`-@file`

Read commands from the specified file.

5.2.3 Operation-specific modifiers

`-a` *`member`*

When used with `-m`, this option causes the archive members specified as `file` arguments on the command line to be moved after the specified archive member.

When used with `-r`, this option causes the files specified as `file` arguments on the command line to be added after the specified archive member.

`-b` *`member`*

When used with `-m`, this option causes the archive members specified as `file` arguments on the command line to be moved before the specified archive member.

When used with `-r`, this option causes the files specified as `file` arguments on the command line to be added before the specified archive member.

`-i` *`archive`*

Equivalent to `-b`.

`-D`

When used with `-r` or `-q`, insert zero values instead of the actual `mtime`, `uid`, and `gid` values, and `0644` instead of file mode, for the archive members specified as `file` arguments on the command line.

This option ensures that checksums on the resulting archives are reproducible when the member contents are identical.

`-o`

Preserve the original modification times of members when extracting them.

`-u`

Conditionally update the archive or extract members.

When used with `-r`, the files specified as `file` arguments on the command line are replaced in the archive if they are newer than their archived versions.

When used with `-x`, the archive members specified as `file` arguments on the command line are extracted only if they are newer than the corresponding files in the file system.

5.2.4 Other operations

`-format` *`flavor`*

Create archives with the specified archive format. The possible values are:

`bsd`	Create BSD-format archives
`gnu`	Alias for `svr4`
`darwin`	Create CSV-format archives
`svr4`	Create SVR4-format archives (Default)

`-M`

Read and execute archiver script commands (Section 5.3) from the standard input.

5.3 Command scripts

If the `-M` option is specified, the archiver utility can be controlled using script commands read from the standard input.

If the standard input is a terminal, the archiver displays the `AR >` prompt before reading a line, and it continues executing even if errors are encountered. If the standard input is not a terminal, the archiver does not display a prompt, and terminates execution on encountering an error.

Each input line contains a single command.

- The words in an input line must be separated by whitespace characters: the first word of the line is the command, while the remaining words are the arguments to the command.
- The command word can be specified in lower or upper case.
- The command arguments can be separated by commas or blanks.
- Empty lines are allowed and ignored.
- Long lines are continued by ending them with the `+` character.
- The `*` and `;` characters start a comment. Comments extend to the end of the line.

When executing an archiver script, the archiver works on a temporary copy of an archive. Changes to the copy are made permanent using the save command.

5.3.1 Commands

The archiver recognizes the following script commands.

`addlib` *`archive`*
`addlib` *`archive`* `(`*`member`* `[, `*`member`*`]...)`

Add the contents of the specified archive to the current archive.

If one or more members are additionally specified, they are added to the current archive.

If no members are specified, the entire contents of the specified archive are added to the current archive.

`addmod` *`member`* `[, `*`member`*`]...`

Add the specified files to the current archive.

`clear`

Discard all the contents of the current archive.

`create` *`archive`*

Create a new archive with the specified name, and make it the current archive.

If the named archive already exists, it is overwritten when the save command is invoked.

`delete` *`member`* `[, `*`member`*`]...`

Delete the specified modules from the current archive.

`directory` *`archive`* `(`*`member`* `[, `*`member`*`]...) [`*`outputfile`*`]`

List the specified members in the archive.

The output format depends on the verbosity level set using the verbose command. Output is sent to the standard output or to the file specified by `outputfile`.

`end`

Exit successfully from the archiver. Any unsaved changes to the current archive are discarded.

`extract` *`member`* `[, `*`member`*`]...`

Extract the specified members from the current archive.

`list`

Display the contents of the current archive in verbose format.

`open` *`archive`*

Open the specified archive and make it the current archive.

`replace` *`member`* `[, `*`member`*`]...`

Replace named members in the current archive with the files specified by arguments member.

The files must be present in the current directory and the specified members must already exist in the current archive.

`save`

Commit all changes to the current archive.

`verbose`

Toggle the verbosity of the directory command.

5.4 Examples

To create a new archive `ex.a` containing three files `ex1.o`, `ex2.o`, and `ex3.o`:

```
hexagon-ar -rc ex.a ex1.o ex2.o ex3.o
```

To add an archive symbol table to an existing archive `ex.a`:

```
hexagon-ar -s ex.a
```

To delete file `ex1.o` from archive `ex.a`:

```
hexagon-ar -d ex.a ex1.o
```

To verbosely list the contents of archive `ex.a`:

```
hexagon-ar -tv ex.a
```

To create a new archive `ex.a` that contains the files `ex1.o` and `ex2.o`, use the `hexagon-ar -M < script` command with the following archiver script:

```
create ex.a              * specify the output archive
addmod ex1.o ex2.o       * add modules
save                     * save pending changes
end                      * exit the utility
```

To create an archive named `lib1.a` with member names that start with `mem` and have the extension, o:

```
hexagon-ar crsU lib1.a mem*.o
```

To print a list with all members in `lib1.a`:

```
hexagon-ar tv lib1.a
```

To delete `mem2.o` from `lib1.a`:

```
hexagon-ar d lib1.a mem2.o
```

To insert `mem2.o` before `mem6.o` in library `lib1.a`:

```
hexagon-ar rb mem6.o lib1.a mem2.o
```

6 Object file symbols

The object file symbols utility lists the symbols in the specified executable files, object files, or object library files.

If no input files are specified, the utility attempts to read from `a.out`.

NOTE: This utility returns zero on success, or greater than zero if an error occurs.

6.1 Command

To start the object files symbols utility from the command line:

hexagon-nm [*option...*] [*file...*]

Where [*option...*] is one or more of the following:

```
--debug-syms      |  -a
--defined-only
--dynamic         |  -D
-e
-f
--format=format   |  -F format
-B  |  -P  |  -o
-g
--help            |  -h
--no-sort         |  -p
--numeric-sort    |  -n  |  -v
--print-armap     |  -s
--print-file-name |  -A
--print-size      |  -S
--radix=radix     |  -t radix
-x
--reverse-sort    |  -r
--size-sort
--undefined-only  |  -u
--version         |  -V
```

6.2 Options

`--debug-syms`
`-a`

Display all symbols, including debugger-only symbols.

`--defined-only`

Display only defined symbols.

`--dynamic`
`-D`

Display only dynamic symbols. This option is meaningful only for shared libraries.

`-e`

Display information only for global and static symbols.

`-f`

Display full output. (Default)

`--format=`*`format`*
`-F` *`format`*

Display output in the specified format. The possible values are:

`bsd`	BSD (Default)
`sysv`	System V
`posix`	POSIX

`-B`

Equivalent to `--format=bsd`.

`-P`

Equivalent to `--format=posix`.

`-o`

If POSIX output was specified (using `-F posix` or `-P`), this option is equivalent to `--radix=o`.

If POSIX output was not specified, it is equivalent to `--print-file-name`.

`-g`

Display information only for global (external) symbols.

`--help`
`-h`

Display the command usage information and then exit.

`--no-sort`
`-p`

Do not sort symbols.

`--numeric-sort`
`-n`
`-v`

Sort symbols numerically by address instead of alphabetically by name.

`--print-armap`
`-s`

For archives, include the index of the archive's members.

`--print-file-name`
`-A`

Write the full pathname or library name of an object on each line, before the rest of the information for a symbol.

If this option is not specified, the utility identifies an input file only once, before its symbols are listed.

`--print-size`
`-S`

Print the size of each symbol instead of its value.

`--radix=`*`radix`*
`-t` *`radix`*

Print numeric values using the specified radix. The possible values are:

- `d` Decimal (Default)
- `o` Octal
- `x` Hexadecimal

`-x`

Write numeric values in hexadecimal (equivalent to `-t x`).

`--reverse-sort`
`-r`

Reverse the order of the sort.

`--size-sort`

Sort symbols by size instead of alphabetically by name.

`--undefined-only`
`-u`

Display only undefined symbols.

`--version`
`-V`

Display the utility version identifier and then exit.

6.3 Output formats

The object file symbols utility can display information in a number of formats, numeric radices, and sort orders.

By default it uses BSD-style output, hexadecimal radix, no alphabetic sorting by name, and no name demangling.

For each symbol listed, the utility presents the following information:

- The library or object name (if `-A` or `--print-file-name` were specified)
- The symbol name
- The symbol type (denoted by a single character):
  - `A` Global absolute symbol
  - `B` Global `bss` symbol (uninitialized data)
  - `C` Common symbol (representing uninitialized data)
  - `D` Global symbol naming initialized data
  - `N` Debugger symbol
  - `R` Read-only data symbol
  - `T` Global text symbol
  - `U` Undefined symbol
  - `V` Weak object
  - `W` Weak reference
  - `a` Local absolute symbol
  - `b` Local `bss` symbol (uninitialized data)
  - `d` Local data symbol
  - `t` Local text symbol
  - `v` Weak object that is undefined
  - `w` Weak symbol that is undefined
  - ? None of the above
- The symbol value
- The symbol size (if applicable)

7 Object file copier

The object file copier utility copies the contents of the specified ELF `infile` object file to a new object file named `outfile`, while performing transformations on the copied file that are specified by the command options.

If the `outfile` argument is not specified, the utility names the copied file `infile`, thus overwriting the original file.

NOTE: This utility returns zero on success, or greater than zero if an error occurs.

7.1 Command

To start the object file copier utility from the command line:

hexagon-llvm-objcopy [*option...*] [*input...*]

Where [*option...*] is one or more of the following:

```
--add-gnu-debuglink=debug-file
--add-section=section=file
--binary-architecture=<value>
--build-id-link-dir=dir
--build-id-link-input=suffix
--build-id-link-output=suffix
--compress-debug-sections=[ zlib | zlib-gnu ]
-decompress-debug-sections
-disable-deterministic-archives  |  -U
-discard-all
--dump-section=section=file
-enable-deterministic-archives  |  -D
-extract-dwo
--globalize-symbol=symbol
--help
--input-target=<value>
-keep-file-symbols
--keep-global-symbol=symbol
--keep-global-symbols=filename
--keep-section=section
--keep-symbol=symbol
-localize-hidden
--localize-symbol=symbol
-only-keep-debug
--only-section=section
--output-target=<value>
--prefix-symbols=prefix
-preserve-dates
```

```
--redefine-sym=old=new
--rename-section=old=new[,flag1,...]
--remove-section=section
--split-dwo=dwo-file
-strip-all
-strip-all-gnu
-strip-debug  |  -g
-strip-dwo
-strip-non-alloc
-strip-sections
--strip-symbol=symbol
-strip-unneeded
--target=<value>
-version
-weaken
--weaken-symbol=symbol
```

7.2 Options

`--add-gnu-debuglink=`*`debug-file`*

Add a `.gnu_debuglink` for the specified debug file.

`--add-section=`*`section=file`*

Make a section named *`section`* with the contents of the specified file.

`--binary-architecture=<`*`value`*`>`

Used when transforming an architecture-less format (such as binary) to another format.

`--build-id-link-dir=`*`dir`*

Set the directory for `--build-id-link-input` and `--build-id-link-output` to *`<dir>`*.

`--build-id-link-input=`*`suffix`*

Hard-link the input to the *`<dir>/xx/xxx<suffix>`* name derived from the Hexagon build ID.

`--build-id-link-output=`*`suffix`*

Hard-link the output to the *`<dir>/xx/xxx<suffix>`* name derived from the Hexagon build ID.

`--compress-debug-sections=[ zlib | zlib-gnu ]`

Compress DWARF debug sections using the specified style.

Supported styles: `zlib-gnu` and `zlib`.

`-decompress-debug-sections`

Decompress DWARF debug sections.

`-disable-deterministic-archives`
`-U`

Disable deterministic mode when copying archives (use real values for UIDs, GIDs, and timestamps).

`-discard-all`Remove all local symbols except file and section symbols.

`--dump-section=`*`section=file`*

Dump the contents of the specified section into the specified file.

`-enable-deterministic-archives`
`-D`

Enable deterministic mode when copying archives (use zero for UIDs, GIDs, and timestamps).

`-extract-dwo`

Remove all sections that are not DWARF `.dwo` sections from `file`.

`--globalize-symbol=`*`symbol`*

Mark the specified symbol as global.

`--help`

Display command usage information and then exit.

`--input-target=<`*`value`*`>`

Format of the input file.

`-keep-file-symbols`

Do not remove file symbols.

`--keep-global-symbol=`*`symbol`*

Convert all symbols except the specified symbol to local.

This option can be repeated to convert all except a set of symbols to local.

`--keep-global-symbols=`*`filename`*

Read a list of symbols from the specified file and run it as if `--keep-global-symbol=`*`symbol`* is set for each symbol.

`filename` contains one symbol per line and can contain comments beginning with `#`. Leading and trailing white spaces are stripped from each line.

This option can be repeated to read symbols from many files.

`--keep-section=`*`section`*

Do not remove specified section.

`--keep-symbol=`*`symbol`*

Do not remove the specified symbol.

`-localize-hidden`

Mark all symbols that have hidden or internal visibility set as local.

`--localize-symbol=`*`symbol`*

Mark the specified symbol as local.

`-only-keep-debug`

Currently ignored. This option is only for compatibility with GNU `objcopy`.

`--only-section=`*`section`*

Remove all but <*`section`*>.

`--output-target=<`*`value`*`>`

Format of the output file.

`--prefix-symbols=`*`prefix`*

Add the specified prefix to the start of every symbol name.

`-preserve-dates`Preserve access and modification timestamps.

`--redefine-sym=`*`old=new`*

Change the symbol name from *`old`* to *`new`*.

`--rename-section=`*`old=new`*`[,`*`flag1`*`,...]`

Rename a section from *`old`* to *`new`*, optionally with specified flags.

Flags supported for GNU compatibility: `alloc`, `load`, `noload`, `readonly`, `debug`, `code`, `data`, `rom`, `share`, `contents`, `merge`, `strings`.

`--remove-section=`*`section`*

Remove the specified section.

`--split-dwo=`*`dwo-file`*

Equivalent to `extract-dwo` on the input file to *`dwo-file`* and `strip-dwo` on the input file.

`-strip-all`

Remove non-allocated sections other than `.gnu.warning*` sections.

`-strip-all-gnu`

Compatible with GNU `objcopy` option, `--strip-all`.

`-strip-debug`
`-g`

Remove all debug information.

`-strip-dwo`

Remove all DWARF `*.dwo` sections from the file.

`-strip-non-alloc`

Remove all non-allocated sections.

`-strip-sections`

Remove all section headers.

`--strip-symbol=`*`symbol`*

Remove the specified symbol.

`-strip-unneeded`

Remove all symbols not required by relocations.

`--target=<`*`value`*`>`

Format of the input and output file.

`-version`

Display the version identifier and then exit.

`-weaken`

Mark all global symbols as weak.

`--weaken-symbol=`*`symbol`*

Mark the specified symbol as weak.

8 Object file viewer

The object file viewer utility displays information about the specified object files or archive files.

NOTE: This utility returns zero on success, or greater than zero if an error occurs.

8.1 Command

To start the object file viewer utility from the command line:

hexagon-llvm-objdump [*option...*] [*file...*]

Where [*option...*] is one or more of the following:

```
-arch=string
-cfg
-disassemble
-dsym=string
-g
-help
-macho
-mattr=a1,+a2,-a3,...
-mno-compound
-no-show-raw-insn
-private-headers
-r
-s
-section-headers
-source
-stats
-t
-triple=string
-unwind-info
-version
```

8.2 Options

`-arch=`*`string`*

Architecture version of the input files. To list the possible values, run the object file viewer utility with the `-version` option.

`-cfg`

Create a CFG for every symbol in the object file and write it to a Graphviz file (MachO only).

`-disassemble`

Display assembler mnemonics for the machine instructions.

`-dsym=`*`string`*

Use the `*.dSYM` file for debug information.

`-g`

Print line information from the debug information (if available).

`-help`

Display command usage information and then exit (for more information, use `-help-hidden`).

`-macho`

Use the MachO-specific object file parser.

`-mattr=`*`a1`*`,`*`+a2`*`,`*`-a3`*`,...`

This option is accepted by the object file viewer utility, but it is ignored.

`-mno-compound`

Disable searching for Hexagon compound instructions.

`-no-show-raw-insn`

When disassembling instructions, do not print the instruction bytes.

`-private-headers`

Display format-specific file headers.

`-r`

Display the relocation entries.

`-s`

Display the contents of each section.

`-section-headers`

Display summaries of the headers for each section.

`-source`

Disassemble with mixed source or assembly code.

`-stats`

Enable statistics output from program (available with Asserts).

`-t`

Display the symbol table.

`-triple=`*`string`*

Architecture target triple to disassemble for.

To list the possible values, run the object file viewer utility with the `-version` option.

`-unwind-info`

Display unwind information.

`-version`

Display the utility version identifier and then exit.

9 Archive indexer

The archive indexer utility updates the archive symbol table in a specified archive file. If the archive file does not contain a symbol table, a new one is added to the file.

NOTE: This utility returns zero on success, or greater than zero if an error occurs.

9.1 Command

To start the archive indexer utility from the command line:

hexagon-ranlib [*option...*] [*archive...*]

Where [*option...*] is one or more of the following:

```
-D
-t
-V
```

9.2 Options

`-D`

For all archive member headers, assign zeros to the `mtime`, `uid`, and `gid` fields, and use mode 0644 for the file mode field.

This option ensures that checksums on the resulting archives are reproducible when member contents are identical.

`-t`

This option is accepted, but it is ignored.

`-V`

Display the utility version identifier and then exit.

10 Object file size

The object file size utility lists the sizes of ELF sections (and optionally the total size) in the specified files. The files can be object files, archive files, or core dumps.

If no input files are specified, the utility attempts to read from a.out.

NOTE: This utility returns zero on success, or greater than zero if an error occurs.

10.1 Command

To start the object file size utility from the command line:

```
hexagon-size [option...] [file...]
```

Where [*option...*] is one or more of the following:

```
--format=format
-A
-B
--help    |  -h
--radix=radix
-d
-o
-x
--version |  -V
```

10.2 Options

`--format=`*format*

Display the output using the specified format (System V or Berkeley). Possible values are `sysv` and `berkeley` (default).

For more information on the display formats, see Section 10.3.

`-A`

Equivalent to `--format=sysv`.

`-B`

Equivalent to `--format=berkeley`.

`--help`
`-h`

Display command usage information and then exit.

`--radix=`*radix*

Display numeric values using the specified radix. The possible values are 8, 10, or 16. The default value is 10.

`-o`

Equivalent to `--radix=8`.

`-d`

Equivalent to `--radix=10`.

`-x`

Equivalent to `--radix=16`.

`--version`
`-V`

Display the utility version identifier and then exit.

10.3 Output formats

The object file size utility can display its output in one of two formats: Berkeley-style or System V-style.

10.3.1 Berkeley-style output

Berkeley-style output consists of an initial header line (labeling the individual fields), followed by one line of output for each ELF object that is either specified directly on the command line or is contained in a specified archive.

Each line contain the following fields:

- Size of the `text` segment in the object
- Size of the `data` segment in the object
- Size of the `bss` segment in the object
- Total size of the object, in decimal or octal format
  - Decimal is used if the specified output radix for numeric values is 10 or 16
  - Octal is used if the radix is 8
- Total size of the object, in hexadecimal format
- Name of the object file

If `--totals` is specified, the output includes an additional line (in the same format) that contains the sum of the respective fields. The file name field for this line contains the string, (`TOTALS`).

10.3.2 System V-style output

System V-style output consists of the following information for each ELF object that is either specified directly on the command line or is contained in a specified archive:

- Name of the object followed by a colon character.
- One header line containing the field labels for the subsequent lines.
- One line for each section contained in the object. Each line has three fields:
  - Section name
  - Section size in the specified radix
  - Section address in the selected radix
- An additional line with the section name set to `Total`, and a `size` field set to the sum of all the reported section sizes.

10.4 Examples

Section sizes for `/bin/ls`:

```
$ hexagon-size /bin/ls
text       data       bss        dec        hex        filename
20975      540        392        21907      5593        /bin/ls
```

Sizes and total for `/bin/ls` and `/bin/dd` in hexadecimal:

```
$ hexagon-size -tx /bin/ls /bin/dd
text       data       bss        dec        hex        filename
0x51ef     0x21c      0x188      21907      5593        /bin/ls
0x3df5     0x170      0x200      16741      4165        /bin/dd
0x8fe4     0x38c      0x388      38648      96f8       (TOTALS)
```

Section sizes for `/bin/ls` in System V format:

```
$ hexagon-size -A /bin/ls
/bin/ls  :
section            size       addr
.interp            21         4194704
.note.ABI-tag      24         4194728
.hash              624        4194752
.dynsym            2088       4195376
.rodata            1472       4216448
.data              80         5267456
.eh_frame          1624       5267536
.dynamic           384        5269160
.ctors             16         5269544
.dtors             16         5269560
.jcr               8          5269576
.got               576        5269584
.bss               528        5270176
.comment           686        0
Total              27110
```

11 Object file strings

The object file strings utility displays the printable character strings contained in the specified files. The files can be object files, archive files, or arbitrary data files.

To control the utility output, specify the minimum length of the strings to be displayed.

If no input files are specified, the utility attempts to read from the standard input.

NOTE: This utility returns zero on success, or greater than zero if an error occurs.

11.1 Command

To start the object file strings utility from the command line:

hexagon-llvm-strings [*option...*] [*file...*]

Where [*option...*] is one or more of the following:

```
--all                 |  -a
--bytes=number        |  -n number  |  -number
--help                |  -h
--print-file-name     |  -f
--radix=radix         |  -t radix   | -o
--version             |  -v
```

11.2 Options

`--all`
`-a`

For ELF objects, scan the entire file for printable strings.

`--bytes=`*number*
`-n` *number*
`-`*number*

Print a contiguous character sequence only if it is at least the specified number of characters long. The default length is four characters.

`--help`
`-h`

Display command usage information and then exit.

`--print-file-name`
`-f`

Print the name of the file before each string.

`--radix=`*radix*
`-t` *radix*

Print the file-relative offset of the string before each string. The offset is displayed in the specified radix.

The possible values are:

- `d` Decimal
- `o` Octal
- `x` Hexadecimal

`-o`

Equivalent to `-t o`.

`--version`
`-v`

Display the utility version identifier and then exit.

11.3 Examples

Strings in `/bin/ls`:

```
$ hexagon-llvm-strings /bin/ls
```

Strings in all sections of `/bin/ln`:

```
$ hexagon-llvm-strings -a /bin/ln
```

Strings in all sections of `/bin/cat`, prefixed with the file name and file-relative offset:

```
$ hexagon-llvm-strings -a -f -t x /bin/cat
```

12 Object file stripper

The object file stripper utility removes information from the specified files. The files can be object files or archive files.

NOTE: This utility returns zero on success, or greater than zero if an error occurs.

12.1 Command

To start the object file stripper utility from the command line:

hexagon-strip [*option...*] [-x] *file...*

Where [*option...*] is one or more of the following:

```
--discard-all                 |  -x
--discard-locals              |  -X
--help                        |  -h
--input-target=objformat      |  -I objformat
--keep-symbol=symbolname      |  -K symbolname
--only-keep-debug
--output-file=filename        |  -o filename
--output-target=objformat     |  -O objformat
--preserve-dates              |  -p
--remove-section=sectionname  |  -R sectionname
--strip-all                   |  -s
--strip-debug  |  -d  |  -g   |  -S
--strip-unneeded
--strip-symbol=symbolname     |  -N symbol
--version                     |  -V
--wildcard                    |  -w
```

12.2 Options

`--discard-all`
`-x`

Remove all non-global symbols from the object file.

`--discard-locals`
`-X`

Remove compiler-generated local symbols from the object file.

`--help`
`-h`

Display command usage information and then exit.

`--input-target=`*`objformat`*
`-I` *`objformat`*

This option is accepted for compatibility with the GNU object file stripper, but it is ignored.

`--keep-symbol=`*`symbolname`*
`-K` *`symbolname`*

Keep the specified symbol even if it would otherwise be stripped. This option can be specified more than once.

`--only-keep-debug`

Remove all information from the object file except for the debugging information.

`--output-file=`*`filename`*
`-o` *`filename`*

Write the output file to a new file with the specified name. The default operation is to overwrite the input file.

`--output-target=`*`objformat`*
`-O` *`objformat`*

Specify file format of the stripped object file. The possible values are:

`ETF_ELF`	ELF object (Default)

`--preserve-dates`
`-p`

Set the access and modification times of the object file to the same as those of the input.

`--remove-section=`*`sectionname`*
`-R` *`sectionname`*

Remove the specified section from the object file.

`--strip-all`
`-s`

Remove all symbols from the object file.

`--strip-debug`
`-d`
`-g`
`-S`

Remove debugging information from the object file.

`--strip-unneeded`

Remove all symbols that are not needed for relocation processing.

`--strip-symbol=`*`symbolname`*
`-N` *`symbol`*

Remove the specified symbol even if it would otherwise be kept. This option can be specified more than once.

`--version`
`-V`

Display the utility version identifier and then exit.

`--wildcard`
`-w`

Use shell-style patterns to specify symbols. The following meta-characters are recognized in patterns:

- `!` If this is the first character of the pattern, invert the sense of the pattern match
- `*` Match any string of characters in a symbol name
- `?` Match zero or one character in a symbol name
- `[` Mark the start of a character class
- `\` Remove the special meaning of the next character in the pattern
- `]` Mark the end of a character class

13 C++ filter

The C++ filter utility translates encoded C++ symbol names to human-readable form. This utility can be used in two ways:

- As a filter—reading encoded names from the standard input, and writing the translated names to the standard output.
- As a command-line translator—reading the encoded names that are passed as command arguments, and writing the translated names to the standard output.

NOTE: This utility returns zero on success, or greater than zero if an error occurs.

13.1 Command

To start the C++ filter utility from the command line:

`hexagon-c++filt` [*`option...`*] [*`encoded-names...`*]

Where [*`option...`*] is one or more of the following:

```
--format=<value>
-help  |  -help-hidden
-strip-underscore
-types
-version
```

13.2 Options

`--format=<`*`value`*`>`

Select a decoration style. The possible values are:

`auto`	Automatically detect the style
`gnu`	GNU (itanium) style

`-help`
`-help-hidden`

Display the command usage information and available options.

Use `-help-hidden` to display more options.

`-strip-underscore`

Strip the leading underscore.

`-types`

Attempt to demangle types as well as function names.

`-version`

Display the version of this program.

14 Address converter

The address converter utility translates the specified program addresses to their corresponding source file names and line numbers in an object file.

The addresses must be in hexadecimal format. If no addresses are specified, the utility attempts to read them from the standard input.

By default, the utility uses `a.out` as the ELF object file. A different object file can be specified in a command argument.

NOTE: This utility returns zero on success, or greater than zero if an error occurs.

14.1 Command

To start the address converter utility from the command line:

`hexagon-addr2line` [*`option...`*] [*`address...`*]

Where [*`option...`*] is one or more of the following:

```
--demangle        |  -C
--exe=pathname    |  -e pathname
--functions       |  -f
--help            |  -H
--section=sectionname
--target=target   |  -b target
--version         |  -V
```

14.2 Options

`--demangle`
`-C`

Demangle C++ names.

`--exe=`*`pathname`*
`-e` *`pathname`*

Use the specified ELF object to translate addresses. The default object file name is `a.out`.

`--functions`
`-f`

Display function names in addition to the file and line number information.

`--help`
`-H`

Display command usage information and then exit.

`--section=`*`sectionname`*

Treat address values as offsets into the specified section.

`--target=`*`target`*
`-b` *`target`*

This option is accepted for compatibility with the GNU address converter, but it is ignored.

`--version`
`-V`

Display the utility version identifier and then exit.

14.3 Output formats

For each address:

- If `-f` is not specified, the utility prints the corresponding file name and line number on a single line.
- If `-f` is specified, the utility prints the function name corresponding to the address on one line, and the associated file name and line number on the following line.

The file name and line number are printed as *`filename`*`:`*`linenumber`*.

If a file or function name cannot be determined, the address converter prints a question mark in its place. If a line number cannot be determined, a zero is printed in its place.

14.4 Examples

Map the `080483c4` address in the default executable `a.out` to a source file name and line number:

```
% addr2line 080483c4
```

Map address `080483c4` in an executable `helloworld`:

```
% addr2line -e helloworld 080483c4
```

Use the address converter as a filter, reading addresses from the standard input:

```
% addr2line
```

Print the function name for an address together with its source file and line number:

```
% addr2line -f 080483c4
```

15 DWARF interpreter

The DWARF interpreter is similar to the address converter. Like the address converter, it translates the specified program addresses to their corresponding source file names and line numbers in an object file. But it also does the following:

- Covers data structures and provides memory layout information
- Does the processing in batch mode, thus allowing for much faster translation of a large set of addresses—the address converter (addr2line) translates one address at a time.

The DWARF interpreter (`addr2src`) has a limited ability to correct faulty DWARF information. The intended use of this utility is to script the processing of large traces of addresses, where it can handle mixed text and data.

15.1 Command

To start the DWARF interpreter utility from the command line:

`hexagon-addr2src` *`file`* [*`option...`*]

Where [*`option...`*] is one or more of the following:

```
--batch-file=filename
--debug
--get-padding
--help  |  -h
--lookup=<address>
--size=<uint>
--version
```

15.2 Options

`-- batch-file=`*`filename`*

List of addresses in hexadecimal format with a 0x prefix; one address per line with no separators.

`--debug`

Verbalize the translation mechanism. This option is intended only for debugging of the utility.

`--get-padding`

Find the padding present in the data structures of the ELF.

`--help`
`-h`

Display command usage information and then exit.

`--lookup=<address>`

Look up a variable <`address`> in the debug information or symbol table, and output the name and members at that location.

`--size=<uint>`

Range of addresses to look up, starting at `--lookup` (the default value is 1).

`--version`

Display the utility version identifier and then exit.

15.3 Output formats

The address, mangled symbol name, source file name and line number, column, discriminator, and inline stack location are printed as:

address,*symbolname*,*filename*:*linenumber*,*column*,*discriminator*,*inline_stack_location*

If DWARF information is not found for a specified address, only the *address* itself is printed.

15.4 Examples

Translate the list of addresses in the `batch.txt` file from an executable `helloworld`:

```
% addr2src helloworld --batch-file batch.txt
```

Translate `1024` consecutive addresses starting at address `0x1000` from an executable `helloworld`:

```
% addr2src helloworld -lookup=0x00001000 -size=1024
```

Get padding information for `1024` consecutive addresses starting at address `0x1000` from an executable `helloworld`:

```
% addr2src helloworld -lookup=0x00001000 -size=1024 --get-padding
```

16 ELF file viewer

The ELF file viewer utility displays information about the contents of the specified ELF object files. If no input files are specified, the utility attempts to read from the standard input.

NOTE: This utility returns zero on success, or greater than zero if an error occurs.

16.1 Command

To start the ELF file viewer utility from the command line:

hexagon-llvm-readelf [*option...*] [*input...*]

Where [*option...*] is one or more of the following:

```
-demangle  |  -C
-dyn-symbols
-dynamic-table
-elf-section-groups  |  -g
-expand-relocs
-file-headers  |  -h
-help
-needed-libs
-program-headers
-relocations         |  -r
-section-data        |  sd
-section-relocations  |  -sr
-section-symbols  |  -st
-sections  |  -s
-symbols  |  -t
-version
-unwind  |  -u
```

16.2 Options

`-demangle`
`-C`

Print demangled symbol names in the output.

`-dyn-symbols`

Display the dynamic symbol table (only for ELF object files).

`-dynamic-table`

Display the ELF `.dynamic` section table (only for ELF object files).

`-elf-section-groups`
`-g`

Display section groups (only for ELF object files).

`-expand-relocs`

When used with `-relocations`, display each relocation in an expanded multi-line format.

`-file-headers`
`-h`

Display the file headers.

`-help`

Display the command usage information and then exit.

`-needed-libs`

Display the required libraries (only for ELF object files).

`-program-headers`

Display the program headers.

`-relocations`
`-r`

Display the relocation entries in the file.

`-section-data`
`sd`

When used with `-sections`, display section data for each section shown.

`-section-relocations`
`-sr`

When used with `-sections`, display relocations for each section shown.

`-section-symbols`
`-st`

When used with `-sections`, display symbols for each section shown.

`-sections`
`-s`

Display all sections.

`-symbols`
`-t`

Display the symbol table.

`-version`

Display the utility version identifier and then exit.

`-unwind`
`-u`

Display unwind information.

A Acknowledgments

This document includes content derived from the LLVM Project (under the terms of the LLVM Release License) and the ELF Tool Chain Project (under the terms of the BSD License).

A.1 LLVM Release License

The description of the Hexagon object file viewer utility is subject to the conditions and disclaimers defined in the LLVM Release License:

llvm.org/releases/3.6.0/LICENSE.TXT

A.2 BSD License

The descriptions of the Hexagon utilities listed below are subject to the following conditions and disclaimers.

Archiver	Copyright (c) 2007, 2009-2012, Joseph Koshy
Object file symbols	Copyright (c) 2007, Hyogeol Lee
Object file copier	Copyright (c) 2008-2009, 2011, Joseph Koshy
Archive indexer	Copyright (c) 2007,2009-2012, Joseph Koshy
Object file size	Copyright (c) 2007, S.Sam Arun Raj Copyright (c) 2008, 2011, Joseph Koshy
Object file strings	Copyright (c) 2007, S.Sam Arun Raj
Object file stripper	Copyright (c) 2011, Joseph Koshy
C++ filter	Copyright (c) 2011, Kai Wang
Address converter	Copyright (c) 2009,2010, Joseph Koshy
ELF file viewer	Copyright (c) 2009,2011, Joseph Koshy

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.