CodeAurora FORUM

Hexagon Binutils

A GNU Manual

80-N2040-1 Rev. F

September 1, 2011

Submit technical questions to:
https://www.codeaurora.org

**Authors Roland H. Pesch, Steve Chamberlain, Jay Fenlason, Jeffrey M. Osier,
Dean Elsner, Cygnus Support, Ben Leslie, QUALCOMM Incorporated, Qualcomm Innovation Center, Inc.**

**Published by
Code Aurora Forum
48377 Fremont Blvd., Ste. 117
Fremont, CA 94538
U.S.A.**

Contents

1 Introduction 11

1.1 Overview 11
1.2 Features 12
1.3 Using the tools 13
1.4 Processor version support 14
1.5 Using the document 14
1.6 Notation 15
1.7 System requirements 15
1.8 Feedback 15

2 Assembler 16

2.1 Overview 16
2.2 Using the assembler 17
2.2.1 Starting the assembler 18
2.2.2 Assembler options 19
2.2.2.1 Assembler information 20
2.2.2.2 Input and output files 21
2.2.2.3 Listing 22
2.2.2.4 Listing configuration 23
2.2.2.5 Assembler messages 23
2.2.2.6 Code generation 24
2.2.2.7 Symbol handling 25
2.2.2.8 Processor version 25
2.2.3 Screen messages 26
2.2.4 Warning and error messages 26
2.2.5 List files 26
2.3 Basic syntax 27
2.3.1 Character set 27
2.3.2 Symbols 27
2.3.3 Whitespace 27
2.3.4 Keywords 28
2.3.5 Directives 28
2.3.6 Numbers 28
2.3.7 Characters 29
2.3.8 Expressions 31

2.3.9 Operators 32
2.3.10 Statements 33
2.3.11 Comments 34
2.4 Sections and relocation 35
2.4.1 Code and data sections 36
2.4.2 Block storage section 37
2.4.3 Global sections 38
2.4.4 Hexagon processor memory sections 38
2.4.5 User-defined sections 39
2.4.6 Section memory assignment 40
2.4.7 Sub-sections 40
2.4.8 Assembler internal sections 41
2.5 Symbols 42
2.5.1 Symbol names 42
2.5.2 Temporary symbols 42
2.5.3 Symbol assignment 42
2.5.4 Labels 43
2.5.5 Local labels 43
2.5.5.1 Symbolic local labels 43
2.5.5.2 Numeric local labels 43
2.5.6 Dot symbol 44
2.5.7 Symbol attributes 45
2.5.7.1 Symbol value 45
2.5.7.2 Symbol type 45
2.5.7.3 Symbol size 45
2.5.7.4 Symbol scope 46
2.5.7.5 Symbol visibility 46
2.5.7.6 Symbol descriptor 46
2.6 Directives 47
2.6.1 .2byte 49
2.6.2 .3byte 49
2.6.3 .4byte 50
2.6.4 .abort 50
2.6.5 .align 50
2.6.6 .altmacro 51
2.6.7 .ascii 52
2.6.8 .asciz 52
2.6.9 .balign 53
2.6.10 .block 53
2.6.11 .byte 54
2.6.12 .common 54
2.6.13 .data 54
2.6.14 .desc 55

2.6.15 .double 55
2.6.16 .eject 55
2.6.17 .else 55
2.6.18 .elseif 55
2.6.19 .end 56
2.6.20 .endif 56
2.6.21 .endm 56
2.6.22 .endr 56
2.6.23 .equ 56
2.6.24 .equiv 56
2.6.25 .err 57
2.6.26 .exitm 57
2.6.27 .falign 57
2.6.28 .file 57
2.6.29 .fill 58
2.6.30 .float 58
2.6.31 .global 58
2.6.32 .half 58
2.6.33 .hidden 59
2.6.34 .hword 59
2.6.35 .if 59
2.6.36 .include 60
2.6.37 .int 60
2.6.38 .internal 60
2.6.39 .irp 61
2.6.40 .irpc 61
2.6.41 .lcommon 62
2.6.42 .list 62
2.6.43 .loc 63
2.6.44 .long 63
2.6.45 .macro 64
2.6.46 .noaltmacro 66
2.6.47 .nolist 66
2.6.48 .octa 66
2.6.49 .org 66
2.6.50 .p2align 67
2.6.51 .protected 67
2.6.52 .psize 68
2.6.53 .purgem 68
2.6.54 .quad 68
2.6.55 .reloc 68
2.6.56 .rept 69
2.6.57 .sbttl 69

2.6.58 .section ... 69
2.6.59 .set ... 70
2.6.60 .short ... 70
2.6.61 .single ... 70
2.6.62 .size ... 71
2.6.63 .sleb128 ... 71
2.6.64 .skip ... 71
2.6.65 .space ... 71
2.6.66 .stab ... 72
2.6.67 .string ... 73
2.6.68 .string8 ... 73
2.6.69 .string16 ... 73
2.6.70 .string32 ... 73
2.6.71 .string64 ... 74
2.6.72 .symver ... 74
2.6.73 .text ... 75
2.6.74 .title ... 75
2.6.75 .type ... 76
2.6.76 .uleb128 ... 76
2.6.77 .word ... 77
2.7 Preprocessing ... 77
2.8 Processor-specific features ... 78

3 Linker ... 79

3.1 Overview ... 79
3.2 Using the linker ... 80
3.2.1 Starting the linker ... 80
3.2.2 Linker options ... 82
3.2.3 Environment variables ... 107
3.3 Linker scripts ... 108
3.3.1 Script example ... 109
3.3.2 Script commands ... 110
3.3.3 Semicolons ... 112
3.3.4 Comments ... 112
3.3.5 Strings ... 112
3.3.6 Expressions ... 112
3.3.6.1 Integers ... 113
3.3.6.2 Symbol names ... 113
3.3.6.3 Location counter ... 114
3.3.6.4 Operators ... 114
3.3.6.5 Evaluation ... 114
3.3.6.6 Predefined functions ... 115

3.3.7 Symbol assignment ... 119
3.3.7.1 PROVIDE ... 120
3.3.7.2 PROVIDE_HIDDEN ... 120
3.3.7.3 Source code access ... 120
3.3.8 MEMORY command ... 121
3.3.9 SECTIONS command ... 123
3.3.9.1 Output section descriptions ... 124
3.3.9.2 Input section descriptions ... 125
3.3.9.3 Input section archive files ... 126
3.3.9.4 Input section wildcards ... 126
3.3.9.5 Input section common symbols ... 128
3.3.9.6 Input section garbage collection ... 129
3.3.9.7 Output section data ... 129
3.3.9.8 Output section fill ... 129
3.3.9.9 Output section constructors ... 130
3.3.9.10 Output section discarding ... 130
3.3.9.11 Output section attributes ... 131
3.3.9.12 Overlay descriptions ... 134
3.3.10 ENTRY command ... 136
3.3.11 PHDRS command ... 136
3.3.12 VERSION command ... 138
3.3.12.1 Source file symbol binding ... 140
3.3.12.2 Multiple function versions ... 140
3.3.12.3 Multiple symbol definitions ... 140
3.3.12.4 External symbol binding ... 141
3.3.13 Other commands ... 141
3.4 Processor-specific memory layout ... 144
3.4.1 Hexagon processor memories ... 144
3.4.2 Address mapping ... 145
3.4.3 MMU programming ... 148
3.5 Binary file access ... 148
3.5.1 Functional overview ... 149
3.5.2 Information loss ... 149
3.5.3 Canonical object file format ... 150
3.6 Processor-specific features ... 151

4 Utilities 153

4.1 Overview 153
4.2 Using the utilities 154
4.3 Archiver 155
4.3.1 Archiver options 156
4.3.1.1 Command options 156
4.3.1.2 Modifier options 158
4.3.2 Archiver command scripts 159
4.4 Object file symbols 161
4.5 Object file copier 165
4.6 Object file viewer 173
4.7 Archive indexer 178
4.8 Object file size 178
4.9 Object file strings 180
4.10 Object file stripper 181
4.11 C++ filter 184
4.12 Address converter 186
4.13 ELF file viewer 187

A Acknowledgments 191

A.1 Overview 191
A.2 Assembler 191
A.3 Linker 193
A.4 Utilities 193

B License Statements 194

B.1 History 194
B.2 Free software 195
B.3 Free software needs free documentation 195
B.4 GNU GENERAL PUBLIC LICENSE 197
B.4.1 Preamble 197
B.4.2 How to apply these terms to your new programs 202
B.5 GNU FREE DOCUMENTATION LICENSE 203
B.5.1 ADDENDUM: How to use this license for your documents 209

Figures

Figure 1-1 Using the tools 13
Figure 2-1 Using the assembler 17
Figure 2-2 Section type relocation 37
Figure 3-1 Using the linker 80
Figure 3-2 TCM memory map 146

Tables

Table 2-1 Code and data sections 36
Table 2-2 Processor global data sections 38
Table 2-3 Hexagon processor-specific memory sections 38
Table 2-4 User-defined sections 39
Table 3-1 Linker script commands 111
Table 3-2 Processor memories 144
Table 3-3 Hexagon processor address symbols 147
Table 4-1 Software development utilities 153

1 Introduction

1.1 Overview

Binutils (short for *binary utilities*) are a set of software tools which manipulate object code. They are used with compilers, debuggers, and profilers to support software development for the Qualcomm® Hexagon™ processor.

Binutils includes the following tools:

- Assembler
- Linker
- Archiver
- Object file symbol lister
- Object file copier
- Object file viewer
- Archive indexer
- Object file section size lister
- Object file string lister
- Object file stripper
- C++ filter
- Address converter
- ELF file viewer

This document describes the Hexagon processor-specific versions of these tools. The compilers, debuggers, and profilers are described in separate documents.

NOTE The binutils for the Hexagon processor are based on GNU Binutils 2.20. For more information see http://www.gnu.org.

1.2 Features

Assembler

- Macro processor
- Conditional assembly
- Constant expression folding
- Local labels

Linker

- Linker command language files
- Memory partitioning
- Comprehensive link map reports

Utilities

- Object file library management (ar, ranlib)
- Object file management (copy, strip)
- Object file properties (nm, objdump, size, etc.)
- ELF file viewing (readelf)

1.3 Using the tools

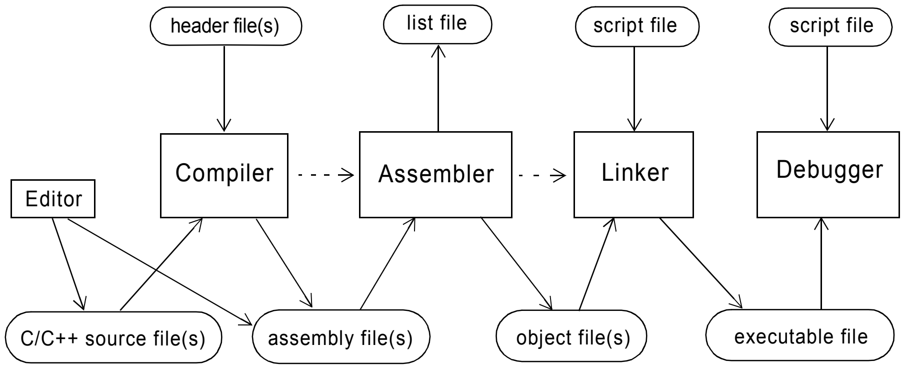

Figure 1-1 Using the tools

1. A C, C++, or assembly language *source file* is first created with a text editor.
2. The C or C++ compiler translates a C or C++ source file into an *output file* which (depending on whether the compiler invokes the assembler or linker) may be an assembly source file, object file, or executable file. The compiler optionally inputs *header files* which define library functions.
3. The assembler translates an assembly language source file into an *object file*. It optionally produces a *list file*.

   Object files contain the assembled object code, information used by the linker to create executable programs, and (optionally) symbolic information for use by the debugger.

   List files list the assembly language source text along with information on how it was translated into object code.
4. The linker links object files into executable programs. It inputs one or more object files and a *script file*, and outputs an *executable file*.

   Linker script files specify how the input files are to be linked.
5. The debugger controls the execution of programs. It inputs an executable file and optional *script file*.

   Debugger script files contain debugger commands, and are used to automate debug tasks.

1.4 Processor version support

The software development tools support version V2, V3, and V4 of the Hexagon processor:

- The compiler and assembler support command options (`-mv2`, `-mv3`, `-mv4`) that specify the processor version for which the tool will generate output files.
- The linker supports the same command options to specify the version of object file it will link. Attempts to link object files with different processor versions will result in an error message.
- The other tools automatically determine the processor version of an input object file from information stored in the file.

For more information on these (and related) command options, see the Assembler and Linker chapters in this document.

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

1.5 Using the document

This document is designed to serve as a reference for experienced C programmers with assembly language experience.

The document contains four chapters and two appendices:

- Chapter 1, *Introduction*, presents an overview of the tools and the document.
- Chapter 2, *Assembler*, describes the assembler.
- Chapter 3, *Linker*, describes the linker.
- Chapter 4, *Utilities*, describes the archive and file utilities.
- Appendix A, *Acknowledgments*, lists the people who have contributed to the development of the tools.
- Appendix B, *License Statements*, lists the license statements for this document.

1.6 Notation

This document uses italics for terms and document names:

executable object file

Hexagon V4 Programmer's Reference Manual

Courier font is used for computer text:

```
.Ltext0:
          .section     .rodata
          .p2align 3
          .string      "hello, world\n"
```

The following notation is used to define command syntax:

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

1.7 System requirements

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

1.8 Feedback

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

`https://support.cdmatech.com`

2 Assembler

2.1 Overview

The assembler translates the Hexagon processor's assembly language into object code. Assembly language is stored in source files, which are text files usually created with a text editor. Object code is stored in object files, which are data files produced by the assembler. Object files are stored in ELF format.

The assembler supports features such as macros, conditional text, and include files.

This chapter covers the following topics:

- Using the assembler
- Basic syntax
- Sections and relocation
- Symbols
- Directives
- Preprocessing
- Processor-specific features

2.2 Using the assembler

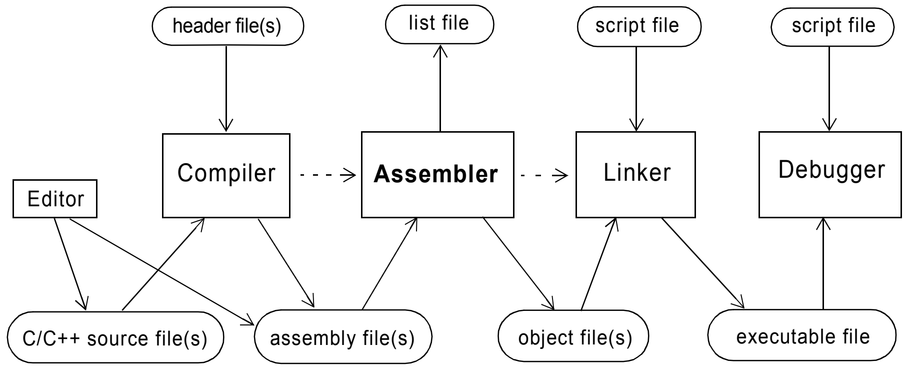

Figure 2-1 Using the assembler

The assembler translates assembly language source files into object files. It also optionally produces list files.

Source files are usually created with a text editor. Word processors should not be used unless they can save files in plain-text (or “text-only”) format.

Object files are produced by the assembler. They contain the assembled object code, information used by the linker to create executable programs, and (optionally) symbolic information for use by the debugger. Object files are input to the linker.

List files are optionally produced by the assembler. They include the assembly language source text along with information on how it was assembled into object code. List files are useful for reading and debugging assembly language programs.

Source and object files can contain more than one *section*, where a section is defined as a contiguous unit of code or data.

NOTE The assembler is primarily used to assemble the output from the C compiler.

2.2.1 Starting the assembler

To start the assembler from a command line, type:

```
hexagon-as [option...] [input_file...]
```

The assembler accepts zero or more source files as the assembly language input. The file names must include the name extension if any.

- If multiple source files are specified, their contents are concatenated in the order of the specified file names and treated as the contents of a single source file.
- If no source file is specified, the assembler reads from the standard input.
- If the source file is empty, the assembler generates an empty object file.

Command switches are used to control various assembler options. A switch consists of one or two dash characters followed by a switch name and optional parameter. Note that switch names are case-sensitive. Switches must be separated by at least one space.

The assembler always produces a single object file. If you don't specify an output file name (with the `-o` switch), the default object file name is `a.out`.

Option files

Assembler command arguments can be specified in a text file rather than on the command line. The file is specified on the command line as the argument of the special command option `@`. For example:

```
hexagon-as @myoptions
```

Here the file named *myoptions* contains the command arguments for the `hexagon-as` command. The file contents are inserted into the command line in place of the specified `@`*file* option.

NOTE Command arguments stored in argument files must be delimited by whitespace characters.

The `@`*file* option can appear in argument files – it is processed recursively.

Option help

To list the available command options, type:

```
hexagon-as --help
```

The assembler displays on the screen the proper command line syntax, followed by a list of the available command options.

2.2.2 Assembler options

Assembler options are used to control various assembler features from the command line.

Many options have alternate abbreviated switches defined for ease of use. These long and short forms, listed below as alternatives, are functionally equivalent.

The standard option names can also be truncated, as long as you specify enough of the option name to uniquely identify the option.

NOTE Option names can be prefixed with either "`-`" or "`--`".

The options are specified by the command switches listed below.

```
-h | --help
--alternate
--target-help
--version
--v
--statistics
--mfalign-info

-o filename
--MD filename
-I pathname
@filename

-a[lhsncdg][=filename]
--listing-lhs-width width
--listing-lhs-width2 width
--listing-rhs-width width
--listing-cont-lines lines

-W | --no-warn
--warn
--mfalign-warn
--fatal-warnings
-D

--gstabs
--gstabs+
--gdwarf2
-R
-Z
-G size

--mno-extender
--mno-jumps
--mno-jumps-long
--mno-pairing
--mno-pairing-duplex
--mno-pairing-branch
--mpairing-info
```

```
--defsym name=value
--strip-local-absolute
-L | --keep-locals
-f
--reduce-memory-overheads

-march archname
-mcpu archname
-mv2
-mv3
-mv4
```

2.2.2.1 Assembler information

`-h`
`--help`
Display assembler command options and exit.

`--alternate`
Enable additional features in macro processor (Section 2.6.6).

`--target-help`
Display Hexagon processor-specific assembler command options and exit.

`--version`
Display assembler build version and exit.

`-v`
Display assembler build version.

`--statistics`
Display the amount of memory (in bytes) and the execution time (in seconds) used by the assembler while assembling the input files.

`--mfalign-info`
Display statistics on `.falign` usage (Section 2.6.27).

2.2.2.2 Input and output files

`-o` *`filename`*

Name the object file with the specified name. The default name is `a.out`.

`--MD` *`filename`*

Write include dependencies to a file with the specified name. The resulting file is suitable for use as a makefile with the `make` utility.

`-I` *`pathname`*

Cause the assembler to search for include files (Section 2.6.36) in the directory with the specified path.

If no search path is specified the assembler assumes that all include files are stored in the current directory. If multiple search paths are specified (with multiple `-I` options), the assembler will search them in left-to-right order.

`@`*`filename`*

Use command arguments stored in the specified file (Section 2.2.1).

NOTE No space can appear between the at sign (`@`) and *filename*.

2.2.2.3 Listing

`-a``[`**`lhsncdg`**`] [`**`=`***`filename`*`]`

Generate an assembly listing (Section 2.2.2.3) with the specified list options.

Multiple list options can be specified in a single command-line option by combining the list option letters (e.g., `-aln`).

The default list options are `'lhs'`.

`l`

Generate an assembly language listing as part of the listing.

`h`

Generate a high-level language listing as part of the listing.

`h` must be used with `l` (e.g., `-ahl`) to generate a listing, and the original C source program must have been compiled with the `-g` option.

`s`

Generate a symbol table as part of the listing.

`n`

Disable forms processing.

`n` must be used with `l` (e.g., `-anl`) to generate a listing.

`c`

Omit false conditionals from the listing.

False conditionals are source lines which the assembler skips over because of a false conditional directive (`.if`, `.ifdef`, `.else`, etc.).

`c` must be used with `l` (e.g., `-acl`) to generate a listing.

`d`

Omit debugging directives from the listing.

`g`

Include general information about the assembly (including the assembler version, command options specified, and time stamp).

`-a=`*`filename`*

Write the listing to a file with the specified name. The default listing is written to the standard output.

The file name must be specified at the end of the `-a` option (e.g., '`-aln=foo`').

NOTE The listing format and appearance can be further controlled using the listing configuration options (Section 2.2.2.4) and the assembler directives `.list`, `.nolist`, `.psize`, `.eject`, `.title`, and `.sbttl` (Section 2.6).

2.2.2.4 Listing configuration

`--listing-lhs-width` *`width`*

Specify the maximum number of words displayed on each line in the object code section of a listing. The default value is 1.

If a given line generates more words than can fit on a line with the specified width, the additional words appear on subsequent lines of the listing (which are known as *continuation* lines).

`--listing-lhs-width2` *`width`*

Specify the maximum number of words displayed on continuation lines in the object code section of a listing. The default value is the value of `--listing-lhs-width`.

If the specified value is less than the value of `--listing-lhs-width`, then this option is ignored.

`--listing-rhs-width` *`width`*

Specify the maximum number of characters in the lines of the assembly source file. The default value is 100.

`--listing-cont-lines` *`lines`*

Specify the maximum number of continuation lines generated for a single line in the object code section of a listing.

If a given line generates more words than can fit on the specified number of continuation lines, the additional lines are truncated.

NOTE These options are used with the list file options (Section 2.2.2.3).

2.2.2.5 Assembler messages

`-W`
`--no-warn`

Suppress warning messages generated by the assembler.

`--warn`

Do not suppress warning messages generated by the assembler. This option is enabled by default.

`--mfalign-warn`

Generate a warning message whenever the `.falign` directive (Section 2.6.27) causes a NOP instruction to be inserted into the object code (either in an existing instruction packet, or in its own packet).

`--fatal-warnings`

Treat warning messages as error messages (Section 2.2.4).

`-D`

Generate assembler debugging messages.

2.2.2.6 Code generation

`--gstabs`
: Generate *stabs* debugging information for each assembler line.

`--gstabs+`
: Generate *stabs* debugging information (with GNU extensions) for each assembler line.

`--gdwarf2`
: Generate DWARF 2 debugging information for each assembler line.

`-R`
: Generate object file which appends all data section data to the text section (Section 2.4.1). The length of the object file data section is zero.

`-Z`
: Generate object file even after one or more error messages are generated. The assembler normally does not generate an object file after an error message.

  When this option is used the assembler displays a message listing the number of errors and warnings detected, and indicates that the output file is invalid.

`-G` *`size`*
: Allocate common symbols in the processor global data section (Section 2.4.3) if they are less than or equal to the specified size. The size is expressed in bytes. The default is 8.

  Common symbols are declared with the assembler directives `.common` and `.lcommon` (Section 2.6).

NOTE The `-G` option is also supported in the linker (Section 3.2.2).

`--mno-extender`
: Disable the use of constant extenders.

`--mno-jumps`
: Disable automatic extension of branch instructions.

`--mno-jumps-long`
: Disable automatic extension of branch instructions whose range was not shortened due to pairing of packet instructions.

`--mno-pairing`
: Disable pairing of packet instructions.

`--mno-pairing-duplex`
: Disable pairing of duplex instructions.

`--mno-pairing-branch`
: Disable pairing of branch instructions.

`--mpairing-info`
: Report instruction pairing statistics.

NOTE The `--m` options are for the V4 processor version only. They are ignored with the other processor versions (Section 2.2.2.8). For more information see the *Hexagon V4 Programmer's Reference Manual*.

2.2.2.7 Symbol handling

`--defsym` *`name=value`*

Define a symbol that can be referenced in the assembly language. `value` must be an integer constant (Section 2.3.6).

Defined symbols can be used to control conditional directives (Section 2.6.35).

`--strip-local-absolute`

Remove from the generated symbol table all local symbols that are assigned absolute values.

`-L`
`--keep-locals`

Retain temporary symbols (Section 2.5.2) in the generated symbol table. By default the assembler removes temporary symbols.

If temporary symbols are retained in both the assembler and linker, they can be used in debugging.

`-f`

Speed up the assembler by stopping it from performing whitespace and comment preprocessing on the input files.

This option should be used only when assembling programs that were generated by the compiler. If it is used with programs that require normal preprocessing, the assembler will not work correctly.

`--reduce-memory-overheads`

Reduce the amount of memory used by the assembler for symbol lookup, at the expense of causing longer assembly times.

2.2.2.8 Processor version

`-march` *`archname`*
`-mcpu` *`archname`*

Specify the target Hexagon processor architecture (i.e., the instruction set) of the output file, which can be `hexagonv2`, `hexagonv3`, or `hexagonv4`. The default is `hexagonv2`.

`-mv2`

Equivalent to: `-march hexagonv2`

`-mv3`

Equivalent to: `-march hexagonv3`

`-mv4`

Equivalent to: `-march hexagonv4`

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

2.2.3 Screen messages

The assembler normally operates silently from the command line without generating any standard screen messages. However, if the listing options are used (Section 2.2.2.3) the assembler may generate an assembly listing to the terminal.

2.2.4 Warning and error messages

The assembler writes any warning and error messages to the standard error file (which is usually your terminal):

- Warnings report non-critical events and conditions which do not prevent the assembler from continuing to assemble the program.
- Errors report serious problems which terminate the assembler.

Warning and error messages have the following format:

```
filename:NNN: Message Text
```

`filename` indicates the source file that contains the problem. This is usually the current input file.

NNN indicates the line number in the source file where the problem occurred.

2.2.5 List files

The assembler generates an assembly list file if the list option is used (Section 2.2.2.3).

List files contain the assembly language source text along with information on how it was assembled into object code. They are useful for reading and debugging programs.

Example of assembly list file:

```
HEXAGON GAS  file.s                      page 1
Main heading
Sub heading
   1                    .text
   2                    .title "Main heading"
   3                    .sbttl "Sub heading"
   4 0000 01C00078      r1=#0
   5 0004 02C00078      r2=#0
   6
```

The page header appears at the top of every page in a list file. The header displays the tool name (`HEXAGON GAS`), current source file name (`file.s` in this case), page number, and, if defined with the directives `.title` (Section 2.6.74) and `.sbttl` (Section 2.6.57), the current page title and subtitle.

The remaining lines in a list file have the following format:

```
    line offset code source
```

The `line` field displays the line number in the source file of the associated source text.

`offset` displays the object-file-relative offset of the generated object code.

`code` displays the generated object code that corresponds to the assembly instruction.

`source` displays the original source text.

2.3 Basic syntax

This section describes the basic syntax of assembly language, including the set of valid characters and the rules for writing symbols, numbers, expressions, and comments.

2.3.1 Character set

The assembler recognizes the following characters:

- Alphabetic (`A`..`Z`, `a`..`z`)
- Numeric digits (`0`..`9`)
- Punctuation (including `+`, `[`, _, etc.)
- Non-printing (space, tab, newline, etc.)

2.3.2 Symbols

A *symbol* is defined as one or more of the following characters:

- All letters (both upper and lower case)
- digits
- the characters `‘_.$’`

No symbol can begin with a digit. Case is significant. There is no limit on symbol length: all characters are significant. Symbols are delimited by characters not in that set, or by the beginning of a file (since the source program must end with a newline, the end of a file is not a possible symbol delimiter).

2.3.3 Whitespace

Whitespace is one or more blanks or tabs, in any order. Whitespace is used to separate symbols, and to make programs neater for people to read. Except within character constants (Section 2.3.7), any whitespace means the same as exactly one space.

2.3.4 Keywords

Keywords are symbols reserved by the assembler. They cannot be redefined in the assembly language code. Keywords include the names of registers, assembly instructions, and assembly directive parameters.

For more information on keywords see the *Hexagon Programmer's Reference Manual*.

2.3.5 Directives

Assembler directives are items in a source file that the assembler recognizes as commands. Directives are used to declare sections, constants, and external symbols. They also control assembler features such as macros and include files.

Directives begin with a period character (.) and may be followed by one or more parameters. A space cannot appear after the period. Directive names are case-sensitive.

For example:

```
.set debug, 1
.if debug
```

For more information on directives see Section 2.6.

2.3.6 Numbers

Numbers can appear in hexadecimal, binary, octal, decimal, or floating-point format:

- Hexadecimal numbers begin with `'0x'` or `'0X'` and are followed by one or more hexadecimal digits (`'0123456789abcdefABCDEF'`)
- Binary numbers begin with `'0b'` or `'0B'` and are followed by one or more binary digits (`'01'`)
- Octal numbers begin with `'0'` and are followed by one or more octal digits (`'01234567'`)
- Decimal numbers begin with a non-zero digit and are followed by one or more decimal digits (`'0123456789'`)
- Floating point numbers begin with `'0e'` or `'0E'`and are followed by the following characters:
- An optional sign (`'+'` or `'-'`)
- An optional *integer part*: zero or more decimal digits
- An optional *fractional part*: `'.'` followed by zero or more decimal digits
- An optional exponent, consisting of:
    - An `'E'` or `'e'`
    - Optional sign: either `'+'` or `'-'`
    - One or more decimal digits

NOTE At least one of the integer part or fractional part must be present in a floating-point number.

To denote a negative number use the prefix operator `‘-’` (Section 2.3.9).

The assembler distinguishes between three types of numbers according to how they are stored in memory:

- *Integers* are stored as 32-bit integers
- *Bignums* are stored as 64-bit integers
- *Flonums* are stored as 32- or 64-bit floating point numbers

A bignum has the same syntax and semantics as an integer except that the number (or its negative) takes more than 32 bits to represent in binary. The distinction is made because in some places integers are permitted while bignums are not.

A flonum represents a floating point number. The translation is indirect: a decimal floating point number from the text is converted by the assembler to a generic binary floating point number of more than sufficient precision. This generic floating point number is converted to the target floating point format by a Hexagon processor-specific portion of the assembler.

NOTE The assembler performs all processing using integers. Flonums are computed independently of any floating point hardware in the host computer.

2.3.7 Characters

The assembler supports two types of character constants:

- A *character* represents a single character stored in one byte. Its value can be used in numeric expressions.
- A *string* (properly called a string *literal*) contain zero or more characters stored in multiple bytes. String values cannot be used in numeric expressions.

Characters consist of a single quote (‘) immediately followed by that character. For example:

```
‘e
```

The quote character must be an acute (left) quote, not a grave (right) one.

Strings are delimited by double-quote characters (“). They can contain double-quote or null characters. For example:

```
"string"
```

Special characters are included in characters and strings by preceding the special character with a backslash character (`'\'`). For example:

```
'\\
```

This example represents a single backslash character, with the first backslash serving as the escape character for the second.

The assembler supports the following escape characters for use in character and string constants:

`\b`	Mnemonic for backspace; for ASCII this is octal code 010.
`\f`	Mnemonic for form-feed; for ASCII this is octal code 014.
`\n`	Mnemonic for newline; for ASCII this is octal code 012.
`\r`	Mnemonic for carriage return; for ASCII this is octal code 015.
`\t`	Mnemonic for horizontal tab; for ASCII this is octal code 011.
`\`*`NNN`*	Octal character code. *NNN* denotes 3 octal digits. For compatibility with other UNIX systems, 8 and 9 are accepted as digits: for example, `\008` has the value 010, and `\009` the value 011.
`\x`*`hex-digits...`*	Hexadecimal character code. All trailing hex digits are combined. Either upper or lower case `x` works.
`\\`	Represents a single backslash character.
`\"`	Represents a single double-quote character. Needed in strings to represent this character, as an unescaped `` `"' `` would terminate the string.
`\`*`anything-else`*	Any other character when escaped by \ generates a warning message and translates the constant as if the escape character was not present. The assembler assumes that you used an escape sequence because you did not want the literal value of the following character. However, because the specified character is undefined, the assembler knows it is assembling the wrong code and so warns you.

Which characters are escapable, and what those escapes represent, is machine-dependent. If you are in doubt, do not use an escape sequence.

A newline (or at-sign `'@'`) immediately following an acute accent is taken as a literal character and does not count as the end of a statement (Section 2.3.10).

NOTE The value of a character constant in a numeric expression is the 8-bit ASCII code for that character.

2.3.8 Expressions

Expressions specify numeric or address values, and can be used wherever such values are allowed, including instructions and variable declarations. Whitespace can precede and/or follow an expression.

An *integer expression* is defined as one or more *arguments* delimited by *operators*.

Arguments are symbols, numbers or subexpressions.

NOTE In other contexts arguments are sometimes called “arithmetic operands”. To avoid confusing them with instruction operands, the term “argument” as used here refers to parts of expressions only, reserving “operand” for processor instruction operands.

Symbols are evaluated to yield {*section NNN*} where *section* is a section name (Section 2.4). *NNN* is a signed, 2’s complement 32-bit integer.

Numbers are usually integers.

A number can be a flonum or bignum (Section 2.3.6). In this case, the assembler uses only the low-order 32 bits of the bignum, and generates a warning message to that effect.

Subexpressions consist of a left parenthesis `‘(’` followed by an integer expression, followed by a right parenthesis `‘)’`; or a prefix operator followed by an argument.

NOTE The result of an expression must evaluate to either an absolute number or an offset into a particular section (Section 2.4). If an expression result is not absolute, and the assembler does not have enough information to determine its section, the assembler flags the expression with an error message.

An empty expression has no value: it is just whitespace or null. Wherever an absolute expression is required, you can omit the expression, and the assembler assumes a value of (absolute) 0.

2.3.9 Operators

Operators are arithmetic functions such as `+` or `%`. Prefix operators are followed by an argument. Infix operators appear between their arguments. Operators can be preceded and/or followed by whitespace.

Prefix operators

The assembler supports the following *prefix operators*. They each take one argument, which must be absolute.

`-`	*Negation*. Two’s complement negation.
`~`	*Complementation*. Bitwise not.

Infix operators

Infix operators take two arguments, one on either side. Operators have precedence, but operations with equal precedence are performed left to right. Apart from `+` or `-`, both arguments must be absolute, and the result is absolute.

Highest Precedence

`*`	*Multiplication.*
`/`	*Division*. Truncation is the same as the C operator `'/'`
`%`	*Remainder.*
`<` `<<`	*Shift Left.* Same as the C operator `'<<'`.
`>` `>>`	*Shift Right.* Same as the C operator `'>>'`.

Intermediate precedence

`|`	*Bitwise Inclusive Or.*
`&`	*Bitwise And.*
`^`	*Bitwise Exclusive Or.*
`!`	*Bitwise Or Not.*

Lowest Precedence

+ *Addition*. If either argument is absolute, the result has the section of the other argument. You cannot add together arguments from different sections.

- *Subtraction*. If the right argument is absolute, the result has the section of the left argument. If both arguments are in the same section, the result is absolute. You cannot subtract arguments from different sections.

NOTE It's meaningful only to add or subtract the *offsets* in an address; you can only have a defined section in one of the two arguments.

2.3.10 Statements

A *statement* ends at a newline character (`'\n'`). The newline character is considered part of the preceding statement. Newlines within character constants are an exception: they do not end statements.

NOTE It is an error to end any statement with the end-of-file character: the last character of any input file should be a newline.

You can write a statement on more than one line if you put a backslash (\) immediately in front of any newlines within the statement. When the assembler reads a backslashed newline both characters are ignored. You can even put backslashed newlines in the middle of symbol names without changing the meaning of your source program.

An empty statement is allowed, and can include whitespace. It is ignored.

A statement begins with zero or more labels (Section 2.5.4), and is optionally followed by a key symbol which determines what kind of statement it is. The key symbol determines the syntax of the rest of the statement. If the symbol begins with a dot `'.'` then the statement is an assembler directive. If the symbol begins with a letter the statement is an assembly language *instruction*: it assembles into a machine language instruction.

2.3.11 Comments

The assembler supports two forms of comments:

- C-style delimited comments
- C++-style line comments

Delimited comments begin with `/*` and end with `*/`. They can span a number of lines, and cannot be nested. For example:

```
/*
  The only way to include a newline ('\n') in a comment
  is to use this sort of comment.
*/

/* This sort of comment does not nest. */
```

Line comments begin with // and go to the end of the source line:

```
.set debug, 1  // Include debugging code
```

In both cases the comment is equivalent to a single space character.

Logical file names

To be compatible with past assemblers, lines that begin with `'#'` have a special interpretation. Following the `'#'` should be an absolute expression (Section 2.3.8) which specifies the logical line number of the *next* line. Following the expression is an optional string (Section 2.3.7); if present, it specifies the new logical file name. The rest of the line, if any, should be whitespace.

If the first non-whitespace characters on the line are not numeric, the line is ignored. (Just like a comment.)

```
                           // This is an ordinary comment.
# 42-6 "new_file_name"     // New logical file name
                           // This is logical line # 36.
```

NOTE This feature is deprecated, and may disappear from future versions of the assembler.

2.4 Sections and relocation

Assembly language programs are structured in units called *sections*. Sections are assembled and linked together to form executable programs. The sections that make up a program can be stored in a single source file, or they can be maintained as a collection of separately assembled files.

Roughly, a section is a range of addresses, with no gaps; all data stored within the address range is treated the same for some particular purpose. For example, the following source code defines a "read-only" data section:

```
.Ltext0:
          .section     .data
          .p2align     3
.LC0:
          .string      "hello, world\n"
          .string      "stddef.h"
          ...
```

The linker reads many object files (partial programs) and combines their contents to form a runnable program. When the assembler generates an object file, the partial program is assumed to start at address 0. The linker assigns the final addresses for the partial program, so that different partial programs do not overlap. This is actually an oversimplification, but it suffices to explain how the assembler uses sections.

The linker moves blocks of bytes of your program to their run-time addresses. These blocks are moved as rigid units: their length does not change, nor does the order of bytes within them. Such a rigid unit is called a *section*. Assigning run-time addresses to sections is called *relocation*. It includes the task of adjusting references to object-file addresses so they refer to the proper run-time addresses.

An object file written by the assembler has at least three sections, any of which can be empty. These are named *text*, *data* and *bss* sections.

Within the object file, the text section starts at address `0`, the data section follows the text section, and the bss section follows the data section.

To inform the linker about which data changes when the sections are relocated, and how to change that data, the assembler also writes to the object file details of the relocation needed. To perform relocation the linker requires the following information for each occurrence of an address reference in the object file:

- The location in the object file of the address reference
- The length (in bytes) of the reference
- The section referred to by the address
- The numeric value of (*address*) - (*start-address of section*)
- Whether or not the reference is Program Counter (PC) relative

In fact, every address the assembler ever uses is expressed as:

```
(section) + (offset into section)
```

Further, most expressions the assembler computes have this section-relative nature.

NOTE The notation {*secname N*} is used to specify "offset *N* into section *secname*."

The concept of a section is extended to an *undefined* section. Any address whose section is unknown at assembly time is by definition rendered {`undefined U`}, where *U* is filled in later with a run-time address. Since numbers are always defined, the only way to generate an undefined address is to reference an undefined symbol. A reference to a named common block would be such a symbol: its value is unknown at assembly time so it has section *undefined*.

NOTE The term *section* is commonly used to describe groups of sections in a linked program. The linker assigns all partial programs' text sections to contiguous addresses in the linked program. It is customary to refer to the *text section* of a program, meaning all the addresses of all partial programs' text sections. The same terminological convention is used for the data and bss sections.

Some sections are manipulated by the linker; others are defined solely for use by the assembler, and have no meaning except during assembly.

2.4.1 Code and data sections

The assembler defines standard sections for containing program code and data.

Table 2-1 lists the standard code and data sections.

Table 2-1 Code and data sections

Section Name	Description

`.text`
`.data`
`.rodata`
	These sections contain the program code and data. The assembler and linker treat the sections as functionally equivalent. However, when a program is running it is customary for the text section to be unalterable.
	The `.text` section contains program code.
	The `.data` section of a running program is usually alterable: for example, C variables are typically stored in the data section. `.rodata` specifies a read-only data section (which is used to store constant data).
	Additional text and data sections are predefined to directly reference Hexagon processor-specific memories (Section 2.4.4), or are user-defined (Section 2.4.5).

`.bss`	This section contains zeroed bytes when your program begins running. It is used to store uninitialized variables or common data. The length of each partial program's bss section is important, but because the section starts out containing zeroed bytes, there is no need to store explicit zero bytes in the object file. The `bss` section is used to eliminate explicit zeros from object files.

Figure 2-2 shows a high-level example of how the sections of two partial programs are relocated in memory. Memory addresses are defined to lie on the horizontal axis.

```
                      +-----+----+--+
partial program #1:   |ttttt|dddd|00|
                      +-----+----+--+
                      text   data bss

                      +---+---+---+
partial program #2:   |TTT|DDD|000|
                      +---+---+---+
                      text data bss

                      +--+---+-----+--+----+---+-----+~~
linked program:       |  |TTT|ttttt|  |dddd|DDD|00000|
                      +--+---+-----+--+----+---+-----+~~
    addresses:        0 ...
```

Figure 2-2 Section type relocation

NOTE If sections are not specified in the source code, the assembler automatically assigns the program code and data to the sections listed in Table 2-1.

2.4.2 Block storage section

The block storage section (abbreviated as "bss") is used for local common variable storage. You can allocate address space in the bss section, but you can not dictate data to load into it before your program executes. When your program starts running, all the contents of the bss section are zeroed bytes.

Table 2-1 above defines the bss section.

The `.lcommon` pseudo-op defines a symbol in the bss section; see Section 2.6.41.

The `.common` pseudo-op can be used to declare a common symbol, which is another form of uninitialized symbol; see Section 2.6.12.

Alternatively, you can switch to the bss section with the `.section` directive (Section 2.6.58) and define symbols within the section in the usual manner (though bss symbols can only be assigned zero values). Typically a bss section contains only symbol definitions and `.skip` directives (Section 2.6.64).

2.4.3 Global sections

The assembler includes predefined sections which are used to assign program data to the global data area.

Table 2-2 lists the predefined section names for the global data sections.

Table 2-2 Processor global data sections

Section Name	Description
`.sdata`	global `.data` section (Section 2.4.1)
`.sbss`	global `.bss` section (Section 2.4.2)

2.4.4 Hexagon processor memory sections

The assembler includes predefined text and data sections which are used to assign program code or data to the following Hexagon processor-specific (or MSM-specific) memories:

- External bus interface (EBI)
- Tightly-coupled memory (TCM)
- Stacked memory interface (SMI)

In addition to specifying the memory, the predefined sections also specify the cache properties of the stored code/data.

Table 2-3 lists the predefined section names for the Hexagon processor-specific memories.

Table 2-3 Hexagon processor-specific memory sections

Section Name	Description
`.ebi_code_cached`	Code
`.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`	
`.ebi_data_uncached`	Data (uncached)
`.tcm_data_uncached`	
`.smi_data_uncached`	

For example:

```
.section .tcm_code_cached
.section .ebi_data_cached_wt
```

For more information on the Hexagon processor memories and memory cache see the *Hexagon Programmer's Reference Manual*.

NOTE When they are used the Hexagon processor memory sections must be assigned to specific memory areas – this is done using the linker. For more information see Section 2.4.6.

Program code and data cannot be stored in the same section.

2.4.5 User-defined sections

User-defined sections are used for storing code or data in non-standard memory configurations. For example, a program may define a dedicated data section for accessing memory-mapped hardware devices.

User-defined sections are created by specifying section names that are distinct from the predefined section names (Table 2-1 through Table 2-3). For example:

```
.section my_data
```

The type of a user-defined section (code, data, read-only data: see Section 2.4.1) is automatically determined by the objects stored in the section. Table 2-4 shows the mapping.

Table 2-4 User-defined sections

Section Contents	Section Type
Code	`.text`
Data	`.data`
Literals or strings	`.rodata`

The type of a user-defined section is conventionally included in the section name by prefixing the name with the corresponding predefined section name. For example:

```
.section .data.my_data
.section .text.my_code
.section .rodata.my_strings
```

NOTE By default the linker assigns all sections with similar type to contiguous addresses in the linked program. This behavior can be changed by creating a linker script (Section 3.3).

2.4.6 Section memory assignment

When they are used the Hexagon processor memory sections (Section 2.4.4) must be assigned to specific memory areas – this is done using the linker.

All other sections (including any user-defined sections) are automatically assigned to EBI memory during linking.

NOTE For more information on section memory assignment see Section 3.4.

2.4.7 Sub-sections

Assembled bytes are conventionally assigned to one of two sections: code or data. A program may contain separate groups of data in named sections that are not contiguous in the assembler source, but that you want to end up next to each other in the object file. The assembler allows you to use *subsections* for this purpose. Each section can contain one or more numbered subsections, with the section numbers ranging from 0 to 8192. An object assembled into a specific subsection is assigned to a location in the object file which is adjacent to all the other objects in the program that were assigned to the same subsection.

For example, a programmer may want to store all program constants in the text section of a program, but not have the constants intermixed with the program code being assembled. In this case, the subsection directive `'.text 0'` can appear before each section of code being output, and the subsection directive `'.text 1'` before each group of constants.

Subsections are optional. If you do not use subsections, all items are assigned by default to subsection number zero.

Each subsection is zero-padded up to a multiple of four bytes. (Subsections may be padded by different amounts on different flavors of the assembler.)

Subsections appear in an object file in numeric order, lowest numbered to highest. (All this to be compatible with other people's assemblers.) Note that an object file contains no representation of subsections; the linker and other programs that manipulate object files see no trace of them. They just see all text subsections as a text section, and all data subsections as a data section.

To specify which subsection you want subsequent statements assembled into, use a numeric argument to specify it, in a '`.text` *expression*' or a '`.data` *expression*' statement. The *expression* parameter specifies an absolute expression (Section 2.3.8). If you specify just `.text` then '`.text 0`' is assumed. Likewise `.data` means '`.data 0`'. Assembly begins at `text` address 0. For example:

```
.text 0     # The default subsection is text 0.
.ascii "This lives in the first text subsection. *"
.text 1
.ascii "But this lives in the second text subsection."
.data 0
.ascii "This lives in the data section,"
.ascii "in the first data subsection."
.text 0
.ascii "This lives in the first text section,"
.ascii "immediately following the asterisk (*)."
```

Each section has a *location counter* incremented by one for every byte assembled into that section. Because subsections are merely a convenience restricted to the assembler there is no concept of a subsection location counter. There is no way to directly manipulate a location counter; however, the `.align` directive changes it, and any label definition captures its current value. The location counter of the section where statements are being assembled is said to be the *active* location counter.

2.4.8 Assembler internal sections

These sections are meant only for the internal use of the assembler. They have no meaning at run-time. You do not really need to know about these sections for most purposes; but they can be mentioned in the assembler warning messages, so it might be helpful to have an idea of their meanings to the assembler. These sections are used to permit the value of every expression in your assembly language program to be a section-relative address.

`ASSEMBLER-INTERNAL-LOGIC-ERROR!`
An internal assembler logic error has been found. This means there is a bug in the assembler.

`expr section`
The assembler stores complex expression internally as combinations of symbols. When it needs to represent an expression as a symbol, it puts it in the `expr` section.

2.5 Symbols

Symbols are a central concept in assembly language:

- Programmers use symbols to name items in a program
- The assembler uses symbols to reference items in a program
- The linker uses symbols to perform linking
- The debugger uses symbols to present program information
- The profiler uses symbols to present profile data

2.5.1 Symbol names

Each symbol has exactly one name (as defined in Section 2.3.2). For example:

```
test_array
```

Each name in an assembly language program refers to exactly one symbol. A symbol can be referenced by its name any number of times in a program.

2.5.2 Temporary symbols

Temporary symbols are symbols whose names begin with `'.L'`. For example:

```
.L_text
```

The assembler treats temporary symbols like regular symbols, but normally does not save their definitions in the object file. Thus, temporary symbols are not visible when linking, debugging, or profiling.

However, if the `-L` option (Section 2.2.2.7) is used the assembler saves temporary symbols in the object file as local symbols. If the linker is similarly instructed to save local symbols, the temporary symbols can then be used in debugging and profiling.

NOTE Temporary symbols are distinct from the local and global symbol types that are supported in the ELF format.

2.5.3 Symbol assignment

Symbols can be assigned arbitrary values in a program by using the `.set` directive (Section 2.6.59). For example:

```
.set x, 3
```

NOTE The assembler does *not* support symbol assignment of the form "*`symbol = value`*".

2.5.4 Labels

Labels consist of a symbol immediately followed by a colon `':'`. For example:

```
foo:
```

A label symbol is automatically assigned the current value of the active location counter, and can be used to specify that value. For example:

```
jump foo
```

The assembler generates a warning if you try to assign the same label symbol to two different locations: the first label definition overrides any other definitions.

NOTE Whitespace is permitted before or after a label, but cannot appear between a label's symbol and the following colon.

Labels should *not* be used in embedded assembly functions, as they generate profile data that can cause problems in the profiler utilities. To avoid this problem, use local labels (Section 2.5.5) instead.

2.5.5 Local labels

Local labels are labels that use temporary symbols. Using local labels in a program reduces the number of label symbols, and – because temporary symbols are not saved in the program object file – can also reduce the number of symbols visible during debugging and profiling.

Local labels can be specified in two forms:

- Symbolic local labels
- Numeric local labels

2.5.5.1 Symbolic local labels

Symbolic local labels are declared and used like regular labels (Section 2.5.4), but specify a temporary symbol (Section 2.5.2) as the label symbol. For example:

```
.L_foo:
```

2.5.5.2 Numeric local labels

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

```
2:
```

Numeric local labels enable programmers to re-use the same set of numeric label names (`0:`, `1:`, `2:`, ...) throughout a program.

Numeric local labels can be used for both backward and forward references. To refer to the preceding definition of a numeric label, write ***N*b** (where *N* specifies the same non-negative integer that appears in the label). To refer to the following definition of a numeric label, write ***N*f**. For example:

```
33: nop;
    jump 33b;      // jump to preceding local label 33
33: nop;
```

NOTE The **b** stands for "backwards" and the **f** for "forwards".

Numeric local labels are not generated by the C compiler (except when declared in inline assembly language).

No restrictions exist on using numeric local labels, except that at any point in an assembly program you can reference only the immediately preceding and following instances of a given numeric label symbol (e.g., `33:` as shown in the example above).

The assembler transforms each numeric local label instance into a symbolic local label (Section 2.5.5.1) with a unique name that encodes the numeric local label integer, an ASCII control character (ctrl-B), and a second integer indicating the specific instance of the numeric local label that uses that particular label integer. For example:

```
.L33<ctrl-B>2
```

This name encoding ensures that the generated symbolic local will not conflict with any other program symbols, whether programmer- or assembler-defined.

NOTE The assembler-generated symbolic local label names can appear in error messages, and can be optionally saved in the object file (Section 2.5.2).

2.5.6 Dot symbol

The special symbol `'.'` can be used to specify the current address that the assembler is assembling into. For example:

```
melvin: .long .
```

In this example the period immediately following the `.long` directive (Section 2.6.44) specifies the initial value of the 32-bit number generated by `.long`. The value is the address of the generated number (which is identical to the value of the label `melvin`).

NOTE Assigning a value *to* the dot symbol is equivalent to using the `.org` directive (Section 2.6.49). For example, `'.=.+4'` is equivalent to `'.space 4'`.

2.5.7 Symbol attributes

Along with the symbol name, every symbol has the following attributes:

- `value`
- `type`
- `size`
- `scope`
- `visibility`
- `descriptor`

If you use a symbol without defining it, the assembler assumes the value zero for each of these attributes, and probably will not warn you. This makes the symbol an externally defined symbol, which is generally what you would want in this situation.

2.5.7.1 Symbol value

Symbol values are (usually) 32-bit numeric values. They are set explicitly by the programmer or automatically by the assembler.

For a symbol that labels a location in a text, data, or bss section, the value is the number of addresses from the start of that section to the label. The value of the symbol changes as the linker changes the symbol's section base address during linking.

The value of an undefined symbol is defined as follows:

- A zero value indicates that the symbol is not defined in the assembler source file; the linker will attempt to determine the symbol value from symbols defined in the other files linked to the program. This kind of symbol is created by referencing a symbol name without defining it.
- A non-zero value indicates the symbol was declared in a `.common` directive (Section 2.6.12). The value specifies in bytes (addresses) how much common storage to reserve. The linker changes this symbol value to be the initial address of the allocated common data storage.

2.5.7.2 Symbol type

The `type` attribute of a symbol indicates whether the symbol is a function symbol or object symbol.

It can be set with the directives `.type` (Section 2.6.75) or `.stab` (Section 2.6.66).

2.5.7.3 Symbol size

The `size` attribute of a symbol indicates the size (in bytes) of the object associated with the symbol. It is typically used to store the size of function symbols.

It can be set with the directives `.size` (Section 2.6.62) or `.stab` (Section 2.6.66).

2.5.7.4 Symbol scope

The `scope` attribute of a symbol indicates the scope of the symbol:

- Local – Not visible in other components of a program (default)
- Global – Visible in all components of a program

Symbols can be made global with the directive `.global` (Section 2.6.31).

NOTE The assembler treats undefined symbols as references to global symbols defined in other components.

Local symbols can be removed from the assembler output by using the object file stripper (Section 4.10).

2.5.7.5 Symbol visibility

The `visibility` attribute of a symbol indicates the visibility of the symbol:

- Hidden – Not visible in other components (and implicitly protected)
- Internal – Hidden, and also requires additional processing
- Protected – Always resolved to symbol definition in the current component, even if a definition in another component would normally preempt such a resolution.

It can be set with the directives `.hidden` (Section 2.6.33), `.internal` (Section 2.6.38), or `.protected` (Section 2.6.51).

2.5.7.6 Symbol descriptor

The `descriptor` attribute of a symbol contains an arbitrary 16-bit value.

You can set a symbol's descriptor value by using the `.desc` directive (Section 2.6.14) or `.stab` directive (Section 2.6.66).

NOTE This attribute is not used by the assembler.

2.6 Directives

Assembler directives are items in a source file that the assembler recognizes as commands. Directives are used to declare sections, constants, data, and symbols. They also control assembler features such as macros and include files.

Directives begin with a period character (.) and may be followed by one or more parameters. A space cannot appear after the period.

For example:

```
.set debug, 1
.if debug
```

The assembler recognizes the following directives:

Sections

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

Symbols

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

Symbol attributes

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

Alignment

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

Data allocation

```
.byte [expr [,expr]...]
.2byte [expr [,expr]...]
.3byte [expr [,expr]...]
.4byte [expr [,expr]...]
.word [expr [,expr]...]
.hword [expr [,expr]...]
.half [expr [,expr]...]
.short [expr [,expr]...]
.int [expr [,expr]...]
.long [expr [,expr]...]
.octa [bignum [,bignum]...]
.quad [bignum [,bignum]...]
.single [flonum [,flonum]...]
.double [flonum [,flonum]...]
.float [flonum [,flonum]...]
.ascii ["string" [,"string"]...]
.asciz ["string" [,"string"]...]
.fill repeat, size, value
.space size, fill
.skip size, fill
.block size, fill
.string "string" [,"string"]...
.string8 "string" [,"string"]...
.string16 "string" [,"string"]...
.string32 "string" [,"string"]...
.string64 "string" [,"string"]...
```

Conditionals

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

Macros

```
.macro name
.macro name argument...
.endm
.exitm
.purgem
.altmacro
.noaltmacro
```

Include files

```
.include "filename"
```

Debug

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

Assembler control

```
.end
.abort
.err
.eject
.rept [count]
.endr
```

Listings

```
.list
.nolist
.title "heading"
.sbttl "subheading"
.psize lines, columns
```

2.6.1 .2byte

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

Generate the specified expressions as 2-byte data items stored in consecutive addresses.

Accepts zero or more expressions separated by commas.

NOTE `.2byte` is equivalent to `.hword` (Section 2.6.34).

2.6.2 .3byte

```
.3byte [expr [,expr]...]
```

Generate the specified expressions as 3-byte data items stored in consecutive addresses.

Accepts zero or more expressions separated by commas.

2.6.3 .4byte

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

Generate the specified expressions as 4-byte data items stored in consecutive addresses.

Accepts zero or more expressions separated by commas.

NOTE `.4byte` is equivalent to `.word` (Section 2.6.77).

2.6.4 .abort

```
.abort
```

Stop assembly and exit the assembler.

2.6.5 .align

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

Pad the location counter (in the current subsection) until it is an integral multiple of the value of the specified expression. For example, `‘.align 8’` advances the location counter until it is an integral multiple of 8.

NOTE If the current location counter is already an integral multiple of the target value, then this directive does nothing.

The first argument (which must be an absolute expression) is the alignment request value expressed in bytes. The value must be an integral multiple of 2, and should not exceed 0x3FFFF.

The second argument (also an absolute expression) specifies the fill value to be stored in the padding bytes. If this argument is omitted the padding bytes are normally zero, but in a text section the space is filled with NOP instructions.

The third argument (also an absolute expression) specifies the maximum number of bytes to be skipped by this alignment directive. If performing the alignment would require skipping more bytes than the specified value, then the alignment is not performed.

NOTE The second argument can be omitted by putting two commas between the first and third arguments. This can be useful if you want the alignment to be filled with NOP instructions when appropriate.

`.align` is equivalent to `.balign` (Section 2.6.9).

2.6.6 .altmacro

```
.altmacro
```

Enable the following additional features in the macro processor (Section 2.6.45):

- Local names
- String delimiters
- Single-character string escape
- Expression results as strings
- Character delimiters

These features are described in detail below.

NOTE The additional features can be disabled with the `.noaltmacro` directive (Section 2.6.46).

This directive is equivalent to using the `--alternate` option (Section 2.2.2).

Local names

```
LOCAL name [,name]
```

The LOCAL directive causes the assembler to replace (in macro expansions) each instance of the specified names with a unique assembler-generated name. In particular, the generated names are different in each macro expansion. This makes it possible to define symbols in macros without concern for creating duplicate symbol definitions. The directive accepts one or more names separated by commas.

String delimiters

Character strings can be delimited in three ways (instead of the usual one):

- "string" (double quote delimiters – default)
- 'string' (single quote delimiters)
- <string> (angle bracket delimiters)

Single-character string escape

Any character (regardless of its normal meaning) can be inserted in a character string by prefixing it with '!' (exclamation mark).

For example, the character string

```
<4.2 !> 3.4!!>
```

yields the following literal text

```
<4.2 > 3.4!!>
```

Expression results as strings

`'%expr'` evaluates the expression *`expr`* and encodes the expression result as a character string.

Character delimiters

The ampersand character (‘`&`’) can be used as a separator character which is removed in the macro expansions. This is useful in cases where a macro definition uses symbols with special meanings in the assembly language that would otherwise prevent the macro from expanding properly.

For example, given the macro

```
.altmacro
.macro label l
l&:
.endm
```

the macro call

```
label my_label
```

generates the following text

```
my_label:
```

without having the assembler misinterpret the string “`l:`” as a label itself rather than as a macro argument followed by ‘`:`’.

2.6.7 .ascii

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

Generate the specified string literals (with no automatic trailing zero byte) as character data items stored in consecutive addresses.

Accepts zero or more string literals (Section 2.3.7) separated by commas.

2.6.8 .asciz

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

Identical to `.ascii`, except that each string is followed by a zero byte.

NOTE The “z” in `.asciz` stands for “zero”.

2.6.9 .balign

```
.balign[wl] abs-expr [,abs-expr] [,abs-expr]
```

Pad the location counter (in the current subsection) until it is an integral multiple of the value of the specified expression. For example, '`.balign 8`' advances the location counter until it is an integral multiple of 8.

NOTE If the current location counter is already an integral multiple of the target value, then this directive does nothing.

The first argument (which must be an absolute expression) is the alignment request value expressed in bytes. The value must be an integral multiple of 2, and should not exceed 0x3FFFF.

The second argument (also an absolute expression) specifies the fill value to be stored in the padding bytes. If this argument is omitted the padding bytes are normally zero, but in a text section the space is filled with NOP instructions.

The third argument (also an absolute expression) specifies the maximum number of bytes to be skipped by this alignment directive. If performing the alignment would require skipping more bytes than the specified value, then the alignment is not performed.

`.balignw` and `.balignl` are variants of `.balign`:

- `.balignw` treats the fill pattern as a two-byte word value
- `.balignl` treats the fill pattern as a four-byte longword value

For example, `.balignw 4,0x368d` aligns to a multiple of 4. If it skips 2 bytes, they are filled with the single value 0x368d. If it skips 1 or 3 bytes, the fill value is undefined.

NOTE The second argument can be omitted by putting two commas between the first and third arguments. This can be useful if you want the alignment to be filled with NOP instructions when appropriate.

2.6.10 .block

```
.block size, fill
```

Generate *size* bytes in consecutive addresses, each of value *fill*.

Both *size* and *fill* are absolute expressions. If the comma and *fill* are omitted, *fill* is assumed to be zero.

`.block` is equivalent to `.space` (Section 2.6.65).

2.6.11 .byte

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

Generate the specified expressions as 1-byte data items stored in consecutive addresses.

Accepts zero or more expressions separated by commas.

2.6.12 .common

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

Declare a common symbol named *symbol*. When linking, a common symbol in one object file can be merged with a defined or common symbol of the same name in another object file. If the linker does not see a definition for the symbol, just one or more common symbols, then it will allocate *length* bytes of uninitialized memory. *length* must be an absolute expression. If the linker sees multiple common symbols with the same name, and they do not all have the same size, it will allocate space using the largest size.

The optional *align* argument specifies the desired alignment of the symbol, expressed as a byte boundary (for example, an alignment of 16 means that the least significant 4 bits of the address should be zero). The alignment must be an absolute expression, and it must be an integral power of two. If the linker allocates uninitialized memory for the common symbol, it will use the alignment when placing the symbol. The default alignment is the largest power of two less than or equal to the size of the symbol, up to a maximum of 16.

The optional *access* argument specifies the size of the smallest memory access to be made to the symbol, expressed in bytes (for example, a value of 2 means that the smallest memory access will be 16-bit). The default is the value specified by the *align* argument.

NOTE The `-G` option (Section 2.2.2.6) determines whether `.common` assigns symbols to the processor global bss section (Section 2.4.3) or the standard bss section.

Both spellings (`.comm` and `.common`) are accepted as the directive name.

2.6.13 .data

```
.data subsection
```

Assemble the following statements onto the end of the data subsection numbered *subsection* (which is an absolute expression). If *subsection* is omitted, it defaults to zero.

2.6.14 .desc

```
.desc symbol, abs-expr
```

Set the `descriptor` attribute (Section 2.5.7.6) of the symbol denoted by *symbol* to the low 16 bits of the specified absolute expression.

NOTE This attribute is not used by the assembler.

2.6.15 .double

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

Generate the specified *flonums* (Section 2.3.6) as 64-bit floating point numbers stored in consecutive addresses.

Accepts zero or more *flonums* separated by commas.

2.6.16 .eject

```
.eject
```

Force a page break at this point, when generating assembly listings.

2.6.17 .else

```
.else
```

Mark the beginning of a section of code, which is assembled only if the condition for the preceding `.if` evaluated to false.

NOTE This directive supports conditional assembly (Section 2.6.35).

2.6.18 .elseif

```
.elseif abs-expr
```

Mark the beginning of a new conditional section of code, which is assembled only if the condition for the preceding `.if` evaluated to false. The conditional argument must be an absolute expression.

NOTE This directive supports conditional assembly (Section 2.6.35).

2.6.19 .end

```
.end
```

Mark the end of a program source file. The assembler does not process anything in the file past the `.end` directive.

2.6.20 .endif

```
.endif
```

Mark the end of a conditional section of code.

NOTE This directive supports conditional assembly (Section 2.6.35).

2.6.21 .endm

```
.endm
```

Mark the end of a macro definition declared with the `.macro` directive. See Section 2.6.45.

2.6.22 .endr

```
.endr
```

Mark the end of a sequence of source lines processed by the `.rept` directive. See Section 2.6.56.

2.6.23 .equ

```
.equ symbol, expr
```

Set the value of *symbol* to *expression*. Equivalent to `.set` (Section 2.6.59).

2.6.24 .equiv

```
.equiv symbol, expr
```

Like `.equ` and `.set` (Section 2.6.59), except that the assembler will signal an error if *symbol* is already defined.

Except for the contents of the error message, this is roughly equivalent to the following sequence of directives:

```
.ifdef SYM
.err
.endif
.equ SYM,VAL
```

2.6.25 .err

```
.err
```

Print an error message and (unless the `-Z` option was used) do not generate an object file. This can be used to signal an error in conditionally compiled code.

2.6.26 .exitm

```
.exitm
```

Perform early exit from a `.macro` directive. See Section 2.6.45.

2.6.27 .falign

```
.falign
```

Ensure that the following instruction packet does not cross a 4-word (16-byte) boundary by padding any empty slots in the previous instruction packets with `NOP` instructions. If this is not possible, pad the current location (in the current subsection) with a new instruction packet comprised solely of `NOP` instructions.

This directive is typically used in jump targets or at the top of a loop body.

NOTE No label can be defined on a statement (Section 2.3.10) containing the `.falign` directive. If necessary a label can be defined after the directive.

If the processor branches to a target whose packet crosses a 128-bit boundary, the resulting instruction fetch will stall for one cycle (hence the "f" in `.falign`). For details see the *Hexagon Programmer's Reference Manual*.

2.6.28 .file

```
.file fileno filename
```

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

fileno is a positive integer which specifies the index of an entry in the file name table. The value must be unique with respect to those specified in other instances of this directive that appear in the program.

filename is a C string literal.

NOTE This directive is active only when generating DWARF 2 debug information (Section 2.2.2.6). For more information see www.dwarfstd.org.

Filename indices are exposed to the user because the filename table is shared with the `.debug_info` section of the DWARF 2 debug information. Thus the user must know the exact indices that table entries will have.

2.6.29 .fill

```
.fill repeat, size, value
```

Generate *repeat* copies of *size* bytes in consecutive addresses.

result, *size* and *value* are absolute expressions. *repeat* can be zero or more. *size* can be zero or more, but if it is more than 8, then it is deemed to have the value 8, compatible with other people's assemblers. The contents of each *repeat* bytes is taken from an 8-byte number. The highest-order 4 bytes are zero. The lowest-order 4 bytes are *value* rendered in the byte-order of an integer on the computer the assembler is assembling for. Each *size* bytes in a repetition is taken from the lowest order *size* bytes of this number. Again, this bizarre behavior is compatible with other people's assemblers.

size and *value* are optional. If the second comma and *value* are absent, *value* is assumed zero. If the first comma and following tokens are absent, *size* is assumed to be 1.

2.6.30 .float

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

Generate the specified *flonums* (Section 2.3.6) as 32-bit floating point numbers stored in consecutive addresses.

Accepts zero or more *flonums* separated by commas.

NOTE `.float` is equivalent to `.single` (Section 2.6.61).

2.6.31 .global

```
.global symbol
.globl symbol
```

Make the specified symbol visible to the linker. If you define *symbol* in your partial program, its value is made available to other partial programs that are linked with it. Otherwise, *symbol* takes its attributes from a symbol of the same name from another file linked into the same program.

NOTE Both spellings (`.globl` and `.global`) are accepted as the directive name.

2.6.32 .half

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

Generate the specified expressions as 16-bit data items stored in consecutive addresses.

Accepts zero or more expressions separated by commas.

NOTE `.half` is equivalent to `.hword` (Section 2.6.34).

2.6.33 .hidden

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

Set the `visibility` attribute of the specified symbols to the value `hidden`.

This overrides the default symbol visibility (which is set by the symbol binding).

Hidden symbols are not visible to other components. They are always considered to be protected as well.

NOTE See also `.internal` (Section 2.6.38) and `.protected` (Section 2.6.51).

2.6.34 .hword

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

Generate the specified expressions as 16-bit data items stored in consecutive addresses.

Accepts zero or more expressions separated by commas.

NOTE `.hword` is equivalent to `.short` (Section 2.6.60).

2.6.35 .if

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

Mark the beginning of a conditional section of code, which is assembled only if the specified absolute expression evaluates to a non-zero value. The end of the conditional section of code is marked by `.endif` (Section 2.6.20). Optionally you can include conditional code for the alternative condition, flagged by `.else` (Section 2.6.17). If several conditions must be checked, `.elseif` (Section 2.6.18) can be used to avoid nesting `.if`/`.else` blocks within each subsequent `.else` block.

The following variants of `.if` are also supported:

```
.ifdef symbol
```

Assembles the following section of code if the specified *symbol* has been defined.

```
.ifndef symbol
.ifnotdef symbol
```

Assembles the following section of code if the specified *symbol* has not been defined. Both spelling variants are equivalent.

2.6.36 .include

```
.include "filename"
```

Include the contents of the specified text file at the current location in the source program.

The code from file *filename* is assembled as if it followed the point of the `.include`; when the end of the included file is reached, assembly of the original file continues. You can control the search paths used with the `-I` command-line option (Section 2.2.2.2).

Quotation marks are required around *filename*.

2.6.37 .int

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

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

Accepts zero or more expressions separated by commas.

NOTE `.int` is equivalent to `.word` (Section 2.6.77).

2.6.38 .internal

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

Set the `visibility` attribute of the specified symbols to the value `internal`.

This overrides the default symbol visibility (which is set by the symbol binding).

Internal symbols are considered to be hidden, and require additional processing to be performed on them.

NOTE See also `.hidden` (Section 2.6.33) and `.protected` (Section 2.6.51).

2.6.39 .irp

```
.irp symbol, values...
```

Evaluate a sequence of statements assigning different values to *symbol*. The sequence of statements starts at the `.irp` directive, and is terminated by an `.endr` directive. For each *value*, *symbol* is set to *value*, and the sequence of statements is assembled. If no *value* is listed, the sequence of statements is assembled once, with *symbol* set to the null string. To refer to *symbol* within the sequence of statements, use \symbol.

For example, assembling

```
.irp    param,1,2,3
move    d\param,sp@-
.endr
```

is equivalent to assembling

```
move    d1,sp@-
move    d2,sp@-
move    d3,sp@-
```

NOTE *symbol* can be defined as any identifier, including symbols normally reserved for use in the instruction syntax. In certain cases this can cause the instruction expansions to not work as expected – for more information see Section 2.6.6.

2.6.40 .irpc

```
.irp symbol, values...
```

Evaluate a sequence of statements assigning different values to *symbol*. The sequence of statements starts at the `.irpc` directive, and is terminated by an `.endr` directive. For each character in *value*, *symbol* is set to the character, and the sequence of statements is assembled. If no *value* is listed, the sequence of statements is assembled once, with *symbol* set to the null string. To refer to *symbol* within the sequence of statements, use \symbol.

For example, assembling

```
.irpc    param,123
move    d\param,sp@-
.endr
```

is equivalent to assembling

```
move    d1,sp@-
move    d2,sp@-
move    d3,sp@-
```

NOTE *symbol* can be defined as any identifier, including symbols normally reserved for use in the instruction syntax. In certain cases this can cause the instruction expansions to not work as expected – for more information see Section 2.6.6.

2.6.41 .lcommon

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

Reserve *length* (an absolute expression) bytes for a local common denoted by *symbol*. The section and value of *symbol* are those of the new local common. The addresses are allocated in the bss section, so that at run-time the bytes start off zeroed. *Symbol* is not declared global (see Section 2.6.31), so is normally not visible to the linker.

The optional *align* argument specifies the desired alignment of the symbol in the bss section, expressed as a byte boundary (for example, an alignment of 16 means that the least significant 4 bits of the address should be zero). The alignment must be an absolute expression, and it must be an integral power of two. If the linker allocates uninitialized memory for the common symbol, it will use the alignment when placing the symbol. The default alignment is the largest power of two less than or equal to the size of the symbol, up to a maximum of 16.

The optional *access* argument specifies the size of the smallest memory access to be made to the symbol, expressed in bytes (for example, a value of 2 means that the smallest memory access will be 16-bit). The default is the value specified by the *align* argument.

NOTE The `-G` option (Section 2.2.2.6) determines whether `.lcommon` assigns symbols to the processor global bss section (Section 2.4.3) or to the standard bss section.

Both spellings (`.lcomm` and `.lcommon`) are accepted as the directive name.

2.6.42 .list

```
.list
```

Control (in conjunction with the `.nolist` directive) whether or not assembly listings are generated. These two directives maintain an internal counter (which is zero initially). `.list` increments the counter, and `.nolist` decrements it. Assembly listings are generated whenever the counter is greater than zero.

By default, listings are disabled. When you enable them (with the `-a` command line option; see Section 2.2.2.3), the initial value of the listing counter is one.

2.6.43 .loc

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

Add rows to the `.debug_line` line number matrix that corresponds to the immediately following assembly instruction.

fileno is a positive integer which specifies the index of an entry in the file name table.

lineno and *column* are positive integers which specify a position in the specified file.

NOTE The *fileno*, *lineno*, and optional *column* arguments are applied to the `.debug_line` state machine before the row is added.

options can specify one or more of the following tokens in any order:

`basic_block`	Set `basic_block` register in `.debug_line` state machine to true
`prologue_end`	Set `prologue_end` register in `.debug_line` state machine to true
`epilogue_end`	Set `epilogue_begin` register in `.debug_line` state machine to true
`is_stmt` *`value`*	Set `is_stmt` register in `.debug_line` state machine to specified value (which must be 0 or 1)
`isa` *`value`*	Set `isa register` in `.debug_line` state machine to specified value (which must be unsigned integer)

NOTE This directive is active only when generating DWARF 2 debug information (Section 2.2.2.6). For more information see www.dwarfstd.org.

Filename indices are exposed to the user because the filename table is shared with the `.debug_info` section of the DWARF 2 debug information. Thus the user must know the exact indices that table entries will have.

2.6.44 .long

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

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

Accepts zero or more expressions separated by commas.

NOTE `.long` is equivalent to `.int` (Section 2.6.37).

2.6.45 .macro

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

Define macro instructions that generate assembly output. For example, given the following macro definition:

```
.macro assign_r1 p
r1=\p
.endm
```

... the macro call:

```
assign_r1 r2
```

... generates the following assembly output:

```
r1=r2
```

The macro name specified in a macro definition is used to subsequently call the macro.

If a macro defines arguments, the argument names are specified in the macro definition following the macro name, and are separated by commas or spaces:

- A default value can be defined for any macro argument by following the argument name with `=`*`default_value`*. The default is used if the argument is passed a blank value.
- A macro argument declared with the suffix `:req` must always be passed a non-blank value.
- A macro argument declared with the suffix `:vararg` is assigned all the remaining arguments passed in a macro call.

The following examples demonstrate the use of macro arguments:

```
.macro comm
```

This directive begins the definition of a macro named `comm` which accepts no arguments.

```
.macro plus1 p, p1
.macro plus1 p p1
```

These directives show two different ways to begin the definition of a macro named `plus1` which accepts the two arguments `p` and `p1`. Within the macro definition the arguments are evaluated by preceding them with a backslash character (for example, `r1=\p+\p1`).

```
.macro reserve_str p1=0 p2
```

This directive begins the definition of a macro named `reserve_str` which accepts two arguments. `p1` is assigned a default value, while `p2` is not. This macro can be called either as `'reserve_str` *`a`*`,`*`b`*`'` (with `\p1` evaluating to *`a`* and `\p2` evaluating to *`b`*) or as `'reserve_str ,`*`b`*`'` (with `\p1` evaluating to its default value `0` and `\p2` evaluating to *`b`*).

NOTE Commas must be used as separators when specifying blank arguments.

In macro calls the argument values can be specified by position or by keyword. For example, the macro call `'plus1 9,17'` is equivalent to `'plus1 p1=17, p=9'`.

```
\@
```

The assembler maintains a counter of how many macros it has executed. The value of this counter is accessible within a macro definition via the pseudo-variable `'\@'`. Note that `'\@'` cannot be used outside a macro definition.

```
.exitm
```

This directive is used to exit early from the current macro definition (Section 2.6.26).

```
.endm
```

This directive marks the end of a macro definition (Section 2.6.21).

A macro can be defined recursively. For example, the following macro definition uses recursion to generate a sequence of data values in memory:

```
.macro  sum from=0, to=5
.long   \from
.if     \to-\from
sum     "(\from+1)",\to
.endif
.endm
```

Given this definition, the call `'sum 0,5'` generates the following assembly statements:

```
.long   0
.long   1
.long   2
.long   3
.long   4
.long   5
```

NOTE Macro names and argument names can be defined as any identifier, including symbols normally reserved for use in the instruction syntax. In certain cases this can cause macro arguments to not work as expected – for more information see the `.altmacro` directive (Section 2.6.6).

If a macro definition attempts to redefine an existing macro name, the assembler will generate an error message. Macro names can be redefined only if they are first recycled with the `.purgem` directive (Section 2.6.53).

If a macro call does not specify a value for a given argument, and the argument does not have a defined default value, then the argument will be evaluated in the macro definition as the null string.

2.6.46 .noaltmacro

```
.noaltmacro
```

Disable additional features in the macro processor (Section 2.6.6).

2.6.47 .nolist

```
.nolist
```

Control (in conjunction with the `.list` directive) whether or not assembly listings are generated. These two directives maintain an internal counter (which is zero initially). `.list` increments the counter while `.nolist` decrements it. Assembly listings are generated whenever the counter is greater than zero.

2.6.48 .octa

```
.octa [bignum [,bignum]...]
```

Generate the specified *bignums* (Section 2.3.6) as 16-byte integers stored in consecutive addresses.

Accepts zero or more *bignums* separated by commas.

NOTE The term "octa" comes from processors in which a "word" is two bytes; hence *octa*-word for 16 bytes.

2.6.49 .org

```
.org new-lc, fill
```

Advance the location counter of the current section to *new-lc*. *new-lc* is either an absolute expression or an expression with the same section as the current subsection. That is, you can't use `.org` to cross sections: if *new-lc* has the wrong section, the `.org` directive is ignored.

`.org` can only increase the location counter, or leave it unchanged; you cannot use `.org` to move the location counter backwards.

Because the assembler tries to assemble programs in one pass, *new-lc* cannot be undefined.

Be aware that the origin is relative to the start of the section, not to the start of the subsection. This is compatible with other people's assemblers.

When the location counter (of the current subsection) is advanced, the intervening bytes are filled with *fill* which should be an absolute expression. If the comma and *fill* are omitted, *fill* defaults to zero.

2.6.50 .p2align

```
.p2align[wl] abs-expr [,abs-expr] [,abs-expr]
```

Pad the location counter (in the current subsection) until it is an integral multiple of 2^x, where x is the value of the specified expression. For example, '`.p2align 3`' advances the location counter until it is an integral multiple of 8 (i.e., 2^3).

NOTE If the current location counter is already an integral multiple of the target value, then this directive does nothing.

The first argument (which must be an absolute expression) can be interpreted as specifying the number of low-order zero bits that the location counter must contain after being advanced. The argument value should not exceed 17.

The second argument (also an absolute expression) specifies the fill value to be stored in the padding bytes. If this argument is omitted the padding bytes are normally zero, but in a text section the space is filled with NOP instructions.

The third argument (also an absolute expression) specifies the maximum number of bytes to be skipped by this alignment directive. If performing the alignment would require skipping more bytes than the specified value, then the alignment is not performed.

`.p2alignw` and `.p2alignl` are variants of `.p2align`:

- `.p2alignw` treats the fill pattern as a two-byte word value
- `.p2alignl` treats the fill pattern as a four-byte longword value

For example, `.p2alignw 2,0x368D` aligns to a multiple of 4. If it skips 2 bytes, they will be filled in with the value 0x368D. If it skips 1 or 3 bytes, the fill value is undefined.

NOTE The second argument can be omitted by putting two commas between the first and third arguments. This can be useful if you want the alignment to be filled with NOP instructions when appropriate.

2.6.51 .protected

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

Set the `visibility` attribute of the specified symbols to the value protected.

This overrides the default symbol visibility (which is set by the symbol binding).

Any references to a protected symbol from within the component that defines it must be resolved to the symbol definition in that component, even if a definition in another component would normally preempt such a resolution.

NOTE See also `.hidden` (Section 2.6.33) and `.internal` (Section 2.6.38).

2.6.52 .psize

```
.psize lines [,width]
```

Specify the number of lines, and optionally the character width, to use for each page in an assembly list file (Section 2.2.5).

The default page line count is 60, and the default character width is 200.

The assembler generates a form-feed whenever the specified number of lines is exceeded, or whenever you explicitly request one using `.eject` (Section 2.6.16).

NOTE If you specify `lines` as 0, no form-feeds are generated except for those explicitly specified with `.eject`.

2.6.53 .purgem

```
.purgem
```

Mark the end of a macro definition declared with the `.macro` directive. See Section 2.6.45.

2.6.54 .quad

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

Generate the specified *bignums* (Section 2.3.6) as 8-byte integers stored in consecutive addresses.

Accepts zero or more *bignums* separated by commas.

If the bignum won't fit in 8 bytes, the assembler generates a warning message; and just takes the lowest order 8 bytes of the bignum.

NOTE The term "quad" comes from processors in which a "word" is two bytes; hence *quad*-word for 8 bytes.

2.6.55 .reloc

```
.reloc offset, reloc_name [, expr]
```

Generate a relocation at the specified offset with type *reloc_name* and optional value *expression*.

If *offset* is a number, the relocation is generated in the current section.

If *offset* is an expression that resolves to a symbol plus offset, the relocation is generated in the given symbol's section.

If specified, *expression* must resolve to an absolute value or to a symbol plus an addend.

2.6.56 .rept

```
.rept [count]
```

Repeat the sequence of lines between the `.rept` directive and the next `.endr` directive *count* times.

The default count value is 0, which specifies no generation of data.

For example, assembling

```
.rept   3
.long   0
.endr
```

is equivalent to assembling

```
.long   0
.long   0
.long   0
```

2.6.57 .sbttl

```
.sbttl ["string"]
```

Set the page header subtitle on an assembly list file (Section 2.2.5). If this directive is not specified (or if it is specified, but with the string argument omitted), the page header subtitle line is set to blank.

This directive normally sets the subtitle on the next page to be generated; however, if it appears within ten lines of the top of the page, it sets the subtitle on the current page.

NOTE If multiple `.sbttl` directives appear on a page, the first affects the current page and the last the next page. Any other instances in-between are ignored.

2.6.58 .section

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

Assemble the following code into a section (Section 2.4) named *name*.

The optional *flags* argument is a quoted string which can contain any combination of the following characters:

a	Section is allocatable
w	Section is writable
x	Section is executable

The optional *type* argument can contain one of the following constants:

`@progbits`	Section contains data
`@nobits`	Section does not contain data (i.e., section only occupies space)

If no flags are specified, the default flags depend upon the section name. If the section name is not recognized, the default will be for the section to have none of the above flags: it will not be allocated in memory, nor writable, nor executable. The section will contain data.

NOTE The assembler predefines several section names to support memory allocation in external memory (Section 2.4.4).

2.6.59 .set

```
.set symbol, expr
```

Set the value of *symbol* to *expression*. This changes *symbol*'s value and type to conform to *expression*. If *symbol* was flagged as external, it remains flagged (see Section 2.5.7).

You can set a symbol many times in the same assembly.

If you set a global symbol, the value stored in the object file is the last value stored into it.

2.6.60 .short

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

Generate the specified expressions as 16-bit numbers stored in consecutive addresses.

Accepts zero or more expressions separated by commas.

NOTE `.short` is equivalent to `.hword` (Section 2.6.34).

2.6.61 .single

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

Generate the specified *flonums* (Section 2.3.6) as 32-bit floating point numbers stored in consecutive addresses.

Accepts zero or more *flonums* separated by commas.

NOTE `.single` is equivalent to `.float` (Section 2.6.30).

2.6.62 .size

```
.size symbol, expr
```

Set the size of *symbol* to *expr*. The symbol size is expressed in bytes. *expr* is an absolute expression (but can include label arithmetic).

NOTE This directive is typically used to set the size of function symbols.

2.6.63 .sleb128

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

Generate the specified expressions as *sleb128*-format numbers stored in consecutive addresses.

Accepts one or more expressions separated by commas.

NOTE *sleb128* stands for "signed little-endian base 128." This is a compact, variable-length representation of numbers used by the DWARF symbolic debugging format.

2.6.64 .skip

```
.skip size, fill
```

Generate *size* bytes in consecutive addresses, each of value *fill*.

Both *size* and *fill* are absolute expressions. If the comma and *fill* are omitted, *fill* is assumed to be zero.

NOTE `.skip` is equivalent to `.space` (Section 2.6.65).

2.6.65 .space

```
.space size, fill
.block size, fill
```

Generate *size* bytes in consecutive addresses, each of value *fill*.

Both *size* and *fill* are absolute expressions. If the comma and *fill* are omitted, *fill* is assumed to be zero.

NOTE `.space` is equivalent to `.skip` (Section 2.6.64).

2.6.66 .stab

```
.stabd type, other, desc
.stabn type, other, desc, value
.stabs string, type, other, desc, value
```

Generate symbols (see Section 2.5) for use by symbolic debuggers. The symbols are not entered in the assembler hash table: they cannot be referenced elsewhere in the source file. Depending on the directive variant, up to five fields are required:

string	Symbol name. It can contain any character except `\000`, and thus is more generalized than ordinary symbol names.
type	Absolute expression. The symbol type is set to the low 8 bits of this expression. Any bit pattern is permitted, but the linker and debugger may not accept nonstandard bit patterns.
other	Absolute expression. The symbol's "other" attribute is set to the low 8 bits of this expression.
desc	Absolute expression. The symbol's descriptor is set to the low 16 bits of this expression.
value	Absolute expression which becomes the symbol's value.

If a warning is detected while reading a `.stabd`, `.stabn`, or `.stabs` statement, the symbol has probably already been created; you get a half-formed symbol in your object file. This is compatible with earlier assemblers!

```
.stabd type , other , desc
```

The "name" of the symbol generated is not even an empty string. It is a null pointer, for compatibility. Older assemblers used a null pointer so they didn't waste space in object files with empty strings. The symbol's value is set to the location counter, relocatably. When your program is linked, the value of this symbol is the address of the location counter when the `.stabd` was assembled.

```
.stabn type , other , desc , value
```

The name of the symbol is set to the empty string `""`.

```
.stabs string , type , other , desc , value
```

All five fields are specified.

2.6.67 .string

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

Generate the specified string literals (with automatic trailing zero byte) as character data stored in consecutive byte addresses.

Accepts one or more string literals (Section 2.3.7) separated by commas.

The strings can contain any of the escape sequences described in Section 2.3.7.

2.6.68 .string8

```
.string8 "string" [,"string"]...
```

Generate the specified string literals (with automatic trailing zero byte) as character data stored in consecutive byte addresses.

Accepts one or more string literals (Section 2.3.7) separated by commas.

The strings can contain any of the escape sequences described in Section 2.3.7.

2.6.69 .string16

```
.string16 "string" [,"string"]...
```

Generate the specified string literals (with automatic trailing zero byte) as character data, with each character value expanded to 16 bits and stored in consecutive halfword addresses.

Accepts one or more string literals (Section 2.3.7) separated by commas.

The strings can contain any of the escape sequences described in Section 2.3.7.

For example, assembling

```
.string16 "BABE"
```

is equivalent to assembling

```
.string "B\0A\0B\0E\0"
```

2.6.70 .string32

```
.string32 "string" [,"string"]...
```

Generate the specified string literals (with automatic trailing zero byte) as character data, with each character value expanded to 32 bits and stored in consecutive word addresses.

Accepts one or more string literals (Section 2.3.7) separated by commas.

The strings can contain any of the escape sequences described in Section 2.3.7.

2.6.71 .string64

```
.string64 "string" [,"string"]...
```

Generate the specified string literals (with automatic trailing zero byte) as character data, with each character value expanded to 64 bits and stored in consecutive doubleword addresses.

Accepts one or more string literals (Section 2.3.7) separated by commas.

The strings can contain any of the escape sequences described in Section 2.3.7.

2.6.72 .symver

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

The `.symver` directive has three variations. All bind symbols to specific version nodes in a source file (Section 3.3.12). This is typically done when assembling files that are to be linked into a shared object.

NOTE Cases also exist where it may make sense to use this directive in objects that are to be bound into an application itself, in order to override a versioned symbol from a shared object.

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

If the specified symbol `name` is defined in the file being assembled, this directive effectively creates a symbol alias with the name “`name2@nodename`”:

- In this case `name` serves as a name of convenience which makes it possible to have definitions for multiple versions of a function in a single source file, and which enables the compiler to determine which version of a function is being referenced.
- `name2` is the name by which the symbol is referenced externally.
- `nodename` specifies the name of a node in the version script supplied to the linker when building a shared library. (If you are overriding a versioned symbol from a shared object, then `nodename` should correspond to the node name of the symbol you are trying to override.)

If the specified symbol `name` is *not* defined in the file being assembled, this directive changes all references to `name` to “`name2@nodename`”.

NOTE The main reason for using `.symver` instead of a regular symbol alias is that the `@` character is not valid in symbol names.

If no references to `name` are made, “`name2@nodename`” will be removed from the symbol table.

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

The double-`@` version of `.symver` indicates that the symbol `name` must exist and be defined in the source file being assembled. The double-`@` version is functionally equivalent to the single-`@` version, except that "`name2@@nodename`" is also used by the linker to resolve references to `name2`.

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

The triple-`@` version specifies that when `name` is not defined in the file being assembled, it is treated as "`name2@nodename`". When `name` *is* defined in the file, it will be changed to "`name2@@nodename`".

2.6.73 .text

```
.text subsection
```

Assemble the following statements onto the end of the text subsection numbered *subsection*, which is an absolute expression. If *subsection* is omitted, subsection number zero is used.

2.6.74 .title

```
.title ["string"]
```

Set the page header title on an assembly list file (Section 2.2.5). If this directive is not specified (or if it is specified, but with the string argument omitted), the page header title line is set to blank.

This directive normally sets the title on the next page to be generated; however, if it appears within ten lines of the top of the page, it sets the title on the current page.

NOTE If multiple `.title` directives appear on a page, the first affects the current page and the last the next page. Any other instances in-between are ignored.

2.6.75 .type

```
.type symbol, type
```

Set the type of *symbol* to *type*. A symbol can be either a function symbol or an object symbol.

The *type* argument can contain one of the following constants:

`@function`	Symbol is function name.
`@object`	Symbol is data object.
`@gnu_indirect_function`	Symbol is indirect function name. When evaluated during relocation processing, symbols with this type are called rather than used as values.
`@tls_object`	Symbol is thread-local data object.
`@common`	Symbol is common data object.
`@notype`	Symbol has no type.
`@gnu_unique_object`	Symbol is globally unique in process. The dynamic linker verifies the uniqueness of symbols with this type.

2.6.76 .uleb128

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

Generate the specified expressions as *uleb128*-format numbers stored in consecutive addresses.

Accepts one or more expressions separated by commas.

NOTE *uleb128* stands for “unsigned little-endian base 128.” This is a compact, variable-length representation of numbers used by the DWARF symbolic debugging format.

2.6.77 .word

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

Generate the specified expressions as 32-bit data items stored in consecutive addresses.

Accepts zero or more expressions separated by commas.

NOTE `.word` is equivalent to `.int` (Section 2.6.37).

2.7 Preprocessing

The assembler internal preprocessor:

- adjusts and removes extra whitespace. It leaves one space or tab before the keywords on a line, and turns any other whitespace on the line into a single space.
- removes all comments, replacing them with a single space, or an appropriate number of newlines.
- converts character constants into the appropriate numeric values.

It does not do macro processing, include file handling, or anything else you may get from your C compiler's preprocessor. You can do include file processing with the `.include` directive (see Section 2.6.36). You can use the C compiler driver to perform other "CPP"-style preprocessing by assigning the input file a `'.S'` suffix. For more information see the *Hexagon GNU C/C++ Compiler: A GNU Manual*.

Excess whitespace, comments, and character constants cannot be used in the portions of the input text that are not preprocessed.

If the first line of an input file is `#NO_APP` or if you use the `'-f'` option, whitespace and comments are not removed from the input file. Within an input file, you can specify whitespace and comment removal in specific parts of the file by inserting a line containing the directive `#APP` before the text that may contain whitespace or comments, and inserting another line containing the directive `#NO_APP` after the text. This feature is mainly intended to support `asm` statements in compilers whose output is otherwise free of comments and whitespace.

2.8 Processor-specific features

The assembler supports the following features which are specific to the Hexagon processor:

- It predefines memory sections for the processor-specific memories (Section 2.4.4) and global data area (Section 2.4.3).
- It checks for various processor restrictions (e.g., loop and branch immediate cannot be in the same instruction packet; read-only/write-only access).
- It encodes instruction packets so that each instruction is assigned to the proper slot.
- It pads loop-end instruction packets with NOPs to ensure the minimum required size *(e.g.,* at least 2 instructions for `endloop0`, and at least 3 for `endloop1`).
- It defines optional alignment and access attributes for common symbols.
- Assembler options:
  - `--mfalign-info` (Section 2.2.2.1)
  - `-G` (Section 2.2.2.6)
  - `--mno-extender`, `--mno-jumps`, `--mno-jumps-long`, `--mno-pairing`, `--mno-pairing-duplex`, `--mno-pairing-branch`, `--mpairing-info` (Section 2.2.2.6)
  - `-march`, `-mcpu`, `-mv2`, `-mv3` , `-mv4` (Section 2.2.2.8)
- Assembler directives:
  - `.comm` (Section 2.6.12)
  - `.falign` (Section 2.6.27)
  - `.lcomm` (Section 2.6.41)
  - `.section` (Section 2.6.58)

NOTE The assembler does *not* support symbol assignment of the form "*`symbol = value`*". Symbols must be assigned using the `.set` directive (Section 2.6.59).

3 Linker

3.1 Overview

The linker combines a number of object and archive files, relocates their data, and resolves symbol references.

The linker accepts command files (known as *linker scripts*) to provide users with control over the linking process.

The linker predefines symbols to provide users with control over how linked programs are assigned to the Hexagon processor memories.

The linker can read, combine, and write object files in different formats: for example, ELF or `a.out`. Different formats can be linked together to produce any available kind of object file.

Aside from its flexibility, the linker is helpful in providing diagnostic information. Whenever possible, it continues executing after encountering an error, allowing you to identify other errors (or, in some cases, to generate an output file in spite of the error).

This chapter covers the following topics:

- Using the linker
- Linker scripts
- Processor-specific memory layout
- Binary file access
- Processor-specific features

3.2 Using the linker

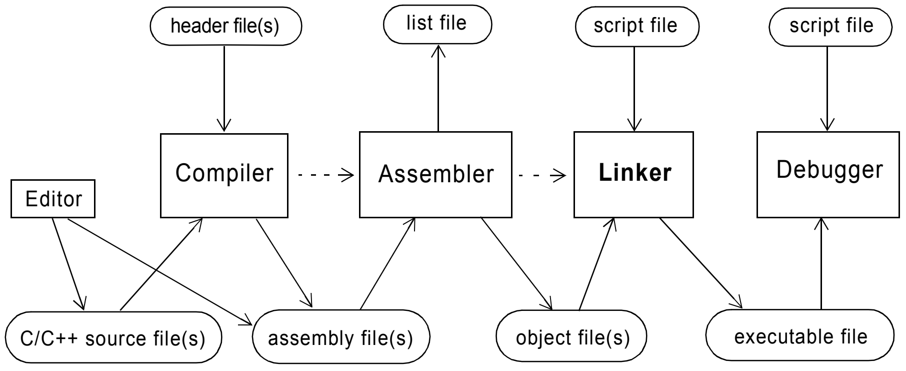

Figure 3-1 Using the linker

The linker links object files into executable user applications. It inputs one or more *object files* and a *script file*, and outputs an *executable file*.

Linker script files (also known as *command files*) specify how the input files are to be linked.

3.2.1 Starting the linker

To start the linker from a command line, type:

```
hexagon-ld [option...] [input_file...]
```

The linker accepts one or more source files as the assembly language input. The file names must include the name extension, if any.

Command switches are used to control various linker options. A switch consists of one or two dash characters followed by a switch name and optional parameter.

NOTE Switch names are case-sensitive. Switches must be separated by at least one space.

Non-option arguments are (with one exception noted below) treated as object files which are to be linked together. They can follow, precede, or be intermixed with command-line options, except that an object file argument cannot be placed between an option and its argument.

Usually the linker is invoked with at least one object file specified as a non-option argument; however, object files can also be specified with the `-l` or `-R` options, or in the linker script. If *no* object file is specified, the linker does not produce any output and issues the message “`No input files`”.

If the linker cannot recognize a linker input file as a valid object file, it assumes the file contains a linker script. A script specified in this way is defined to augment the main linker script (i.e., either the default script (Section 3.4), or the one specified by the `-T` option). This feature permits the linker to link files which appear to be object or archive files, but in fact merely define some symbol values or use the linker commands `INPUT` or `GROUP` to load other objects.

NOTE Specifying a script in this way should only be done to augment the main linker script. If you want to specify a linker command that can logically appear only once (e.g., `SECTIONS` or `MEMORY`), the corresponding script must replace the default linker script (by using `-T`).

The linker generates an error message if a file is linked multiple times.

Option files

Linker command arguments can be specified in a text file rather than on the command line. The file is specified on the command line as the argument of the special command option `@`. For example:

```
hexagon-ld @myoptions
```

Here the file named *myoptions* contains the command arguments for the `hexagon-ld` command. The file contents are inserted into the command line in place of the specified `@`*file* option.

NOTE Command arguments stored in argument files must be delimited by whitespace characters.

The @file option can appear in argument files – it is processed recursively.

Option help

To list the available command options, type:

```
hexagon-ld --help
```

The linker displays on the screen the proper command line syntax, followed by a list of the available command options.

3.2.2 Linker options

Linker options are used to control various linker features from the command line.

The linker supports many command-line options, but in practice only a few of them are used regularly. For example:

```
hexagon-ld -o output /lib/crt0.o hello.o -lc -Map mapfile
```

This tells `t`he linker to create a file named *output* which is the result of linking the file `hello.o` with `/lib/crt0.o` and the library `libc.a` (for more information on libraries see the `-l` option below). The `-Map` option generates a link map file named *mapfile* which describes how the files were linked.

The linker options can be specified in any order, and can be repeated at will. Repeating most options with a different argument will either have no further effect, or override prior occurrences (those further to the left on the command line) of that option. Options that can be meaningfully specified more than once are noted in the descriptions below.

For options whose names are a single letter, option arguments must either follow the option letter without intervening whitespace, or be specified as separate arguments immediately following the option that requires them.

Many options have alternate abbreviated switches defined for ease of use. These long and short forms (listed below as alternatives) are functionally equivalent.

The standard option names can also be truncated, as long as you specify enough of the option name to uniquely identify the option.

NOTE Option names can be prefixed with either "`-`" or "`--`". However, one exception exists to this rule: multiple-letter options that start with a lower-case 'o' must always be preceded by two dashes.

This restriction is intended to reduce confusion with the `-o` option. For example, `-omagic` sets the output file name to `magic`, whereas `--omagic` sets the `NMAGIC` flag on the output.

The linker options are specified by the command switches listed below.

```
-akeyword
--add-needed
--allow-multiple-definitions
--allow-shlib-undefined
--as-needed
-b input-format | --format input-format
-Bdynamic | -dy | -call_shared
-Bgroup
-Bshareable | -shared
-Bstatic | -dn | -non_shared | -static
-Bsymbolic
-Bsymbolic-functions
--build-id[=style]
--check-sections
--cref
```

-d | **-dc** | **-dp**
-dT *file* | **--default-script** *file*
--defsym *symbol=expression*
--demangle
--disable-new-dtags
--dynamic-linker *file*
--dynamic-list *file*
--dynamic-list-cpp-new
--dynamic-list-cpp-typeinfo
--dynamic-list-data
-e *entry* | **--entry** *entry*
-E | **--export-dynamic**
-EB
--eh-frame-hdr
-EL
--enable-new-dtags
--error-unresolved-symbols
--exclude-libs *lib* [,*lib*]
-f | **--auxiliary** *name*
-F *name* | **--filter** *name*
--fatal-warnings
-fini *name*
--force-dynamic
--force-exe-suffix
-g
-G *size* | **--gpsize** *size*
--gc-sections
-h *name* | **-soname** *name*
--hash-size=*number*
--help
-i
-init *name*
-l[:]*archive* | **--library** *archive*
-L*searchdir* | **--library-path** *searchdir*
-m*emulation*
-M | **--print-map**
-Map *file*
-march *archname*
-mcpu *archname*
-mv2
-mv3
-mv4
-n | **--nmagic**
-N | **--omagic**
--no-add-needed
--no-as-needed
--no-allow-shlib-undefined
--no-check-sections
--no-define-common
--no-demangle
--no-export-dynamic
--no-fatal-warnings
--no-gc-sections
--no-keep-memory
--no-omagic
--no-print-gc-sections

--no-undefined
--no-undefined-version
--no-warn-mismatch
--no-whole-archive
--noinhibit-exec
--nostdlib
-o *output* | **--output** *output*
-O *level*
--oformat *output-format*
--print-gc-sections
-q | **--emit-relocs**
-r | **--relocateable**
-R *file* | **--just-symbols** *file*
--reduce-memory-overheads
--retain-symbols-file *file*
-rpath *dir*
-rpath-link *DIR*
-s | **--strip-all**
-S | **--strip-debug**
--section-start *section=org*
--sort-common[=(**ascending**|**descending**)]
--sort-section=(**name**|**alignment**)
--split-by-file [*size*]
--split-by-reloc [*count*]
--stats
--sysroot=*directory*
-t | **--trace**
-T *file* | **--script** *file*
--target-help
-Tbss *org* | **-Tdata** *org* | **-Ttext** *org*
-tcm
-Ttext-segment=*addr*
--traditional-format
--trampolines[=(**yes**|**no**)]
-u *symbol* | **--undefined** *symbol*
--unique[=*section*]
--unresolved-symbols=*method*
-Ur
-v | **--version** | **-V**
--verbose | **--dll-verbose**
--version-script *file*
--warn-alternate-em
--warn-unresolved-symbols
--warn-common
--warn-once
--warn-section-align
--warn-shared-textrel
--whole-archive
--wrap *symbol*
-x | **--discard-all**
-X | **--discard-locals**
-y *symbol* | **--trace-symbol** *symbol*
-Y *path*
-z *keyword*
-(*archive* **... -)** | **--start-group** *archive* **... --end-group**
@*file*

`-a`*`keyword`*

The `-a` option is supported for HP/UX compatibility. The *keyword* argument must be one of the following strings: `archive`, `shared`, or `default`. `-aarchive` is functionally equivalent to `-Bstatic`, and the other two keywords are functionally equivalent to `-Bdynamic`. This option can be used any number of times.

`--add-needed`

Add `DT_NEEDED` tags for each shared library from `DT_NEEDED` tags. This option is the default setting.

`--allow-multiple-definition`

Allow multiple definitions of a symbol to exist, and use the first of the definitions to resolve any references to the symbol. Normally the linker flags multiple symbol definitions with an error.

Identical to `-z muldefs` below.

`--allow-shlib-undefined`

Allow any undefined symbols in shared libraries. This is the default setting.

This option is similar to `--no-undefined` except that it determines the behavior when the undefined symbols are in a shared library rather than a regular object file. It does not affect how undefined symbols in regular object files are handled.

This option is the default setting because a shared library specified at link time may not be the same as the one available at load time; so the symbols might actually be resolvable at load time.

`--as-needed`

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

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

`-b` *`input-format`*
`--format` *`input-format`*

The linker can be configured to support more than one kind of object file. If your linker is configured this way, you can use the `-b` option to specify the binary format for input object files that follow this option on the command line. Even when the linker is configured to support alternative object formats, you do not usually need to specify this, as the linker should be configured to expect as a default input format the most usual format on each machine.

input-format is a text string, the name of a particular format supported by the BFD libraries. (You can list the available binary formats with `objdump -i`.) See Section 3.5.

You can also use `-b` to switch formats explicitly (when linking object files of different formats), by including `-b` *`input-format`* before each group of object files in a particular format. The default format is taken from the environment variable `GNUTARGET` (Section 3.2.3). You can also define the input format from a script using the command `TARGET` (Section 3.3.13).

NOTE This option is useful when linking files with unusual binary formats.

`-Bdynamic`
`-dy`
`-call_shared`

Link against shared libraries. The different variants of this option are for compatibility with various systems. You can use this option multiple times on the command line. It affects library searching for the `-l` options that follow it.

`-Bgroup`

Select group name lookup rules for dynamic objects.

Sets `DF_1_GROUP` flag in the `DT_FLAGS_1` entry in the dynamic section of the ELF dynamic object. This causes the dynamic linker to handle lookups in the dynamic object, and dependencies to be performed only inside the group.

`-Bshareable`
`-shared`

Create a shared library.

`-Bstatic`
`-dn`
`-non_shared`
`-static`

Do not link against shared libraries. The different variants of this option are for compatibility with various systems. You can use this option multiple times on the command line. It affects library searching for `-l` options that follow it.

`-Bsymbolic`
When creating a shared library, bind global symbol references to the symbol definition contained in the shared library (if any). Programs linked against shared libraries can normally override symbol definitions contained in the library.

`-Bsymbolic-functions`
When creating a shared library, bind global function symbol references to the symbol definition contained in the shared library (if any).

`--build-id``[=`*`style`*`]`
Specify the creation of a `.note.gnu.build-id` ELF note section. The contents of the note are unique bits identifying the linked file. *style* can be one of the following values:

`uuid`

Use 128 random bits.

`sha1`

Use a 160-bit SHA1 hash on normative parts of the output contents (default).

`md5`

Use a 128-bit MD5 hash on normative parts of the output contents.

`0xhexstring`

Use an arbitrary bit string specified as an even number of hexadecimal digits (ignoring "`-`" or "`:`" characters between the digit pairs).

`none`

Disable settings established by any `--build-id` options that were specified previously on the linker command line.

NOTE `sha1` and `md5` produce identifiers that are always the same in identical output files, but unique among all nonidentical output files. Thus the section contents should not be compared as a checksum for the file contents.

Linked files can be subsequently modified by other tools, but the build ID bit string that identifies the original linked file will not change.

`--check-sections`
Check section addresses after they have been assigned to see if any overlaps exist. If overlaps are found an error message is generated. This is the default setting.

The linker knows about (and makes allowances for) any sections in overlays. Also, see `--no-check-sections`.

`--cref`
Output a cross reference table. If a linker map file is being generated, the cross reference table is printed to the map file. Otherwise, it is printed on the standard output. The format of the table is intentionally simple so it can be easily processed by a script, if necessary. The symbols are printed out and sorted by name. For each symbol, a list of file names is given. If the symbol is defined, the first file listed is the location of the definition. The remaining files contain references to the symbol.

`-d`
`-dc`
`-dp`

These three options are equivalent; multiple forms are supported for compatibility with other linkers. They assign space to common symbols even if a relocatable output file is specified (with `-r`). The script command `FORCE_COMMON_ALLOCATION` has the same effect (Section 3.3.13).

`-dT` *`file`*
`--default-script` *`file`*

Use the specified file as the default linker script (Section 3.3), but also enable any command options specified on the command line after this option to affect the behavior of the linker script. This is useful when linker commands are auto-generated by tools such as `gcc`, and thus are not directly controllable by the user.

This option is similar to `--script`, except that it delays processing of the script until the rest of the command line is processed.

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

Create a global symbol in the output file, containing the absolute address given by *expression*. You can use this option as many times as necessary to define multiple symbols in the command line. A limited form of arithmetic is supported for the *expression* in this context: you can give a hexadecimal constant or the name of an existing symbol, or use `+` and `-` to add or subtract hexadecimal constants or symbols. If you need more elaborate expressions, consider using the linker command language from a script (Section 3.3.7).

NOTE There should be no whitespace between *symbol*, the equals sign (`=`), and *expression*.

`--demangle``[=`*`style`*`]`

Demangle symbol names in error messages and other output. When the linker is instructed to demangle, it tries to present symbol names in a readable fashion: it strips leading underscores if they are used by the object file format, and converts C++ mangled symbol names into user readable names.

Different compilers have different mangling styles. The optional `style` argument can be used to choose an appropriate demangling style for your compiler.

By default the linker performs demangling unless the environment variable `COLLECT_NO_DEMANGLE` is set. This option overrides the environment variable.

See also `--no-demangle` below.

`--disable-new-dtags`

Do not create new dynamic tags in the ELF executable. This is the default setting.

See also `--enable-new-dtags` below.

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

Set the name of the dynamic linker. This is only meaningful when generating dynamically-linked ELF executables.

NOTE The default dynamic linker is normally correct – do not use this option unless you know what you are doing.

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

Specify the name of a *dynamic list* file, which contains a list of symbols to be processed by the linker.

This option is used to create shared libraries that contain global symbols whose references must not be bound to the definitions within the shared library. It is also used to create dynamically linked executables that specify a list of symbols to add to the executable's symbol table.

The contents of a dynamic list file have the same format as a version node (Section 3.3.12), but without the scope and node name.

`--dynamic-list-cpp-new`

Specify the built-in dynamic list for the C++ `new` and `delete` operators. This option is useful for building shared versions of `libstdc++`. For details see `--dynamic-list`.

`--dynamic-list-cpp-typeinfo`

Specify the built-in dynamic list for C++ runtime type identification. For details see `--dynamic-list`.

`--dynamic-list-data`

Specify all the global data symbols as part of the dynamic list. For details see `--dynamic-list`.

`-e` *`entry`*
`--entry` *`entry`*

Use *entry* as the explicit symbol for beginning execution of your program, rather than the default entry point. See Section 3.3.10 for a discussion of defaults and other ways of specifying the entry point.

`--exclude-libs` *`lib`* `[,`*`lib`*`]`

Do not automatically export symbols from the specified archive libraries. Specifying the symbol `ALL` as a library argument affects the symbols in all archive libraries. Symbols affected by this option are treated as hidden.

`-E`
`--export-dynamic`

When creating a dynamically linked executable, add all symbols to the dynamic symbol table. The dynamic symbol table is the set of symbols visible from dynamic objects at run time. If this option is not used (or if the default setting is restored with `--no-export-dynamic`), the dynamic symbol table normally contains only those symbols referenced by dynamic objects explicitly specified in the link.

If `dlopen` is used to load a dynamic object that needs to refer back to the symbols defined by the program (rather than those defined by some other dynamic object), this option will probably need to be used when linking the program itself. See also `--dynamic-list`.

NOTE This option implicitly asserts `--force-dynamic` to ensure that a dynamic section is created even when no dynamic libraries are linked to a program.

-EB

Link big-endian objects. This affects the default output format.

--eh-frame-hdr

Create `.eh_frame_hdr` section and `PT_GNU_EH_FRAME` segment header in the dynamically-linked ELF executable.

-EL

Link little-endian objects. This affects the default output format.

--enable-new-dtags

Create new dynamic tags in the ELF executable.

--error-unresolved-symbols

Generate errors for any unresolved symbols. This option is the default setting.

-f
--auxiliary *name*

When creating a shared object, set the internal `DT_AUXILIARY` field to the specified name. This informs the dynamic linker that the symbol table of the shared object should be used as an auxiliary filter on the symbol table of the shared object *name*. If you later link a program against this filter object, when you run the program the dynamic linker will see the `DT_AUXILIARY` field. If the dynamic linker resolves any symbols from the filter object, it will first check whether there is a definition in the shared object *name*. If there is one, it will be used instead of the definition in the filter object. The shared object *name* need not exist. Thus, the shared object *name* can be used to provide an alternative implementation of certain functions, perhaps for debugging or for machine-specific performance.

NOTE This option can be specified more than once. The `DT_AUXILIARY` entries are created in the order in which they appear on the command line.

-F *name*
--filter *name*

When creating a shared object, set the internal `DT_FILTER` field to the specified name. This tells the dynamic linker that the symbol table of the shared object which is being created should be used as a filter on the symbol table of the shared object *name*. If you later link a program against this filter object, when you run the program, the dynamic linker will see the `DT_FILTER` field. The dynamic linker will resolve symbols according to the symbol table of the filter object as usual, but it will actually link to the definitions found in the shared object *name*. Thus, the filter object can be used to select a subset of the symbols provided by the object *name*. Some older linkers used the `-F` option throughout a compilation tool chain for specifying object-file format for both input and output object files. The GNU linker uses other mechanisms for this purpose: the `-b`, `--format`, `--oformat` options, the `TARGET` command in linker scripts, and the `GNUTARGET` environment variable. The GNU linker will ignore the `-F` option when not creating a shared object.

`--fatal-warnings`

Treat all warnings as errors.

See also `--no-fatal-warnings` below.

`-fini` *`name`*

When creating an ELF executable or shared object, call the specified name when the executable or shared object is unloaded, by setting `DT_FINI` to the address of the function. The default name is `_fini`.

`--force-dynamic`

Force the output file to include dynamic sections.

`--force-exe-suffix`

Make sure that an output file has a `.exe` suffix. If a successfully built, fully linked output file does not have a `.exe` or `.dll` suffix, this option forces the linker to copy the output file to one of the same name with a `.exe` suffix. This option is useful when using unmodified UNIX makefiles on a Microsoft Windows host, since some versions of Windows will not run an image unless it ends in a `.exe` suffix.

`-g`

Ignored. Provided for compatibility with other tools.

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

Allocate common symbols in the global bss section (Section 2.4.3) if they are less than or equal to the specified size. The size is expressed in bytes. The default is 8.

Common symbols are declared with the assembler directives `.comm` and `.lcomm` (Section 2.6).

The assembler also supports the `-G` option (Section 2.2.2.6). If the assembler assigns a common symbol to the global bss section, the linker will not remove the symbol from that section even if the linker's `-G` option specifies a size smaller than the common symbol. For example:

```
.comm foo, 8
.comm bat, 16

hexagon-as -mv2 -G 8 -c foo.s -o foo.o
hexagon-ld -mv2 -G 4 foo.o -o foo
```

During assembly the common symbol `foo` is assigned to the global bss section by the assembler option `-G 8`. Therefore, during linking `foo` remains assigned to the global bss section even though it exceeds the size specified in the linker (`-G 4`).

If the link command was changed to the following:

```
  hexagon-ld -mv2 -G 16 foo.o -o foo
```

... then because of the larger size limit (-G 16) the common symbol `bat` defined above would be assigned to the global bss section during linking (in spite of being too big to be assigned during assembly).

`--gc-sections`

Remove all unused input sections from the output file (a process known as *garbage collection*).

This option is used in conjunction with the compiler options `-ffunction-sections` and `-fdata-sections`. For more information see *Hexagon GNU C/C++ GNU Compiler: A GNU Manual*.

NOTE This option is not compatible with the `-r` option.

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

When creating a shared object, set the internal DT_SONAME field to the specified name. When an executable is linked with a shared object that has a DT_SONAME field and then the executable is run, the dynamic linker will attempt to load the shared object specified by the DT_SONAME field rather than using the file name given to the linker.

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

Set the default size of the linker's hash tables to a prime number close to the specified value. Increasing *number* decreases link times at the expense of increased memory usage. Conversely, reducing *number* can reduce the memory usage at the expense of speed.

The default value is 1021 if `--reduce-memory-overloads` is used and `--hash-size` is not used.

`--help`

Print a summary of the command-line options on the standard output and exit. No input file is specified when this option is used.

`-i`

Perform an incremental link (same as option `-r`).

`-init` *`name`*

When creating an ELF executable or shared object, call the specified name when the executable or shared object is unloaded, by setting `DT_INIT` to the address of the function. The default name is `_init`.

`-l``[`**`:`**`]`*`archive`*
`--library` `[`**`:`**`]`*`archive`*

Add archive file *archive* to the list of files to link. This option can be used any number of times.

If the optional colon prefix (:) is specified in the file name, the linker will search its path-list for occurrences of the file name "*archive*".

If a : prefix is *not* specified, the linker will search its path-list first for the file name "`lib`*`archive`*`.so`", and then for "`lib`*`archive`*`.a`".

By convention, a `.so` extension indicates a shared library. The linker searches an archive only once, at the location where it is specified on the command line. If the archive defines a symbol that was undefined in some object which appeared before the archive on the command line, the linker will include the appropriate file(s) from the archive. However, an undefined symbol in an object appearing later on the command line will not cause the linker to search the archive again. See the `'-('` option for a way to force the linker to search archives multiple times. You can list the same archive multiple times on the command line. This type of archive searching is standard for UNIX linkers.

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

Add path *searchdir* to the list of paths that the linker will search for archive libraries and linker control scripts. You can use this option any number of times. The directories are searched in the order in which they are specified on the command line. Directories specified on the command line are searched before the default directories. All `-L` options apply to all `-l` options, regardless of the order in which the options appear. The default set of paths searched (without being specified with `-L`) depends on which emulation mode the linker is using, and in some cases also on how it was configured. See Section 3.2.3.

The paths can also be specified in a link script with the `SEARCH_DIR` command. Directories specified this way are searched at the point in which the linker script appears in the command line.

`-m`*`emulation`*

Emulate the *emulation* linker. You can list the available emulations with the `--verbose` or `-V` options. If the `-m` option is not used, the emulation is taken from the `LDEMULATION` environment variable, if that is defined. Otherwise, the default emulation depends upon how the linker was configured.

-M
--print-map

Print a link map to the standard output. A link map provides information about the link, including the following:

Where object files and symbols are mapped into memory.

How common symbols are allocated.

All archive members included in the link, with a mention of the symbol that caused the archive member to be brought in.

The values assigned to symbols.

NOTE Symbol values displayed in square brackets in a link map indicate that the symbol value was computed by an expression which included a reference to the previous value of the same symbol. Such values can be counter-intuitive.

-Map *mapfile*

Print a link map to the file *mapfile*. See the description of the `-M` option above.

-march *archname*
-mcpu *archname*

Specify the target Hexagon processor architecture (i.e., the instruction set) of the output file, which can be `hexagonv2`, `hexagonv3`, or `hexagonv4`. All input files must have been generated with the specified architecture. The default is the architecture specified in the first input file.

-mv2

Equivalent to: `-march hexagonv2`

-mv3

Equivalent to: `-march hexagonv3`

-mv4

Equivalent to: `-march hexagonv4`

-n
--nmagic

Set the text segment to be read-only, and mark the output as `NMAGIC,` if possible.

-N
--omagic

Set the text and data sections to be readable and writable. Also, do not page-align the data segment. If the output format supports UNIX-style magic numbers, mark the output as `OMAGIC`.

--no-allow-shlib-undefined

Do not allow undefined symbols in shared object libraries. `--allow-shlib-undefined` restores the default setting.

--no-add-needed

Do not add `DT_NEEDED` tags for each shared library that comes from `DT_NEEDED` tags. Normally, the linker adds `DT_NEEDED` tags for such libraries. `--add-needed` restores the default behavior.

`--no-as-needed`
Do not limit the addition of `DT_NEEDED` tags to libraries that satisfy a symbol reference which is undefined when the library was linked. This option is the default setting.

`--no-check-sections`
Do not check section addresses for overlaps.

`--no-define-common`
Inhibit the assignment of addresses to common symbols. The script command `INHIBIT_COMMON_ALLOCATION` has the same effect (Section 3.3.13).

This option enables users to decouple the decision to assign addresses to common symbols from the choice of output file type; otherwise a non-relocatable output type forces the assignment of addresses to common symbols.

Using this option allows common symbols that are referenced from a shared object to be assigned addresses only in the main program.This eliminates the unused duplicate space in the shared object, and also prevents any possible confusion over resolving to the wrong duplicate when many dynamic modules exist with specialized search paths for runtime symbol resolution.

`--no-demangle`
Do not demangle symbol names in error messages and other output.

`--no-export-dynamic`
Do not add all symbols to the dynamic symbol table when creating a dynamically linked executable.

`--no-fatal-warnings`
Do not treat warnings as errors. This is the default setting.

`--no-gc-sections`
Do not remove unused input sections from the output file.

`--no-keep-memory`
The linker normally optimizes for speed over memory usage by caching the symbol tables of input files in memory. This option tells the linker to instead optimize for memory usage by rereading the symbol tables, as necessary. This may be required if the linker runs out of memory space while linking a large executable.

`--no-omagic`
Negate most effects of the `-N` option. Set the text section to be read-only, and force the data segment to be page-aligned.

NOTE This option does not enable linking against shared object libraries. To do this use the `-Bdynamic` option.

`--no-print-gc-sections`
Do not list to *stderr* all sections that were removed by garbage collection during linking.

`--no-undefined`

Report unresolved symbol references from regular object files (even if the linker is creating a shared object).

The `--allow-shlib-undefined` option controls the reporting of unresolved references when linking in shared object libraries.

Identical to `-z defs` below.

`--no-undefined-version`

Disallow symbols with undefined versions; if one is detected, generate a fatal error. Normally the linker ignores symbols with undefined versions.

`--no-warn-mismatch`

Normally, the linker will give an error if you try to link together input files that are mismatched for some reason, perhaps because they have been compiled for different processors or for different endian-nesses. This option tells the linker that it should silently permit such possible errors. This option should only be used with care, in cases when you have taken some special action that ensures the linker errors are inappropriate.

`--no-whole-archive`

Disable the effect of the `--whole-archive` option for subsequent archive files specified on the command line.

`--noinhibit-exec`

Retain the executable output file whenever it is still usable. Normally, the linker will not produce an output file if it encounters errors during the link process; it exits without writing an output file when it issues any error, whatsoever.

`-nostdlib`

Only search library directories explicitly specified on the command line. Library directories specified in linker scripts (including linker scripts specified on the command line) are ignored.

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

Use *output* as the name for the program produced by the linker; if this option is not specified, the name `a.out` is used by default. The script command `OUTPUT` can also specify the output file name.

`-O` *`level`*

Optimize the linker output when the specified numeric value is greater than 0. This option affects only shared library generation.

NOTE This option may significantly increase the link time – it should therefore be used only when linking a final binary.

`--oformat` *`output-format`*

The linker can be configured to support more than one kind of object file. If your linker is configured this way, you can use the `--oformat` option to specify the binary format for the output object file. Even when the linker is configured to support alternative object formats, you do not usually need to specify this, as the linker should be configured to produce as a default output format the most usual format on each machine. *output-format* is a text string, the name of a particular format supported by the BFD libraries. (You can list the available binary formats with `objdump -i`.) The script command `OUTPUT_FORMAT` can also specify the output format, but this option overrides it. See Section 3.5.

`--print-gc-sections`

List to *stderr* all sections that were removed by garbage collection during linking. This option is effective only if garbage collection has been enabled with `--gc-sections`. The default setting can be restored with `--no-print-gc-sections`.

`-q`
`--emit-relocs`

Retain relocation sections and contents in fully-linked executables. Post-link analysis and optimization tools may need this information to perform correct modification of executables. This option results in larger executables.

`-r`
`--relocateable`

Generate relocatable output (i.e., generate an output file that can in turn serve as input to the linker). This is often called *partial linking*. As a side effect, in environments that support standard UNIX magic numbers, this option also sets the output file's magic number to `OMAGIC`. If this option is not specified, an absolute file is produced. When linking C++ programs, this option *will not* resolve references to constructors (to do that, use `-Ur`). This option is equivalent to `-i`.

`-R` *`file`*
`--just-symbols` *`file`*

Read symbol names and their addresses from the specified file, but do not relocate them or include them in the output. This allows your output file to refer symbolically to absolute memory locations defined in other programs. For compatibility with other ELF linkers, if the `-R` option is followed by a directory name, rather than a file name, it is treated as the `-rpath` option.

NOTE This option can be specified more than once.

`--reduce-memory-overheads`

Reduce link memory usage at the expense of speed. This option deploys an $O(n^2)$ algorithm for link map file generation, rather than the default O(n) algorithm which uses about 40% more memory for symbol storage.

This option also sets the default hash table size to `1021`, which saves memory at the expense of link time. The hash table size can be overridden with the `--hash-size` option.

--retain-symbols-file *file*

Retain *only* the symbols listed in the specified file, discarding all others. *file* is simply a flat file, with one symbol name per line. This option is especially useful in environments where a large global symbol table is accumulated gradually, to conserve run-time memory. `--retain-symbols-file` does *not* discard undefined symbols, or symbols needed for relocations. You can only specify `--retain-symbols-file` once in the command line. It overrides `-s` and `-S`.

-rpath *dir*

Add a directory to the runtime library search path. This is used when linking an ELF executable with shared objects. All `-rpath` arguments are concatenated and passed to the runtime linker, which uses them to locate shared objects at runtime. The `-rpath` option is also used when locating shared objects which are needed by shared objects explicitly included in the link; see the description of the `-rpath-link` option. If `-rpath` is not used, the contents of the environment variable `LD_RUN_PATH` will be used if it is defined. The `-rpath` option can also be used on SunOS. By default, on SunOS, the linker will form a runtime search patch out of all the `-L` options it is given. If a `-rpath` option is used, the runtime search path will be formed exclusively using the `-rpath` options, ignoring the `-L` options. This can be useful when using GCC, which adds many `-L` options that may be on NFS-mounted file systems. For compatibility with other ELF linkers, if the `-R` option is followed by a directory name, rather than a file name, it is treated as the `-rpath` option.

-rpath-link *dir*

One shared object may require another. This happens when an the linker `-shared` link includes a shared object as one of the input files. When the linker encounters such a dependency when doing a non-dynamic, non-relocatable link, it will automatically try to locate the required shared object and include it in the link, if it is not included explicitly. In such a case, the `-rpath-link` option specifies the first set of directories to search. The option can specify a sequence of directory names either by specifying a list of names separated by colons, or by appearing multiple times. The linker uses the following search paths to locate required shared libraries:

1. Any directories specified by `-rpath-link` options.
2. Any directories specified by `-rpath` options. The difference between `-rpath` and `-rpath-link` is that directories specified by `-rpath` options are included in the executable and used at runtime; whereas, the `-rpath-link` option is only effective at link time.
3. If the `-rpath` and `rpath-link` options were not used, search the contents of the environment variable `LD_RUN_PATH`.
4. For a native linker, the contents of the environment variable `LD_LIBRARY_PATH`.
5. The default directories, normally `/lib` and `/usr/lib`.
6. If the required shared object is not found, the linker will issue a warning and continue with the link.

`-s`
`--strip-all`
Omit all symbol information from the output file.

`-S`
`--strip-debug`
Omit debugger symbol information (but not all symbols) from the output file.

`--section-start` *`section=org`*
Locate a section in the output file at the absolute address specified by *org*. This option can be repeated on the command line to locate multiple sections. *org* must be a single hexadecimal integer. For compatibility with other linkers, you can omit the leading `0x` usually associated with hexadecimal values.

NOTE No whitespace can appear between *section*, "=", and *org*.

`--sort-common``[=(`**`ascending`**`|`**`descending`**`)]`
Sort common symbols (by size) in the specified order in the appropriate output sections. The default sort order is descending.

NOTE Sorting the symbols in a section prevents gaps from occurring between the symbols due to alignment constraints.

`--sort-section=``(`**`name`**`|`**`alignment`**`)`
Apply the sort criteria `SORT_BY_NAME` or `SORT_BY_ALIGNMENT` to all wildcard section patterns in the linker script.

`--split-by-file` `[`*`size`*`]`
Similar to `--split-by-reloc` but creates a new output section for each input file when the specified size is reached. The default size is 1.

`--split-by-reloc` `[`*`count`*`]`
Attempt to create extra sections in the output file so that no single output section in the file contains more than the specified number of relocations. The linker does not split up individual input sections for redistribution, so if a single input section contains more than the specified number of relocations, one output section will contain that many relocations. The default relocation count is 32768.

`--stats`
Compute and display statistics about the operation of the linker, such as execution time and memory usage.

`--sysroot=`*`directory`*
Specify the directory location of `sysroot`, overriding the configure-time default. This option is supported only on toolchains configured with `--with-sysroot`.

`-t`
`--trace`
Print the names of the input files as the linker processes them.

`-T` *`commandfile`*
`--script` *`commandfile`*

Read link commands from the file *commandfile*. These commands replace `the` default linker script (Section 3.4) rather than adding to it, so *commandfile* must specify everything necessary to describe the target format. You must use this option if you want to use a command that can only appear once in a linker script, such as the `SECTIONS` or `MEMORY` command (Section 3.3).

If *commandfile* does not exist, the linker searches for it in the directories specified by any preceding `-L` options. Multiple `-T` options are cumulative.

`--target-help`

Print summary of all target-specific command-line options to the standard output and exit. No input file is specified when this option is used.

`-Tbss` *`org`*
`-Tdata` *`org`*
`-Ttext` *`org`*

Use *org* as the starting address for, respectively, the `bss`, `data`, or the `text` segment of the output file. *org* must be a single hexadecimal integer. For compatibility with other linkers, you can omit the leading `0x` usually associated with hexadecimal values.

`-Ttext-segment=`*`addr`*

When creating an ELF executable or shared object, set the address of the first byte of the text segment to the specified address.

`-tcm`

Define memory sections for the Hexagon processor-specific memories (Section 2.4.4) using the linker's predefined memory address symbols (Section 3.4.2). See Section 3.4.1.

`--traditional-format`

For some targets, the output of the linker is different in some ways from the output of some existing linker. This switch requests the linker to use the traditional format instead.

`--trampolines``[=(`**`yes`**`|`**`no`**`)]`

Generate an indirect branch whenever the destination address of a program branch is too far away to be reached by a regular branch instruction. The default (for both the arguments and the option itself) is `yes`.

NOTE This option may not work when using the `-r` option. If this is the case, try using the compiler option `-ffunction-sections` along with this option. For more information see *Hexagon GNU C/C++ Compiler: A GNU Manual*.

`-u` *`symbol`*
`--undefined` *`symbol`*

Force *symbol* to be entered in the output file as an undefined symbol. Doing this may, for example, trigger linking of additional modules from standard libraries. `-u` can be repeated with different option arguments to enter additional undefined symbols. This option is equivalent to the `EXTERN` linker script command.

`--unique``[=`*`section`*`]`

Create a separate output section for every input section matching the specified section name. If a section name is not specified, create a separate output section for each orphan input section. An orphan section is one not specifically mentioned in a linker script.

This option can be used multiple times on the command line; It prevents the normal merging of input sections with the same name, overriding any output section assignments in a linker script.

`--unresolved-symbols=`*`method`*

Specify how linker should handle any unresolved symbols. *method* can be one of the following values:

`ignore-all`

Do not report any unresolved symbols.

`report-all`

Report all unresolved symbols (default).

`ignore-in-object-files`

Report unresolved symbols contained in shared libraries, but ignore any in regular object files.

`ignore-in-shared-libs`

Report unresolved symbols contained in regular object files, but ignore any in shared libraries. This is useful when creating a dynamic binary where all shared libraries it references are specified as arguments on the linker command line.

NOTE How shared libraries are linked can also be controlled with the `--allow-shlib-undefined` option.

How unresolved symbols are reported can be controlled with the `--warn-unresolved-symbols` option.

`-Ur`

For anything other than C++ programs, this option is equivalent to `-r`. It generates relocatable output (*i.e.,* an output file that can in turn serve as input to the linker. When linking C++ programs, `-Ur` *does* resolve references to constructors, unlike `-r`. It does not work to use `-Ur` on files that were themselves linked with `-Ur`; once the constructor table has been built, it cannot be added to. Use `-Ur` only for the last partial link, and `-r` for the others.

`-v`
`--version`
`-V`

Display the version number for `t`he linker. The `-V` option also lists the supported emulations. No input file is specified when these options are used.

`--verbose | --dll-verbose`
Display the version number for the linker, and list the linker emulations supported. Display which input files can and cannot be opened. Display the linker script if using the default linker script (Section 3.4). No input file is specified when this option is used.

`--version-script` *`file`*
Specify the name of a version script. This option is used when creating shared libraries to specify additional information about the version hierarchy for the library being created. See Section 3.3.12.

`--warn-alternate-em`
Warn if an object has an alternate ELF machine code.

`--warn-unresolved-symbols`
Generate warnings instead of errors for any unresolved symbols. See `--unresolved-symbols`.

`--warn-common`
Warn when a common symbol is combined with another common symbol or with a symbol definition. UNIX linkers allow this somewhat sloppy practice, but linkers on some other operating systems do not. This option allows you to find potential problems from combining global symbols. Unfortunately, some C libraries use this practice, so you may get warnings about symbols in the libraries as well as in your programs. There are three kinds of global symbols, illustrated here in C examples:

```
int i = 1;
```

A definition, which goes in the initialized data section of the output file.

```
extern int i;
```

An undefined reference, which does not allocate space. There must be either a definition or a common symbol for the variable somewhere.

```
int i;
```

A common symbol. If there are only (one or more) common symbols for a variable, it goes in the uninitialized data area of the output file. The linker merges multiple common symbols for the same variable into a single symbol. If they are of different sizes, it picks the largest size. The linker turns a common symbol into a declaration, if there is a definition of the same variable.

The `--warn-common` option can produce five kinds of warnings (listed below). Each warning consists of a pair of lines: the first describes the symbol just encountered, and the second describes the previous symbol encountered with the same name. One or both of the two symbols will be a common symbol.

- Turning a common symbol into a reference because there is already a definition for the symbol.

```
file(section): warning: common of 'symbol'
   overridden by definition
file(section): warning: defined here
```

- Turning a common symbol into a reference because a later definition for the symbol is encountered. This is the same as the previous case except that the symbols are encountered in a different order.

```
file(section): warning: definition of 'symbol'
   overriding common
file(section): warning: common is here
```

- Merging a common symbol with a previous same-sized common symbol.

```
file(section): warning: multiple common
   of 'symbol'
file(section): warning: previous common is here
```

- Merging a common symbol with a previous larger common symbol.

```
file(section): warning: common of 'symbol'
   overridden by larger common
file(section): warning: larger common is here
```

- Merging a common symbol with a previous smaller common symbol. This is the same as the previous case except that the symbols are encountered in a different order.

```
file(section): warning: common of 'symbol'
   overriding smaller common
file(section): warning: smaller common is here
```

`--warn-once`
Only warn once for each undefined symbol rather than once per module that refers to it.

`--warn-section-align`
Warn if the address of an output section is changed because of alignment. Typically, the alignment will be set by an input section. The address will only be changed if it is not explicitly specified; that is, if the `SECTIONS` command does not specify a start address for the section (see Section 3.3.9).

`--warn-shared-textrel`
Generate a warning if the linker adds a `DT_TEXTREL` tag to a shared object.

`--whole-archive`
For each archive mentioned on the command line after the `--whole-archive` option, include every object file in the archive in the link rather than searching the archive for the required object files. This is normally used to turn an archive file into a shared object, forcing every object to be included in the resulting shared object.

NOTE This option can be specified more than once, in conjunction with the `--no_whole-archive` option.

--wrap *symbol*

Use a wrapper function for *symbol*. Any undefined reference to *symbol* will be resolved to `__wrap_`*symbol*. Any undefined reference to `__real_`*symbol* will be resolved to *symbol*. This can be used to provide a wrapper for a system function. The wrapper function should be called `__wrap_`*symbol*. If it wants to call the system function, it should call `__real_`*symbol*. Here is a trivial example:

```
void *
__wrap_malloc (int c)
{
  printf ("malloc called with %the linker\n", c);
  return __real_malloc (c);
}
```

If you link other code with this file using `--wrap malloc`, then all calls to `malloc` will call the function `__wrap_malloc` instead. The call to `__real_malloc` in `__wrap_malloc` will call the real `malloc` function.

You may want to provide a `__real_malloc` function as well, so that links without the `--wrap` option will succeed. If you do this, you should not put the definition of `__real_malloc` in the same file as `__wrap_malloc`; if you do, the assembler may resolve the call before the linker has a chance to wrap it to `malloc`.

-x
--discard-all

Delete all local symbols.

-X
--discard-locals

Delete all temporary symbols (Section 2.5.2).

-y *symbol*
--trace-symbol *symbol*

Print the name of each linked file in which *symbol* appears. This option can be given any number of times. On many systems, it is necessary to prepend an underscore. This option is useful when you have an undefined symbol in your link but do not know where the reference is coming from.

-Y *path*

Add *path* to the default library search path. This option exists for Solaris compatibility.

-z *keyword*

Perform various operations related to shared libraries. A space must appear between the option character and the following keyword.

combreloc

Combine and sort multiple relocation sections. This enables dynamic symbol lookup caching.

common-page-size=*value*

Set emulation common page size to the specified value.

defs

Report unresolved symbol references from regular object files (even if the linker is creating a shared object). Identical to `-no-undefined` above.

`execstack`

Mark object as requiring executable stack.

`initfirst`

When building a shared library, specify that its runtime initialization occurs before the initialization of any other objects brought into the process at the same time, and its runtime finalization occurs after the finalization of any other objects.

`interpose`

Specify that the object's symbol table interposes all symbols but those of the primary executable.

`lazy`

When building a shared library, specify that function call binding is deferred until the functions are actually called (i.e., *lazy* binding), rather than binding at load time. This is the default setting.

`loadfltr`

Specify that the object's filters are processed immediately at runtime.

`max-page-size=`*`value`*

Set emulation maximum page size to the specified value.

`muldefs`

Allow multiple definitions.

`nocombreloc`

Disable the combining of multiple relocation sections.

`nocopyreloc`

Disable the production of copy relocation sections.

`nodefaultlib`

Specify that the object's dependency searches ignore default library search paths.

`nodelete`

Specify that object shouldn't be unloaded at runtime.

`nodlopen`

Specify that object is not available to `dlopen`.

`nodump`

Specify that object is not available to `dldump`.

`noexecstack`

Mark object as requiring executable stack.

`norelro`

Do not create `PT_GNU_RELRO` segment header in object.

`now`

When generating an executable or shared object, specify that the dynamic linker resolve all symbols when the program is started (or when the shared library is linked to using `dlopen`), instead of deferring symbol resolution to when the function is first called. Also known as *non-lazy runtime binding*.

`origin`

Specify that object requires immediate processing of `$ORIGIN`.

`relro`

Create `PT_GNU_RELRO` segment header in object.

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

Search the specified archives repeatedly until no new undefined references are created. The archives can be specified as explicit file names or as `-l` options.

Archives are normally searched only once, and in the order that their names appear on the command line. The ordering is crucial; if a symbol defined in one archive is needed to resolve an undefined symbol reference in another archive that appears later on the command line, then the linker is unable to resolve the symbol reference. Specifying the archives within this option causes the linker to search the archives repeatedly until all such symbol references are resolved.

Using this option has a significant performance cost. It is best used only when unavoidable circular references exist between two or more archives.

The parentheses form of this option may not work on some command shells because the parentheses are interpreted by the shell.

`@`*`file`*

Use command arguments stored in the specified file (Section 3.2.1).

NOTE No space can appear between the at sign (`@`) and the file name.

3.2.3 Environment variables

The linker behavior can be controlled with the following environment variables:

- `GNUTARGET`
- `LDEMULATION`
- `COLLECT_NO_DEMANGLE`

`GNUTARGET` determines the input-file object format if you do not use the `-b` option. Its value should be one of the BFD names for an input format (Section 3.5). If `GNUTARGET` is set to default, BFD attempts to discover the input format by examining binary input files. This method often succeeds, but it can also create potential ambiguities because no method exists for ensuring that the magic number used to specify object file formats is unique. However, the configuration procedure for BFD on each system places the conventional format for that system first in the search list, so ambiguities are resolved in favor of convention.

NOTE If `GNUTARGET` is not defined, the linker uses the natural format of the target.

`LDEMULATION` determines the default emulation if you do not use the `-m` option. The emulation can affect various aspects of linker behavior, particularly the default linker script. You can list the available emulations with the `--verbose` or `-V` options. If the `-m` option is not used and the `LDEMULATION` environment variable is not defined, the default emulation depends upon how the linker was configured.

`COLLECT_NO_DEMANGLE` causes the linker to not demangle symbols. Normally, the linker demangles symbols by default.

NOTE `COLLECT_NO_DEMANGLE` is used in a similar manner by the GCC linker wrapper. The default setting can be overridden by using the options `--demangle` and `--no-demangle`.

3.3 Linker scripts

The linker is controlled by a *linker script* which controls how the input files are linked. The script, which is written in the linker command language, controls the following link properties:

- How sections in the input file are mapped to the output file
- File format and memory layout of the output file
- Runtime load properties of the created segments
- Code execution entry points
- Shared library versions

The linker contains a built-in script which it uses as the *default linker script* for assigning code and data to memory. The default script cannot be modified by the user; however, it can be changed in two ways:

- It can be entirely replaced by a user-defined script file. In this case the file name is specified on the linker command line as the argument of the linker option `-T`.
- It can be augmented (rather than replaced) by specifying a user-defined script file on the linker command as a normal linker input file.

Scripts that are specified as normal linker input files are called *implicit scripts*. Because they augment the default linker script, implicit scripts typically contain only symbol assignments, or the `INPUT`, `GROUP`, or `VERSION` commands (Section 3.3.2).

The linker can determine whether an input file is an object file, an archive file, or an implicit script file. If the linker cannot recognize an input file as one of these file types, it generates an error.

NOTE The default linker script can be viewed by using the `--verbose` option.

Certain options (such as `-N` or `-r`) affect the default linker script.

The default linker script allocates memory sections according to the default values for the Hexagon processor memory layout (Section 3.4.2).

Any input files read by an implicit linker script are read at the position in the command line where the implicit linker script itself was read. This can affect archive searching.

3.3.1 Script example

Linker scripts can be simple or complex, depending on a program's link map and memory layout. For the purposes of explaining linker scripts, the example program presented here contains only the following three sections:

- `.text` – contains program code
- `.data` – contains program data
- `.bss` – contains uninitialized data

The program code should be loaded at memory address 0x20000, and the program data starting at address 0x4000000.

Given the above information, here is the complete linker script for the program:

```
SECTIONS
{
  . = 0x20000;
  .text : { *(.text) }
  . = 0x4000000;
  .data : { *(.data) }
  .bss : { *(.bss) }
}
```

The script consists of a single `SECTIONS` command (Section 3.3.9) which specifies how input sections are mapped to output sections, and how output sections are placed in memory. The `SECTIONS` command consists of a series of statements enclosed in curly brackets.

The first statement in the command sets the value of the location counter symbol (`.`) to the code starting address (`0x20000`). If an output section is not explicitly assigned a starting address (Section 3.3.9.11), it is implicitly assigned the current value of the location counter. The location counter is then incremented by the size of the output section. (The location counter is initialized to 0 at the start of the `SECTIONS` command.)

The second statement defines the example program code section by defining an output section named `.text`, and mapping into it all input sections of type `.text`. The expression `*(.text)` specifies all `.text` input sections in all linker input files. The output section address is set to the current location counter value (`0x20000`).

The remaining three statements define the two data sections (`.data` and `.bss`) in the example program. The `.data` section is assigned to address `0x4000000`, and the `.bss` section is assigned to memory immediately after the end of the `.data` section.

To ensure that each output section has the required address alignment, the linker may automatically increase the location counter at the start of a section. In the above example, the specified addresses for the `.text` and `.data` sections are likely to satisfy any alignment constraints, but the linker may need to create a small alignment gap at the end of the `.data` section in order to properly align the `.bss` section.

NOTE Linker scripts support many features that are not used in this example. For more information see the following document sections.

3.3.2 Script commands

Linker scripts are text files that contain one or more linker script commands.

Commands have the following formats:

- Keywords optionally followed by one or more arguments
- Symbol assignments of the form *symbol* = *expression*

Script commands are separated by semicolons, and whitespace is generally ignored. For example, the following script contains the `PHDRS` and `SECTIONS` commands:

```
PHDRS
{
headers PT_PHDR PHDRS ;
interp PT_INTERP ;
text PT_LOAD FILEHDR PHDRS ;
data PT_LOAD ;
dynamic PT_DYNAMIC ;
} ;

SECTIONS
{
  . = SIZEOF_HEADERS;
  .interp : { *(.interp) } :text :interp
  .text : { *(.text) } :text
  .rodata : { *(.rodata) }    /* defaults to :text */
    ...
  . = . + 0x1000;             /* move to new page in memory */
  .data : { *(.data) } :data
  .dynamic : { *(.dynamic) } :data :dynamic
  ...
}
```

The main linker script commands are `SECTIONS` and `MEMORY`.

`SECTIONS` specifies how input sections are mapped to output sections, and how output sections are placed in memory.

`MEMORY` describes the available memory in the target architecture. If this command is not specified in a script, the linker assumes that sufficient memory is available in a contiguous block for all output.

NOTE Every linker script must contain a `SECTIONS` command – no other script command is required to appear in every script.

Table 3-1 lists the linker script commands.

Table 3-1 Linker script commands

Command	Description
SECTIONS	Link map and section memory placement
MEMORY	Target memory configuration
ENTRY	Code entry point
REGION_ALIAS	Create alias for memory region name
symbol = *expression*; (=, +=, -=, *=, /=, <<=, >>=, *=, |=)	Define symbol with specified value
PROVIDE	Define symbol if referenced but not defined
PROVIDE_HIDDEN	Define hidden PROVIDE symbol
EXTERN	Specify symbol as undefined
PHRDS	Segment load information
VERSION	Shared library version number
INPUT	Specify file as linker input file
STARTUP	Specify file as *first* linker input file
GROUP	Specify file as linker archive file
OUTPUT	Specify output file name
OUTPUT_FORMAT	Specify output file format
OUTPUT_ARCH	Specify output machine architecture
TARGET	Specify input file format
FORCE_COMMON_ALLOCATION	Assign space to common symbols (even for relocatable output files)
INHIBIT_COMMON_ALLOCATION	Inhibit assigning space to common symbols (even relocatable output files)
AS_NEEDED	Add shared libraries only when needed
INSERT	Insert prior script commands before/after the specified section
NOCROSSREFS	Generate error if any cross-references exist between the specified sections
ASSERT	Exit linker with error code if the value of the specified expression is zero.
INCLUDE	Include linker script file
SEARCH_DIR	Add specified path to search list

3.3.3 Semicolons

In linker scripts semicolons (`;`) are generally used as delimiters for aesthetic reasons only, and are otherwise ignored. However, they are required in the following places:

- Semicolons must appear at the end of symbol assignments (Section 3.3.7)
- Semicolons must appear at the end of `PHDRS` commands (Section 3.3.11)

3.3.4 Comments

Comments can be included in linker scripts using the standard C delimiters:

```
/* and */
```

As in C, comments are syntactically equivalent to whitespace.

3.3.5 Strings

Character strings such as file or format names can normally be entered directly without the need for delimiters

If a file name contains a character such as a comma which would otherwise serve to separate file names, the file name can be enclosed in double quotes. Double quote characters cannot be used in file names.

3.3.6 Expressions

Many command arguments accept arithmetic expressions. The syntax for expressions is identical to the expression syntax in C, with the following features:

- All expressions are evaluated as integers of type `long` or `unsigned long`
- All constants are integers
- All C arithmetic operators are provided
- Global variables can be defined, created, and referenced
- Several predefined functions are supported

3.3.6.1 Integers

An octal integer is `0` followed by zero or more of the octal digits (`01234567`).

```
_as_octal = 0157255;
```

A decimal integer starts with a non-zero digit followed by zero or more digits (`0123456789`).

```
_as_decimal = 57005;
```

A hexadecimal integer is `0x` or `0X` followed by one or more hexadecimal digits chosen from `0123456789abcdefABCDEF`.

```
_as_hex = 0xdead;
```

To write a negative integer, use the prefix operator (`-`)`;` see Section 3.3.6.4.

```
_as_neg = -57005;
```

Additionally, the suffixes `K` and `M` can be used to scale a constant by 1024 or (1024*1024) respectively. For example, the following all refer to the same quantity:

```
_fourk_1 = 4K;
_fourk_2 = 4096;
_fourk_3 = 0x1000;
```

3.3.6.2 Symbol names

Unless quoted, symbol names start with a letter, underscore, or point and can include any letters, underscores, digits, points, and hyphens. Unquoted symbol names must not conflict with any keywords. You can specify a symbol which contains odd characters or has the same name as a keyword, by surrounding the symbol name in double quotes:

```
"SECTION" = 9;
"with a space" = "also with a space" + 10;
```

Since symbols can contain many non-alphabetic characters, it is safest to delimit symbols with spaces. For example, `A-B` is one symbol, whereas `A - B` is an expression involving subtraction.

3.3.6.3 Location counter

The special linker variable *dot* (`.`) always contains the current output location counter. Since the `.` always refers to a location in an output section, it must always appear in an expression within a `SECTIONS` command. The `.` symbol can appear anywhere that an ordinary symbol is allowed in an expression, but its assignments have a side effect. Assigning a value to the `.` symbol will cause the location counter to be moved. This can be used to create holes in the output section. The location counter can never be moved backwards.

```
SECTIONS {
  output :
  {
    file1(.text)
    . = . + 1000;
    file2(.text)
    . += 1000;
    file3(.text)
  } = 0x1234;
}
```

In the previous example, `file1` is located at the beginning of the output section, then there is a 1000-byte gap. Then `file2` appears, also with a 1000-byte gap following before `file3` is loaded. The notation "`= 0x1234`" specifies what data to write in the gaps (see Section 3.3.9.11).

3.3.6.4 Operators

The linker recognizes the standard C set of arithmetic operators, with the standard bindings and precedence levels (see Section 3.3.6).

3.3.6.5 Evaluation

The linker uses *lazy evaluation* for expressions: it calculates an expression only when absolutely necessary. The linker needs the value of the start address and the lengths of memory regions to do any linking at all. These values are computed as soon as possible when the linker reads in the command file. However, other values (such as symbol values) are not known or needed until after storage allocation. Such values are evaluated later, when other information (such as the sizes of output sections) is available for use in the symbol assignment expression.

3.3.6.6 Predefined functions

The command language includes a number of predefined functions for use in link script expressions.

`ABSOLUTE(`*`exp`***`)`**

Return the absolute (non-relocatable, as opposed to non-negative) value of the expression *exp*. Primarily useful to assign an absolute value to a symbol within a section definition, where symbol values are normally section-relative.

`ADDR(`*`section`***`)`**

Return the absolute address (VMA) of the named *section*. Your script must previously have defined the location of that section. In the following example, `symbol_1` and `symbol_2` are assigned identical values:

```
SECTIONS {
  ...
  .output1 :
  {
    start_of_output_1 = ABSOLUTE(.);
    ...
  }
  .output :
  {
    symbol_1 = ADDR(.output1);
    symbol_2 = start_of_output_1;
  }
  ...
}
```

`ALIGN(`*`align`***`)`**
`ALIGN(`*`exp, align`***`)`**

Return either the current location counter (`.`) or the specified expression aligned to the next *align* boundary. *align* must be an expression whose value is a power of two. This is equivalent to:

```
(. + align - 1) & ~(align - 1)
```

`ALIGN` does not change the value of the location counter – it merely performs arithmetic on it. As an example, to align the output `.data` section to the next `0x2000` byte boundary after the preceding section and to set a variable within the section to the next `0x8000` boundary after the input sections:

```
SECTIONS {
  ...
  .data ALIGN(0x2000): {
    *(.data)
    variable = ALIGN(0x8000);
  }
  ...
}
```

The first use of `ALIGN` in this example specifies the location of a section because it is used as the optional *address* attribute of a section definition (Section 3.3.9.11). The second use simply defines the value of a variable. The predefined function `NEXT` is closely related to `ALIGN`.

The two-argument variation allows an expression to be aligned upwards to the specified alignment value. `ALIGN(.,`*`align`*`)` is equivalent to `ALIGN(`*`align`*`)`.

`ALIGNOF(`*`section`*`)`

Return the alignment (in bytes) of the specified allocated section. If the section is not allocated the linker generates an error. In the following example the section alignment is stored as the first value in the section.

```
SECTIONS{ ...
  .output {
    LONG (ALIGNOF (.output))
    ...
    }
... }
```

`DATA_SEGMENT_ALIGN(`*`maxpagesize`*`, `*`commonpagesize`*`)`

Evaluate the following two expressions:

```
(ALIGN(maxpagesize) + (. & (maxpagesize - 1)))
(ALIGN(maxpagesize) + (. & (maxpagesize - commonpagesize)))
```

Return the second expression if it uses fewer `commonpagesize`-sized pages for the data segment than the first expression; otherwise, return the first expression.

data segment is defined as the difference between the expression result and the value returned by `DATA_SEGMENT_END`.

If the second expression is returned, it indicates that the alignment value will save `commonpagesize` bytes of runtime memory, at the expense of wasting up to `commonpagesize` bytes in secondary storage.

`DATA_SEGMENT_ALIGN` has the following restrictions: it must be used directly in the `SECTIONS` command; it cannot be used in any output section descriptions; and it can be used only once in a linker script.

`commonpagesize` must be less than or equal to `maxpagesize`, and must also be the system page size that the object is to be optimized for (while still working on system page sizes of up to `maxpagesize`).

Example:

```
. = DATA_SEGMENT_ALIGN(0x10000, 0x2000);
```

`DATA_SEGMENT_END(`*`exp`*`)`

Return the end of a data segment (for use in evaluating `DATA_SEGMENT_ALIGN`):

```
. = DATA_SEGMENT_END(.);
```

`DATA_SEGMENT_RELRO_END(``offset, exp`**`)`**

Return the second argument.

In addition, if the option “`-z relro`” is specified, pad `DATA_SEGMENT_ALIGN` so the expression `(exp+offset)` is aligned to the most commonly-used page boundary for the target platform.

If “`-z relro`” is *not* specified, do nothing.

This function must always appear between `DATA_SEGMENT_ALIGN` and `DATA_SEGMENT_END`.

```
. = DATA_SEGMENT_RELRO_END(24, .);
```

`DEFINED(``symbol`**`)`**

Return `1` if *symbol* is both in the linker global symbol table and defined prior to the statement containing this function; otherwise, return `0`. You can use this function to provide default values for symbols. For example, the following command-file fragment shows how to set a global symbol `begin` to the first location in the `.text` section, but if a symbol called `begin` already existed, its value is preserved:

```
SECTIONS { ...
  .text : {
    begin = DEFINED(begin) ? begin : . ;
    ...
  }
  ...
}
```

`LENGTH(``memory`**`)`**

Return length of the specified memory region.

`LOADADDR(``section`**`)`**

Return absolute load address (LMA) of the named *section*. This is normally the same as `ADDR`, but it can be different if the `AT` attribute is specified in the section definition (Section 3.3.9.11).

`MAX(``exp1, exp2`**`)`**

Return the maximum of *exp1* and *exp2*.

`MIN(``exp1, exp2`**`)`**

Return the minimum of *exp1* and *exp2*.

`NEXT(``exp`**`)`**

Return the next unallocated address that is a multiple of *exp*. This function is closely related to `ALIGN(`*exp*`)`; unless you use the `MEMORY` command to define discontinuous memory for the output file, the two functions are equivalent.

`ORIGIN(``memory`**`)`**

Return the origin of the specified memory region.

`SEGMENT_START(``segment, default`**`)`**

Return the base address of the specified segment. If a value has been explicitly specified for this segment (with the `-T` option) return that value; otherwise, return the default value.

SIZEOF(*section*)

Return the size in bytes of the named *section*, if that section has been allocated. In the following example, `symbol_1` and `symbol_2` are assigned identical values:

```
SECTIONS { ...
  .output
  {
    .start = . ;
    ...
    .end = . ;
  }
  symbol_1 = .end - .start ;
  symbol_2 = SIZEOF(.output);
  ...
}
```

SIZEOF_HEADERS
sizeof_headers

Return the size in bytes of the output file headers. You can use this number as the start address of the first section to facilitate paging.

If a linker script uses this function, the linker must compute the number of program headers before it has determined all the section sizes and addresses. If the linker subsequently discovers that it requires additional program headers, it will generate the error “Not enough room for program headers”. To avoid this do one of the following: a) avoid using this function; b) revise the linker script to avoid forcing the linker to use additional program headers; or c) explicitly define the program headers using `PHDRS` (Section 3.3.11).

3.3.7 Symbol assignment

Symbols can be defined in linker scripts, and can be assigned values using any of the C assignment operators:

```
symbol = expression ;
symbol &= expression ;
symbol += expression ;
symbol -= expression ;
symbol *= expression ;
symbol /= expression ;
```

Symbols are defined to have global scope within the script.

Symbol assignment statements differ in two ways from the operators used in linker script expressions:

- Assignment can be performed only at the root of an expression. For example, `a=b+3;` is allowed, but `a+b=3;` is an error.
- Assignment statements must be terminated with a trailing semicolon (`;`)

Assignment statements can appear in the following locations:

- As independent commands in a linker script
- As independent statements within a `SECTIONS` command
- As part of the contents of a section definition in a `SECTIONS` command

The first two are equivalent in effect: both define a symbol with an absolute address. The last defines a symbol whose address is relative to a particular section (Section 3.3.9).

When a linker expression is evaluated and assigned to a variable, it is given either an absolute or a relocatable type. An absolute expression type is one in which the symbol contains the value that it will have in the output file. A relocatable expression type is one in which the value is expressed as a fixed offset from the base of a section.

The type of an expression is controlled by its position in the script file. A symbol assigned within a section definition is created relative to the base of the section; a symbol assigned in any other place is created as an absolute symbol. Because a symbol created in a section definition is relative to the base of the section, it remains relocatable if relocatable output is requested. A symbol can be created with an absolute value even when assigned in a section definition by using the absolute assignment function `ABSOLUTE`. For example, the following command creates an absolute symbol whose address is the last byte of an output section named `.data`:

```
SECTIONS {
  ...
  .data :
  {
    *(.data)
    _edata = ABSOLUTE(.) ;
  }
  ...
}
```

The linker attempts to postpone evaluating an assignment until all terms in the source expression are known (see Section 3.3.6.5). For instance, the size of a section cannot be known until after allocation, so assignments dependent upon these properties are not performed until after allocation.

Some expressions (such as those depending upon the location counter *dot)* must be evaluated during allocation. If the result of an expression is required, but the value is not available, an error message is generated. For example:

```
SECTIONS {
  ...
  text 9+this_isnt_constant :
  {
    ...
  }
  ...
}
```

This command generates the error message "`Non constant expression for initial address`".

3.3.7.1 PROVIDE

The `PROVIDE` command is used to conditionally define a symbol in a linker script. The command defines the specified symbol only if the symbol is referenced but not defined by any object specified in the link.

The command has the following syntax:

```
PROVIDE(symbol = expression)
```

3.3.7.2 PROVIDE_HIDDEN

The `PROVIDE_HIDDEN` command is equivalent to `PROVIDE` (Section 3.3.7.1), but the resulting defined symbol is hidden and not exported.

3.3.7.3 Source code access

Linker script symbols can be accessed from program source code; however, they have no accessible value in source code, and can be referenced there only as addresses (using the C operator `'&'`).

3.3.8 MEMORY command

The linker's default configuration permits allocation of all available memory. You can override this configuration by using the `MEMORY` command.

The `MEMORY` command describes the location and size of blocks of memory in the target. By using it carefully, you can describe which memory regions can be used by the linker, and which memory regions it must avoid. The linker does not shuffle sections to fit into the available regions, but does move the requested sections into the correct regions, and generates errors when the regions become too full.

A linker script can contain at most one occurrence of the `MEMORY` command. However, you can define as many blocks of memory within it as you want. The command syntax is:

```
MEMORY
  {
    name (attr) : ORIGIN = origin, LENGTH = len
    ...
  }
```

The `MEMORY` syntax elements are described below.

`name`

A name used internally by the linker to refer to the region. Any symbol name can be used – region names have a separate name space and therefore do not conflict with the names of symbols, files, or sections. Within the `MEMORY` command each memory region name must be distinct. However, you can create aliases to existing memory region names using the `REGION_ALIAS` command (Section 3.3.13).

`attr`

An optional list of attributes which specify whether to use a particular memory to place sections that are not listed in the linker script. Valid attribute lists must be made up of the characters `RWXAIL!` that match section attributes.

NOTE If the attribute list is entirely omitted, the parentheses can be omitted too.

The attribute list can include the following characters:

Letter	Section Attribute
R	Read-only sections.
W	Read/write sections.
X	Sections containing executable code.
A	Allocated sections.
I	Initialized sections.
L	Same as `I`.
!	Invert the sense of any of the following attributes.

`origin`

The start address of the region in physical memory. It is an expression that must evaluate to a constant before memory allocation is performed. The keyword `ORIGIN` can be abbreviated to `org` or `o` (but not, for example, `ORG`).

`len`

The size in bytes of the region (an expression). The keyword `LENGTH` can be abbreviated to `len` or `l`.

For example, consider a memory configuration that has two regions available for allocation: one starting at 0 for 256 kilobytes and the other starting at `0x40000000` for four megabytes. The `rom` memory region will get all sections that are either read-only or contain executable code, while the `ram` memory region will get the remaining sections.

```
MEMORY
  {
  rom (rx)  : ORIGIN = 0, LENGTH = 256K
  ram (!rx) : org = 0x40000000, l = 4M
  }
```

Once you have defined a region of memory named *mem*, you can direct specific output sections there by using a command ending in `>`*`mem`* within the `SECTIONS` command (see Section 3.3.9). If an output section has no specified address, the linker sets the address to the next available address in the memory region. If the combined output sections directed to a region are too big for the region, the linker issues an error message.

NOTE The MEMORY command is not used to specify processor-specific memory layouts. This is done using the processor-specific memory features; for details see Section 3.4.

The origin and length of a memory can be specified in linker script expressions using the predefined functions `ORIGIN` and `LENGTH` (Section 3.3.6.6).

3.3.9 SECTIONS command

The SECTIONS command is used to specify how input sections are mapped to output sections, and how output sections are placed in memory.

The command has the following high-level syntax:

```
SECTIONS
{
  sections-statement
  sections-statement
   ...
}
```

Only one `SECTIONS` command can be declared in a script file; however, the command can contain an arbitrary number of statements for specifying the necessary mapping and placement information.

A `SECTIONS` command can contain the following section statements:

- Output section descriptions (Section 3.3.9.1)
- An `ENTRY` command (Section 3.3.10)
- Symbol assignments (Section 3.3.7)
- Overlay descriptions (Section 3.3.9.12)

The `ENTRY` command and symbol assignments can be declared in a linker script either inside or outside the `SECTIONS` command. They are permitted inside `SECTIONS` for convenience in using the location counter in those commands. Doing this makes the linker script more readable by having the commands appear at meaningful locations in the layout of the output file.

If you do not use a `SECTIONS` command, the linker places each input section in an identically-named output section in the order that the sections are encountered in the input files. If, for example, all input sections are present in the first file, then the section order in the output file will match the section order in the first input file. The first section is assigned address zero.

NOTE The linker command line option `-M` can be used to verify how input sections are being mapped to output sections. This option generates a map file showing the precise mapping.

3.3.9.1 Output section descriptions

The most frequently used statement in the `SECTIONS` command is the *output section description*, which specifies the properties of an output section: its location, alignment, contents, fill pattern, and target memory region.

Output section descriptions have the following high-level syntax:

```
output-section-name :
   {
    output-section-statement
    output-section-statement
    ...
   }
```

An output section name can consist of any sequence of characters. (However, the name must be quoted if it does not conform to the standard syntax for linker symbol names.)

Whitespace is required around the section name declaration to ensure that the name is unambiguous. Line breaks and other white space characters are optional.

For example, the following output section description consists of only a section name (`.text`) and a single input section description:

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

An output section description consists of one or more statements, where each statement can be one of the following:

- A symbol assignment (Section 3.3.7)
- An input section description (Section 3.3.9.2)

Input section descriptions are used to specify section mappings (Section 3.3.9.2), output section data (Section 3.3.9.7), and section fill values (Section 3.3.9.8).

NOTE Output section descriptions can optionally specify a number of attributes. For more information see Section 3.3.9.11.

The output section name `/DISCARD/` is reserved – for more information see Section 3.3.9.10.

3.3.9.2 Input section descriptions

Input section descriptions specify the input sections that are mapped into output sections. They have the following syntax:

```
file
file( section )
file( section , section, ... )
file( section section ... )
```

An input section description consists of a file name optionally followed by a list of section names in parentheses. For example:

```
data.o(.data)
```

The file name specifies the input file that containing the input section (`data.o`), while the section name specifies the input section itself (`.data`). An input section description can specify specific input files, specific input sections, or a combination of the two.

NOTE Specifying just a file name as an input section description indicates that *all* sections in the specified file should be included in the output section.

The most common input section description specifies all the input sections that share a particular name. This is done by using wildcard characters (Section 3.3.9.4) in the file name. For example:

```
*(.text)
```

This example specifies the `.text` sections that are contained in all the linker input files.

A single input section description can explicitly specify more than one section name. For example:

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

This specifies the `.text` and `.rdata` sections contained in all the linker input files. Note that this file specification causes the specified sections to appear in the output section in the same order that they are matched in the linker input files. To create an output data section that contains all the `.text` sections followed by all the `.rdata` sections, put two separate input section descriptions in the output section description:

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

NOTE If an input file has already been specified in a previous input section description, then the subsequent specification is constrained to specifying only those sections that have not yet been assigned to an output section.

If a specified input file was not included on either the command line or in an `INPUT` command (Section 3.3.13), the linker attempts to open the file as an input file, as though it were specified on the command line.

3.3.9.3 Input section archive files

Input section descriptions can specify input sections stored in archive files. This is done using the following syntax:

```
archive::file
archive::
:file
```

Specifying both an archive and file name specifies a particular file in an archive:

```
my_arch:my_file
```

Specifying just an archive name (suffixed with "`:`") specifies *all* files in an archive:

```
my_arch:
```

Specifying just a file name (prefixed with "`:`") specifies a file that is not in any archive:

```
:my_file
```

Both the archive and file names can include wildcard characters (Section 3.3.9.4).

NOTE On Windows platforms, `c:my_file` is interpreted as a drive letter and file name rather than as an archive file specification.

Archive file specifications can also be used with `EXCLUDE_FILE` (Section 3.3.9.4).

3.3.9.4 Input section wildcards

Wildcard characters can be used in input section descriptions (Section 3.3.9.2) to specify the names of archives, files, and sections.

- An asterisk (`*`) matches any number of characters.
- A question mark (`?`) matches any single character.
- The sequence `[`*`chars`*`]` matches a single instance of any of the *`chars`*]
- A hyphen (`-`) can be used to specify character ranges in a bracketed character sequence.
- A backslash (`\`) can be used to quote the character immediately following it.

When a file name is matched with a wildcard, the wildcard characters do not match a slash (`/`) because it is used to separate directory names on UNIX. A pattern consisting of a single asterisk (`*`) is an exception: it always matches any file name. In a section name, the wildcard characters match a slash (`/`).

NOTE Wildcard patterns only match files that are explicitly specified on the command line. The linker does not search directories to expand wildcards.

Wildcard exclusions

The keyword `EXCLUDE_FILE` can be used to exclude certain files from matching a file name wildcard. The list of files to exclude is included in parentheses following the keyword. For example:

```
*(EXCLUDE_FILE (*crtend.o *otherfile.o) .ctors)
```

This example specifies the `.ctors` sections from all files *(i.e,* `*(.ctors)`) except for the two files `crtend.o` and `otherfile.o` (which are specifically excluded).

Duplicate matches

If a file name matches more than one wildcard pattern, or if a file name appears explicitly and is also matched by a wildcard pattern, the linker uses the first match in the linker script. For example, the following sequence of input section descriptions is most likely invalid, because the `data.o` rule will not be used:

```
.data : { *(.data) }
.data1 : { data.o(.data) }
```

Sorting matches

Normally the linker places files and sections matched by wildcards in the order they are encountered during the link. This order can be changed by prefixing a wildcard specification with the keyword `SORT_BY_NAME` or `SORT_BY_ALIGNMENT`. For example:

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

When `SORT_BY_NAME` is specified in an input section description, the linker sorts the files or sections by name (in ascending order) before placing them in the output file.

`SORT_BY_ALIGNMENT` is similar to `SORT_BY_NAME`, but sorts sections by alignment instead of by name.

NOTE `SORT` is an alias for `SORT_BY_NAME`.

Nested sorting

The section sorting commands can be nested one level deep in a wildcard specification. For example:

```
SORT_BY_NAME (SORT_BY_ALIGNMENT (wildcard))
```

Sort input sections first by name first, and then (if two sections have the same name) by alignment.

```
SORT_BY_ALIGNMENT (SORT_BY_NAME (wildcard))
```

Sort input sections first by alignment first, and then (if two sections have the same alignment) by name.

NOTE The other sorting command combinations are redundant or invalid.

Command-line sorting

The linker includes the command line option `--sort-sections` for specifying the section sort order. When this option is used with the linker script sorting commands, the sorting commands always take precedence over the command line option. Thus:

- `SORT_BY_NAME (w)` with `--sort-sections alignment` is equivalent to `SORT_BY_NAME (SORT_BY_ALIGNMENT (w))`.
- `SORT_BY_ALIGNMENT (w)` with `--sort-section name` is equivalent to `SORT_BY_ALIGNMENT (SORT_BY_NAME (w))`.

NOTE The command line option is ignored if the linker script contains nested section sorting commands.

Wildcard example

This example command shows how wildcard patterns can be used to partition files in an output section:

```
SECTIONS {
  .text : { *(.text) }
  .DATA : { [A-Z]*(.data) }
  .data : { *(.data) }
  .bss :  { *(.bss) }
}
```

It specifies the following mappings:

- All `.text` sections should be placed in `.text`
- All `.bss` sections should be placed in `.bss`
- All `.data` sections stored in files whose names begin with an uppercase alphabetic character (i.e., A-Z) should be placed in `.DATA`
- All `.data` sections stored in all other files should be placed in `.data`.

3.3.9.5 Input section common symbols

Input section descriptions can specify the common symbols in an input file by using the keyword `COMMON` as the name of an input section. This makes it possible to independently place the common symbols in a program.

Common symbols in input files are conventionally mapped to the `.bss` section in the output file. For example:

```
.bss { *(.bss) *(COMMON) }
```

3.3.9.6 Input section garbage collection

The keyword `KEEP` can be used to protect specific sections from garbage collection when using the linker command line option `--gc-sections`. For example:

```
KEEP(*(.init))
KEEP(SORT_BY_NAME(*)(.ctors))
```

3.3.9.7 Output section data

Data values can be placed in an output section with the input section description commands `BYTE`, `SHORT`, `LONG`, `QUAD`, and `SQUAD`. They have the following syntax:

```
BYTE(expression)
SHORT(expression)
LONG(expression)
QUAD(expression)
SQUAD(expression)
```

These commands store the value of the specified expression at the current location counter. After storing the value, the location counter is incremented by the number of bytes stored.

`QUAD` values are zero-extended to 64 bits, while `SQUAD` values are sign-extended to 64 bits.

Note that the data value commands can only be used in input section descriptions (Section 3.3.9.2) – they cannot be used directly as output section statements. For example, the following command is valid:

```
SECTIONS { .text : { *(.text) ; LONG(1) } .data : { *(.data) } }
```

But this command will generate a linker error:

```
SECTIONS { .text : { *(.text) } LONG(1) .data : { *(.data) } }
```

3.3.9.8 Output section fill

The input section description command `FILL` assigns a uniform fill value to the unspecified memory regions of an output section. It has the following syntax:

```
FILL(expression)
```

`FILL` affects only the part of the section following the command; therefore, it can be used more than once to assign different fill patterns to different parts of an output section.

`FILL` must appear in an input section description (Section 3.3.9.2) – it cannot be used as an output section statement.

NOTE `FILL` is similar to the output section fill attribute (Section 3.3.9.11), but unlike the attribute does not affect the entire section. If both `FILL` and the attribute are used, `FILL` takes precedence.

3.3.9.9 Output section constructors

To support C++ constructors and destructors, GNU C++ requires certain items to be defined in an object file:

- The symbols `__CTOR_LIST__` and `__CTOR_END__` mark the respective start and end of the list of global constructors.
- Similarly, `__DTOR_LIST__` and `__DTOR_END__` mark the respective start and end of the list of global destructors.

Each list consists of an initial word containing the number of entries, a series of entry addresses, and a terminating zero word. The compiler is responsible for generating code which executes the entries in the lists: GNU C++ normally calls constructors from a subroutine __main; a call to __main is automatically inserted into the startup code for main. GNU C++ normally runs destructors either by using `atexit`, or directly from the function exit.

GNU C++ normally arranges to put the addresses of global constructors and destructors into sections named `.ctors` and `.dtors`. The linker script can be used to build the sort of table that the GNU C++ runtime code expects to see. For example:

```
__CTOR_LIST__ = .;
LONG((__CTOR_END__ - __CTOR_LIST__) / 4 - 2)
*(.ctors)
LONG(0)
__CTOR_END__ = .;
__DTOR_LIST__ = .;
LONG((__DTOR_END__ - __DTOR_LIST__) / 4 - 2)
*(.dtors)
LONG(0)
__DTOR_END__ = .;
```

NOTE Normally the compiler and linker handle these issues automatically. However, you may need to consider this when using C++ with your own linker scripts.

3.3.9.10 Output section discarding

The linker does not create output sections with no contents. This is for convenience when referring to input sections that may or may not exist. For example:

```
.foo { *(.foo) }
```

creates a `.foo` section in the output file only if a `.foo` section exists in at least one input file. Other linker script commands that allocate space in an output section will also create the output section.

Address assignments are ignored (Section 3.3.9.11) on discarded output sections, except when the script defines symbols in the output section. In that case the linker obeys the assignments, possibly advancing the location counter even though the section is discarded.

The output section name `/DISCARD/` can be used to discard input sections. Any sections assigned to an output section named `/DISCARD/` are not included in the final link output.

3.3.9.11 Output section attributes

Output section descriptions (Section 3.3.9.1) can optionally specify a number of section attributes. Here is the full syntax:

```
output-section-name [address] [(type)] :
  [AT(lma)]
  [ALIGN(section_align)]
  [SUBALIGN(subsection_align)]
  [constraint]
  {
    output-section-statement
    output-section-statement
    ...
  } [>region] [AT>lma_region] [:phdr :phdr ...] [=fill]
```

output-section-name

See Section 3.3.9.1

address

Specify the VMA (virtual memory address) of an output section. If no address is specified, the linker sets it based on the *region* attribute (if present), or (if not) on the current value of the location counter.

If an address is specified, the output section address is set to the specified value. If neither an address nor a region is specified, the address is set to the current value of the location counter after it has been aligned to the output section's alignment requirements (which is defined as the strictest alignment of any input section contained in the output section).

For example, consider the differences between the following section descriptions:

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

The first description sets the address of the `.text` output section to the current value of the location counter. The second sets the address to the current value of the location counter aligned to the strictest alignment of a `.text` input section.

The address value can be specified as an expression (Section 3.3.6). For example:

```
.text ALIGN(0x10) : { *(.text) }
```

This statement aligns a section on a 0x10 byte boundary (i.e., the lowest four bits of the section address are zero). It works because `ALIGN` returns the current location counter aligned upward to the specified value.

Specifying an address for a section changes the value of the location counter, provided that the section is non-empty. (Empty sections are ignored.)

`(`*`type`*`)`

Specify the output section type. *type* can have the following values:

`NOLOAD`

Mark the section so it is not loaded into memory when the program is run.

`DSECT`
`COPY`
`INFO`
`OVERLAY`

Mark the section so no memory is allocated for it when the program is run. (Note that these type values are rarely used.)

By default the linker sets the attribute of an output section based on the input sections mapped into it.

`AT (`*`lma`***`)`**
`AT>`*`lma_region`*

Specify the LMA (virtual memory address) of an output section, either as an address or as a memory region:

- *lma* is an expression that specifies the section load address.
- Alternatively, *lma_region* can be used to specify a memory region for the section's load address (Section 3.3.8).

If neither an address nor a region is specified as a section attribute, the linker by default sets the LMA so that the difference between the VMA and LMA for the section is the same as that for the preceding output section in the same region.

If no preceding output section exists, or the section is not allocatable, the linker sets the LMA equal to the VMA.

These attributes are intended to simplify the building of a ROM image. For example:

```
SECTIONS
  {
  .text 0x1000 : { *(.text) _etext = . ; }
  .mdata 0x2000 :
    AT ( ADDR(.text) + SIZEOF ( .text ) )
    { _data = . ; *(.data); _edata = . ;  }
  .bss 0x3000 :
    { _bstart = . ;  *(.bss) *(COMMON) ; _bend = . ;}
}
```

This command creates two output sections: one named `.text` (which starts at 0x1000) and another named `.mdata` (which is placed at the end of `.text` even though its relocation address is 0x2000). The symbol `_data` is defined with the value 0x2000.

The run-time initialization code (for C programs, usually `crt0`) for use with a ROM generated this way would include code similar to the following to copy the initialized data from the ROM image to its runtime address. Note that the code references symbols defined in the linker script (Section 3.3.7.3).

```
char *src = &_etext;
char *dst = &_data;

/* ROM has data at end of text; copy it. */
while (dst < &_edata) {
  *dst++ = *src++;
}

/* Zero bss */
for (dst = &_bstart; dst< &_bend; dst++)
  *dst = 0;
```

NOTE When the `AT` attribute is used in a linker script, placing sections so they appear *after* any debug sections may yield unexpected results.

`ALIGN(`*`section_align`***`)`**
Align 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_align`***`)`**
Align 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`
Specify the output section constraint. *constraint* can have the following 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`*
Assign the section to the specified memory region (Section 3.3.8).

`:`*`phdr`*
Assign the section to a previously-defined program segment (Section 3.3.11). 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.

To prevent a section from being assigned to a segment when it would normally default to one, specify the attribute as `:NONE`.

`=fill`

Assign the value of the specified expression to all unspecified memory areas of the section.

Any unallocated areas in the current output section are filled with the two least significant bytes of the value, with the fill value repeated as necessary.

If the expression specifies a hexadecimal value (Section 3.3.6.1), then an arbitrarily long sequence of hex digits can be used to specify the fill pattern, with any leading zeros being treated as part of the fill pattern. For all other cases, the fill pattern is defined as the four least significant bytes of the expression value.

NOTE The fill pattern is treated as big-endian.

The fill attribute is similar to the FILL command (Section 3.3.9.8), but unlike the command it affects the entire section. If both the command and the attribute are used, the command takes precedence.

3.3.9.12 Overlay descriptions

An overlay description provides a simple way to describe sections which are loaded as part of a single memory image, but are run (at different times) at the same memory address. At runtime, an overlay manager is responsible for copying the overlaid sections into and out of memory as required.

Overlays are described in a linker script using the `OVERLAY` command. The command is declared in a `SECTIONS` command like an output section description (Section 3.3.9.1). The command syntax is as follows:

```
OVERLAY [start] : [NOCROSSREFS] [AT ( ldaddr )]
  {
    secname1
      {
        output-section-command
        output-section-command
        ...
      } [:phdr...] [=fill]
    secname2
      {
        output-section-command
        output-section-command
        ...
     } [:phdr...] [=fill]
   ...
 } [>region] [:phdr...] [=fill]
```

Everything is optional except the `OVERLAY` keyword and the section names (shown above as *secname1* and *secname2*).

Section definitions within an `OVERLAY` command are identical to those defined in the `SECTIONS` command, except that no addresses or memory regions can be defined for overlay sections.

The sections are all defined with the same starting address. The load addresses of the sections are arranged such that they are consecutive in memory starting at the load address used for the `OVERLAY` as a whole (as with normal section definitions, the load address is optional, and defaults to the start address; the start address is also optional, and defaults to the current value of the location counter).

If the `NOCROSSREFS` keyword is specified (Section 3.3.13), and any references exist among the sections, the linker generates an error. Since the sections all run at the same address, it normally does not make sense for one section to refer directly to another.

For each section within the `OVERLAY`, the linker automatically provides two symbols:

- `__load_start_`*`secname`* specifies the starting load address of the section.
- `__load_stop_`*`secname`* specifies the final load address of the section.

C (or assembler) code can use these symbols to move the overlaid sections around as necessary.

NOTE The linker automatically removes any characters in *`secname`* that are not valid in C identifiers.

At the end of the overlay, the value of the location counter is set to the start address of the overlay plus the size of the largest section.

For example:

```
OVERLAY 0x1000 : AT (0x4000)
  {
    .text0 { o1/*.o(.text) }
    .text1 { o2/*.o(.text) }
  }
```

This command defines both `.text0` and `.text1` to start at address 0x1000. '.text0' will be loaded at address 0x4000, and `.text1` will be loaded immediately after `.text0`. The following symbols will be defined if referenced: `__load_start_text0`, `__load_stop_text0`, `__load_start_text1`, and `__load_stop_text1`.

The C code to copy overlay `.text1` into the overlay area would look like the following.

```
extern char __load_start_text1, __load_stop_text1;
memcpy ((char *) 0x1000, &__load_start_text1,
         &__load_stop_text1 - &__load_start_text1);
```

Note that the `OVERLAY` command can be expressed using the more basic script commands. The above example could have been written identically as follows:

```
.text0 0x1000 : AT (0x4000) { o1/*.o(.text) }
PROVIDE (__load_start_text0 = LOADADDR (.text0));
PROVIDE (__load_stop_text0 = LOADADDR (.text0) + SIZEOF (.text0));
.text1 0x1000 : AT (0x4000 + SIZEOF (.text0)) { o2/*.o(.text) }
PROVIDE (__load_start_text1 = LOADADDR (.text1));
PROVIDE (__load_stop_text1 = LOADADDR (.text1) + SIZEOF (.text1));
. = 0x1000 + MAX (SIZEOF (.text0), SIZEOF (.text1));
```

3.3.10 ENTRY command

The linker command language includes a command specifically for defining the first executable instruction in an output file (its *entry point*). Its argument is a symbol name:

```
ENTRY(symbol)
```

Like symbol assignments, the `ENTRY` command can be placed either as an independent command in the command file, or among the section definitions within the `SECTIONS` command. Whatever makes the most sense for your layout.

`ENTRY` is only one of several ways of choosing the entry point. You can indicate it in any of the following ways (listed in descending order of priority: methods higher in the list override methods lower down):

- The `-e` *entry* command option
- The `ENTRY(`*symbol*`)` command in a linker control script
- The value of the symbol `start`, if present
- The address of the first byte of the `.text` section, if present
- The address `0`

For example, you can use these rules to generate an entry point with an assignment statement. If no symbol `start` is defined within your input files, you can simply define it, assigning it an appropriate value:

```
start = 0x2020;
```

The example shows an absolute address, but you can use any expression. For example, if your input object files use some other symbol-name convention for the entry point, you can just assign the value of whatever symbol contains the start address to `start`:

```
start = other_symbol ;
```

3.3.11 PHDRS command

The ELF object file format uses *program headers*, which are read by the system loader and describe how the program should be loaded into memory. These program headers must be set correctly to run the program on a native ELF system. The linker creates reasonable program headers by default. However, it can be useful to specify the program headers more precisely; the `PHDRS` command can be used to do this. When the `PHDRS` command is used, the linker does not generate any program headers itself.

NOTE This document does not describe the details of how the system loader interprets program headers; for more information see the ELF ABI. The program headers of an ELF file can be displayed using the `-p` option of the `objdump` utility.

The PHDRS command has the following syntax:

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

name is used only for reference in the SECTIONS command of the linker script. It does not get put into the output file.

Certain program header types describe segments of memory that are loaded from the file by the system loader. In the linker script, the contents of these segments are specified by directing allocated output sections to be placed in the segment. To do this, the command describing the output section in the SECTIONS command should use `:name`, where *name* is the name of the program header as it appears in the PHDRS command.
See Section 3.3.11.

It is normal for certain sections to appear in more than one segment. This merely implies that one segment of memory contains another. This is specified by repeating `:name`, using it once for each program header in which the section is to appear.

If a section is placed in one or more segments using `:name`, all subsequent allocated sections that do not specify :*name* are placed in the same segments. This is for convenience, since generally a whole set of contiguous sections is placed in a single segment. To prevent a section from being assigned to a segment when it would normally default to one, use :NONE.

The FILEHDR and PHDRS keywords, which can appear after the program header type, also indicate contents of the segment of memory. The FILEHDR keyword means that the segment should include the ELF file header. The PHDRS keyword means that the segment should include the ELF program headers themselves.

The *type* can be one of the following. The numbers indicate the value of the keyword.

PT_NULL (0)	Indicates an unused program header.
PT_LOAD (1)	Indicates that this program header describes a segment to be loaded from the file.
PT_DYNAMIC (2)	Indicates a segment where dynamic linking information can be found.
PT_INTERP (3)	Indicates a segment where the name of the program interpreter can be found.
PT_NOTE (4)	Indicates a segment holding note information.
PT_SHLIB (5)	A reserved program header type, defined but not specified by the ELF ABI.
PT_PHDR (6)	Indicates a segment where the program headers can be found.
expression	An expression giving the numeric type of the program header. This can be used for types not defined above.

You can specify that a segment should be loaded at a particular address in memory. This is done with an AT expression. This is identical to the AT command used in the SECTIONS command (Section 3.3.9). Using the AT command for a program header overrides any information in the SECTIONS command.

Normally, the segment flags are set based on the sections. The FLAGS keyword can be used to explicitly specify the segment flags. The value of *flags* must be an integer. It is used to set the p_flags field of the program header.

Here is an example of the use of PHDRS. This shows a typical set of program headers used on a native ELF system.

```
PHDRS {
  headers PT_PHDR PHDRS ;
  interp PT_INTERP ;
  text PT_LOAD FILEHDR PHDRS ;
  data PT_LOAD ;
  dynamic PT_DYNAMIC ;
}

SECTIONS {
  . = SIZEOF_HEADERS;
  .interp : { *(.interp) } :text :interp
  .text : { *(.text) } :text
  .rodata : { *(.rodata) } /* defaults to :text */
  ...
  . = . + 0x1000; /* move to a new page in memory */
  .data : { *(.data) } :data
  .dynamic : { *(.dynamic) } :data :dynamic
  ...
}
```

3.3.12 VERSION command

The linker command script includes a command for specifying a version script.

NOTE Version scripts are meaningful only for shared objects.

A version script can be build directly into the linker script that you are using, or you can supply the version script as just another input file to the linker at the time that you link. The command script syntax is:

VERSION { *version_script_contents* }

NOTE Version scripts can also be specified with the --version-script option (Section 3.2.2).

Versioning is specified in the script by defining a tree of version nodes with the names and interdependencies specified in the version script. The version script can specify which symbols are bound to which version nodes, and it can reduce a specified set of symbols to local scope so they are not globally visible outside of the shared object.

The following example demonstrates the version script language:

```
VERS_1.1 {
 global:
     foo1;
 local:
     old*;
     original*;
     new*;
};

VERS_1.2 {
     foo2;
} VERS_1.1;

VERS_2.0 {
     bar1; bar2;
} VERS_1.2;
```

In this example three version nodes are defined:

- `VERS_1.1` is the first version node defined. It has no other dependencies. The symbol `foo1` is bound to this version node, and a number of symbols that have appeared within various object files are reduced in scope to local so they are not visible outside of the shared object.
- Next, the node `VERS_1.2` is defined. It depends on `VERS_1.1`. The symbol `foo2` is bound to this version node.
- Finally, the node `VERS_2.0` is defined. It depends on `VERS_1.2`. The symbols `bar1` and `bar2` are bound to this version node.

Symbols defined in the library that aren't specifically bound to a version node are effectively bound to an unspecified base version of the library. You can bind all otherwise unspecified symbols to a given version node using `'global: *'` somewhere in the version script.

Lexically the names of the version nodes have no specific meaning other than what they might suggest to the person reading them. The `2.0` version definition could just as well have appeared in between the ones for `1.1` and `1.2`. However, this would be a confusing way to write a version script.

When you link an application against a shared object which has versioned symbols, the application itself knows which version of each symbol it requires; it also knows which version nodes it needs from each shared object it is linked against. Therefore, at runtime the dynamic loader can ensure that the libraries you linked against do in fact supply all the version nodes that the application needs to resolve all the dynamic symbols. In this way the dynamic linker can know with certainty that all the external symbols it needs are resolvable without having to search for each symbol reference.

The fundamental problem being addressed with this approach is that references to external functions are typically bound on an as-needed basis, and are not all bound when the application starts up. So if a shared object is out of date, a required interface may be missing; when the application tries to use that interface, it may suddenly and unexpectedly fail. With symbol versioning when starting a program, the user receives a warning if the libraries being used with the application are too old.

3.3.12.1 Source file symbol binding

Symbols can be bound to version nodes in the source file where the symbols are defined rather than in the versioning script. This is intended to reduce the burden on library maintainers. For example, the following declaration can appear in a C source file:

```
__asm__(".symver original_foo,foo@VERS_1.1");
```

This declaration renames the function `original_foo` to be an alias for `foo` bound to the version node `VERS_1.1`.

NOTE The `local:` directive can be used to prevent the symbol `original_foo` from being exported.

3.3.12.2 Multiple function versions

Multiple versions of the same function can appear in a given shared object. In this way an incompatible change to an interface can take place without increasing the major version number of the shared object, while still allowing applications linked against the old interface to continue to function.

This can only be accomplished by using multiple `.symver` directives in the assembler. For example:

```
__asm__(".symver original_foo,foo@");
__asm__(".symver old_foo,foo@VERS_1.1");
__asm__(".symver old_foo1,foo@VERS_1.2");
__asm__(".symver new_foo,foo@@VERS_2.0");
```

In this example `foo@` represents the symbol `foo` bound to the unspecified base version of the symbol. The source file that contains this example would define four C functions: `original_foo`, `old_foo`, `old_foo1`, and `new_foo`.

3.3.12.3 Multiple symbol definitions

When you have multiple definitions of a given symbol, there needs to be some way to specify a default version to which external references to this symbol are bound. This can be accomplished with the `foo@@VERS_2.0` type of `.symver` directive. Only one version of a symbol can be declared 'default' in this manner, otherwise you would effectively have multiple definitions of the same symbol.

3.3.12.4 External symbol binding

To bind a reference to a specific version of a symbol in a shared object, you can use the aliases of convenience *(i.e.,* `old_foo`). Or use the `.symver` directive to specifically bind to an external version of the function in question.

You can also specify the language in the version script:

```
VERSION extern "lang" { version-script-commands }
```

The possible value for *lang* are "`C`" and "`C++`". At link time the linker iterates over the list of symbols and demangles them according to *`lang`* before matching them to the patterns specified in *`version-script-commands`*.

Demangled names can contain spaces and other special characters. Wildcard patterns (Section 3.3.9.4) can be used to match demangled names, or a double-quoted string to match the string exactly. In the latter case, be aware that minor differences (such as differing whitespace) between the version script and the demangler output will cause a mismatch. As the exact string generated by the demangler might change in the future, even if the mangled name does not, you should check that all of your version directives are behaving as you expect when you upgrade.

3.3.13 Other commands

The linker script command language includes a number of commands that are used for specialized purposes. They are similar in purpose to command-line options.

`AS_NEEDED ( `*`file, file, ...`*` )`
: Treat the specified files as if they appear directly in an `INPUT` or `GROUP` command, except for shared libraries, which are added only when they are actually needed.

  This command limits the adding of `DT_NEEDED` tags to libraries that satisfy a symbol reference (from regular objects) which is undefined when the library was linked, or – if the library is not found in the `DT_NEEDED` lists of the other libraries linked up to that point – a reference from another shared library. It is equivalent to the `--as-needed` option.

  This command can be used only in the `INPUT` or `GROUP` command, among other filenames.

`ASSERT (`*`expr, message`*`)`
: Verify that the specified expression is non-zero. If it is zero, exit the linker with an error code and print the specified message.

`EXTERN ( `*`symbol symbol`*` ... )`
: Force the specified symbols to be entered in the output file as undefined symbols. Doing this may trigger linking of additional modules from standard libraries. This command can be used multiple times. It has the same effect as the `-u` command-line option.

`FORCE_COMMON_ALLOCATION`
: Force the linker to assign space to common symbols even if a relocatable output file is specified (`-r`). This command has the same effect as the `-d` command-line option.

GROUP (*file, file, ...*)
GROUP (*file file ...*)

This command is like `INPUT`, except that the named files should all be archives, and they are searched repeatedly until no new undefined references are created. See the description of "`-(`" in Section 3.2.1.

INCLUDE *file*

Include the linker script *file* at this point. The file will be searched for in the current directory and in any directory specified with the `-L` option. You can nest calls to `INCLUDE` up to 10 levels deep.

`INCLUDE` can be used at the top level in a linker script, in `MEMORY` or `SECTIONS` commands, or in output section descriptions.

INHIBIT_COMMON_ALLOCATION

Force the linker to omit the assignment of addresses to common symbols even for a non-relocatable output file. This command has the same effect as the `--no-define-common` command-line option.

INPUT (*file, file, ...*)
INPUT (*file file ...*)

Include binary input files in the link as if they were specified on the command line.

If all the linker input files are specified in the linker script, the program can be linked by invoking the linker with no arguments but the command option `-T`.

If a *sysroot prefix* is configured, and the filename starts with the `/` character, and the script being processed was located inside the *sysroot prefix*, the filename is searched for in the *sysroot prefix*. Otherwise, the linker tries to open the file in the current directory. If it is not found, the linker searches through the archive library search path. (For details see the command option `-L`.)

If a filename argument is specified in `INPUT` as `-l`*`file`*, the linker automatically transforms the name to `libfile.a` (as is done by the command option `-l`).

When you use the `INPUT` command in an implicit linker script, the files are included in the link at the point at which the linker script file is included. This can affect archive searching.

INSERT [**AFTER** | **BEFORE**] *output_section*

Insert all previous linker script statements before or after the specified section, and cause `-T` to not override the default script. The specific point of insertion is the same as that used for orphan sections (Section 3.3.6.3). This command is used in linker scripts specified by the `-T` option.

Example:

```
SECTIONS
{
    OVERLAY :
    {
        .ov1 { ov1*(.text) }
         .ov2 { ov2*(.text) }
    }
}
INSERT AFTER .text;
```

NOTE Linker script statements are inserted *after* the linker maps the input sections to output sections. Prior to insertion, statements in the `-T` script occur before the default script statements in the internal linker representation of the script (because `-T` scripts are parsed before the default script). In particular, input section assignments are mapped to `-T` output sections before those in the default script.

`NOCROSSREFS ( `*`section section`*` ... )`

Direct the linker to generate an error for any references among the specified sections.

In certain types of programs, particularly on embedded systems, when one section is loaded into memory, another section is not. Any direct references between the two sections would be errors. For example, an error would occur if code in one section called a function defined in the other section.

The `NOCROSSREFS` command accepts a list of section names. If the linker detects any cross references between the sections, it generates an error and returns a non-zero exit status. The `NOCROSSREFS` command uses output section names, defined in the `SECTIONS` command. It does not use the names of input sections.

`OUTPUT ( `*`file`*` )`

Use this command to name the link output file with the specified file name. The effect of `OUTPUT(`*file*`)` is identical to the effect of `-o` *file*, which overrides it. You can use this command to supply a default output file name.

`OUTPUT_ARCH ( `*`bfdname`*` )`

Specify a particular output machine architecture, with one of the names used by the BFD back-end routines (Section 3.5). This command is often unnecessary; the architecture is most often set implicitly by either the system BFD configuration or as a side effect of the `OUTPUT_FORMAT` command.

`OUTPUT_FORMAT ( `*`bfdname`*` )`

When the linker is configured to support multiple object code formats, you can use this command to specify a particular output format. *bfdname* is one of the names used by the BFD back-end routines (Section 3.5). The effect is identical to the effect of the `--oformat` command-line option. This selection affects only the output file. The related command `TARGET` affects primarily input files.

`REGION_ALIAS ( `*`alias, region`*` )`

Creates an alias named *alias* for the specified memory *region*.This allows flexible mapping of output sections to memory regions.

`SEARCH_DIR ( `*`path`*` )`

Add *path* to the list of paths where the linker looks for archive libraries. `SEARCH_DIR(`*path*`)` has the same effect as the command option `-L`*path*.

If both the script command and the command option are used, the linker searches both paths; paths specified using the option are searched first.

`STARTUP ( `*`file`*` )`

Ensure that *file* is the first input file used in the link process.

The command is equivalent to the `INPUT` command, except that the specified file name becomes the first input file to be linked (as though it were specified first on the command line).

TARGET (*format*)

When the linker is configured to support multiple object code formats, you can use this command to change the input-file object code format (like the command-line option `-b` or its synonym `--format`). The argument *format* is one of the strings used by BFD to name binary formats. If `TARGET` is specified but `OUTPUT_FORMAT` is not, the last `TARGET` argument is also used as the default format for the linker output file (Section 3.5). If you do not use the `TARGET` command, the linker uses the value of the environment variable `GNUTARGET`, if available, to select the output file format. If that variable is also absent, the linker uses the default format configured for your machine in the BFD libraries.

3.4 Processor-specific memory layout

The Hexagon processor's memory layouts require the following values which are not supplied by the `MEMORY` and `SECTIONS` commands (Section 3.3.8 and Section 3.3.9):

- The base addresses of the Hexagon processor memories (EBI, TCM, SMI) in both virtual and physical memory.
- The base-relative addresses of the Hexagon processor memory sections (Section 2.4.4) within their corresponding memories.

These values are normally assigned automatically as part of the application build process; however, for stand-alone applications that require nonstandard memory layouts, the values must be explicitly assigned by the user.

This section describes the Hexagon processor memories and the features that support user-specified Hexagon memory layouts.

3.4.1 Hexagon processor memories

The Hexagon processor can access different types of memory. Table 3-2 lists the memories and their uses.

Table 3-2 Processor memories

Name	Description	Function
EBI	External bus interface	Main memory
TCM	Tightly-coupled memory	Fast internal memory
SMI	Stacked memory interface	External memory

Program code and data are stored by default in EBI memory; however, the user can explicitly assign them to EBI, TCM, or SMI by using the corresponding Hexagon processor memory sections (Section 2.4.4). For example:

C code:

```
int ainit[4]__attribute__((section(".tcm_data_uncached"))) =
                            {1, 2, 3, 4}; void foo()
__attribute__((section(".tcm_code_cached")));
```

Assembly code:

```
.section .ebi_code_cached
```

NOTE Static data (such as C globals) and interrupt service routine code are typically assigned to TCM for improved performance.

Section memory assignment

The processor memory sections *must* be assigned to specific memory areas before they can be used. This can be done in one of two ways:

- Use the `-tcm` option (Section 3.2.2) to automatically assign the memory sections using the linker's predefined memory address symbols (Section 3.4.2).
- Modify the linker script to use the `SECTIONS` command (Section 3.3.9) to map the default memory sections (`.text`, `.data`, etc.) to processor memory sections.

3.4.2 Address mapping

The memories are mapped to specific address ranges in the Hexagon processor's virtual and physical memory address spaces.

For example, Figure 3-2 shows the memory mapping for an application which uses EBI, TCM, and SMI memories:

- The EBI, TCM, and SMI parts of the application are defined to occupy a continuous range of addresses (0-9000) in virtual memory.
- The non-TCM (i.e., EBI and SMI) parts of an application are defined to have a direct 1:1 mapping between virtual and physical memory addresses.
- TCM memory is defined to occupy 256K bytes of memory starting at base address 0xD8000000 (the default value).
- The TCM memory sections of the application (i.e., `tcm_data_cached` and `tcm_data_cached_wt`) are aligned to 4K page boundaries in both the virtual and physical address spaces.

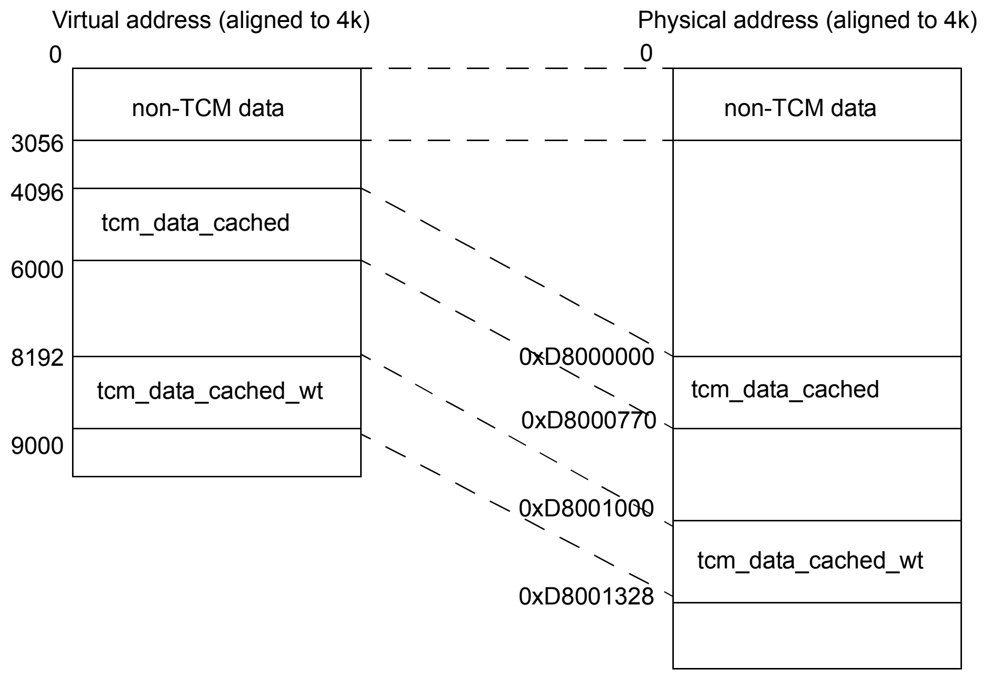

Figure 3-2 TCM memory map

The default application build process automatically assigns all application code and data to a contiguous area of EBI memory starting at base address 0.

To provide user control over the base addresses of the EBI/TCM/SMI memories and sections, the linker predefines a number of symbols which can be set from the linker command line. For example:

```
hexagon-ld --defsym TCM_PA_START=0xD8000000 app1.o
hexagon-ld --defsym TCM_PA_START=0xD8000000+sizeof(app1.o) app2.o
```

In the above example, the predefined symbol `TCM_PA_START` is used to set the physical base address of TCM memory for two applications `app1` and `app2`. The base address of `app1` is set to `0xD8000000`, while the base address for `app2` is set to immediately follow the TCM used by `app1`.

User-defined memory layouts must conform to the following restrictions:

- Sections must start on 4K page boundaries in both virtual and physical memory.
- Sections within an application must not overlap in virtual memory.
- Sections within and between applications must not overlap in physical memory.

Table 3-3 lists the predefined symbols for specifying the memory layout.

Table 3-3 Hexagon processor address symbols

Symbol	Description	Default Value
EBI_VA_START	Base address of EBI/TCM/SMI in virtual memory	.
TCM_VA_START		
SMI_VA_START		
EBI_PA_START	Base address of EBI/TCM/SMI in physical memory	.
TCM_PA_START		0xD8000000
SMI_PA_START		.
EBI_CODE_CACHED_ALIGN	Base-address-relative offset of EBI/TCM/SMI code sections in physical memory	0x1000
TCM_CODE_CACHED_ALIGN		
SMI_CODE_CACHED_ALIGN		
EBI_DATA_CACHED_ALIGN	Base-address-relative offset of EBI/TCM/SMI data sections in physical memory (cached data)	0x1000
TCM_DATA_CACHED_ALIGN		
SMI_DATA_CACHED_ALIGN		
EBI_DATA_CACHED_WT_ALIGN	Base-address-relative offset of EBI/TCM/SMI data section in physical memory (write-through cached data)	0x1000
TCM_DATA_CACHED_WT_ALIGN		
SMI_DATA_CACHED_WT_ALIGN		
EBI_DATA_UNCACHED_ALIGN	Base-address-relative offset of EBI/TCM/SMI data sections in physical memory (uncached data)	0x1000
TCM_DATA_UNCACHED_ALIGN		
SMI_DATA_UNCACHED_ALIGN		

NOTE The symbol '`.`' in Table 3-3 indicates that by default the linker assigns the section to the next available address in memory (starting at zero).

The EBI/TCM/SMI sections are assigned by default to start on 4K page boundaries in both virtual and physical memory.

To provide user control over the memory alignment of these sections, the linker predefines the `*_ALIGN` symbols which can be set from the linker command line. For example:

```
hexagon-ld --defsym TCM_DATA_CACHED_ALIGN=0x4000
```

NOTE The `--section-start` option (Section 3.2.2) can alternatively be used to relocate individual sections to specific addresses in their memory areas.

3.4.3 MMU programming

Stand-alone applications that use nonstandard memory layouts must explicitly program the Hexagon processor's memory management unit (MMU) with the address mapping (Section 3.4.2) for each Hexagon processor memory. The application code performs this task at runtime by calling the processor-specific library function `add_translation`. For example:

```
add_translation(4096, 0xD8000000, 2);
add_translation(8192, 0xD8001000, 4);
```

`add_translation` accepts three parameters:

- A virtual memory address
- The corresponding physical memory address (which depending on the address value may access any of the memories)
- A mode parameter specifying the cache properties

For more information on `add_translation` see the *Hexagon Stand-alone Application Build Guide*.

For more information on memory management see the *Hexagon V2 Programmer's Reference Manual*.

3.5 Binary file access

The linker accesses object and archive files using the *Binary File Descriptor* (BFD) libraries. These libraries allow the linker to use the same routines to operate on object files whatever the object file format. A different object file format can be supported simply by creating a new BFD back end and adding it to the library. To conserve runtime memory, however, the linker and associated tools are usually configured to support only a subset of the object file formats available. You can use `objdump -i` (Section 4.6) to list all the formats available for your configuration.

As with most implementations, BFD is a compromise between several conflicting requirements. The major factor influencing BFD design was efficiency; any time used converting between formats is time which would not have been spent had BFD not been involved. This is partly offset by abstraction payback; since BFD simplifies applications and back ends, more time and care can be spent optimizing algorithms for a greater speed.

One minor artifact of the BFD solution that you should bear in mind is the potential for information loss. There are two places where useful information can be lost using the BFD mechanism: during conversion and during output. See Section 3.5.2.

3.5.1 Functional overview

When an object file is opened, BFD subroutines automatically determine the format of the input object file. They then build a descriptor in memory with pointers to routines that will be used to access elements of the object file data structures.

As different information from the object files is required, BFD reads from different sections of the file and processes them. For example, a very common operation for the linker is processing symbol tables. Each BFD back end provides a routine for converting between the object file's representation of symbols and an internal canonical format. When the linker asks for the symbol table of an object file, it calls through a memory pointer to the routine from the relevant BFD back end that reads and converts the table into a canonical form. The linker then operates upon the canonical form. When the link is finished and the linker writes the output file's symbol table, another BFD back-end routine is called to take the newly created symbol table and convert it into the chosen output format.

3.5.2 Information loss

Information can be lost during output. The output formats supported by BFD do not provide identical facilities, and information that can be described in one form has nowhere to go in another format. One example of this is alignment information in `b.out`. There is nowhere in an `a.out` format file to store alignment information on the contained data, so when a file is linked from `b.out` and an `a.out` image is produced, alignment information will not propagate to the output file. (The linker will still use the alignment information internally, so the link is performed correctly).

Information can be lost during canonicalization. The BFD internal canonical form of the external formats is not exhaustive; there are structures in input formats for which there is no direct representation internally. This means that the BFD back ends cannot maintain all possible data richness through the transformation between external to internal and back to external formats.

This limitation is only a problem when an application reads one format and writes another. Each BFD back end is responsible for maintaining as much data as possible, and the internal BFD canonical form has structures which are opaque to the BFD core, and exported only to the back ends. When a file is read in one format, the canonical form is generated for BFD and the application. At the same time, the back end saves any information which may otherwise be lost. If the data is then written back in the same format, the back end routine will be able to use the canonical form provided by the BFD core as well as the information it prepared earlier. Since there is a great deal of commonality between back ends, there is no information lost when linking or copying big-endian COFF to little-endian COFF, or `a.out` to `b.out`. When a mixture of formats is linked, the information is only lost from the files whose format differs from the destination.

3.5.3 Canonical object file format

The greatest potential for loss of information occurs when there is the least overlap between the information provided by the source format, that stored by the canonical format, and that needed by the destination format. A brief description of the canonical form may help you understand which kinds of data you can count on preserving across conversions.

`files`

Information stored on a per-file basis includes target machine architecture, particular implementation format type, a demand pageable bit, and a write protected bit. Information like UNIX magic numbers is not stored here, only the magic numbers' meaning. So, a `ZMAGIC` file would have both the demand pageable bit and the write protected text bit set. The byte order of the target is stored on a per-file basis so that big- and little-endian object files can be used with one another.

`sections`

Each section in the input file contains the name of the section, the section's original address in the object file, size and alignment information, various flags, and pointers into other BFD data structures.

`symbols`

Each symbol contains a pointer to the information for the object file which originally defined it, its name, its value, and various flag bits. When a BFD back end reads in a symbol table, it relocates all symbols to make them relative to the base of the section where they were defined. Doing this ensures that each symbol points to its containing section. Each symbol also has a varying amount of hidden private data for the BFD back end. Since the symbol points to the original file, the private data format for that symbol is accessible. `ld` can operate on a collection of symbols of wildly different formats without problems. Normal global and simple local symbols are maintained on output, so an output file (no matter its format) will retain symbols pointing to functions and to global, static, and common variables. Some symbol information is not worth retaining; in `a.out`, type information is stored in the symbol table as long symbol names. This information would be useless to most COFF debuggers; the linker has command line switches to allow users to throw it away. There is one word of type information within the symbol, so if the format supports symbol type information within symbols (for example, COFF, IEEE, Oasys) and the type is simple enough to fit within one word (nearly everything but aggregates), the information will be preserved.

`relocation level`

Each canonical BFD relocation record contains a pointer to the symbol to relocate to, the offset of the data to relocate, the section the data is in, and a pointer to a relocation type descriptor. Relocation is performed by passing messages through the relocation type descriptor and the symbol pointer. Therefore, relocations can be performed on output data using a relocation method that is only available in one of the input formats. For instance, Oasys provides a byte relocation format. A relocation record requesting this relocation type would point indirectly to a routine to perform this, so the relocation can be performed on a byte being written to a 68k COFF file, even though 68k COFF has no such relocation type.

`line numbers`

Object formats can contain, for debugging purposes, some form of mapping between symbols, source line numbers, and addresses in the output file. These addresses have to be relocated along with the symbol information. Each symbol with an associated list of line number records points to the first record of the list. The head of a line number list consists of a pointer to the symbol, which allows finding out the address of the function whose line number is being described. The rest of the list is made up of pairs: offsets into the section and line numbers. Any format that can simply derive this information can pass it successfully between formats (COFF, IEEE, and Oasys).

3.6 Processor-specific features

The linker supports the following features which are specific to the Hexagon processor:

Data alignment

The linker predefines the following symbols for aligning the default sections (Section 3.3.9) in memory:

- `TEXTALIGN`
- `DATAALIGN`
- `RODATAALIGN`

These symbols are used to align `.text`, `.data`, and `.rodata` sections (Section 2.4.1) to specific page boundaries in memory.

If these symbols are defined on the linker command line, they cause the corresponding sections to be aligned to the specified 1K-byte address boundary. For example:

```
hexagon-ld --defsym DATAALIGN=256
```

This specifies that the data section will be aligned on a 256K byte boundary.

If these symbols are not defined on the linker command line, their default value is 4K (i.e., 4 * 1K).

Hexagon processor memory layout

The linker predefines the following symbols for assigning code and data to the Hexagon processor’s memories:

- `(EBI|SMI|TCM)_VA_START`
- `(EBI|SMI|TCM)_PA_START`
- `(EBI|SMI|TCM)_*_ALIGN`

For more information see Section 3.4.

Command-line options

The linker supports the following command-line options (Section 3.2.2):

- `--force-dynamic`
- `-G, --gpsize`
- `-march, -mcpu, -mv2, -mv3, -mv4`
- `-trampolines`

4 Utilities

4.1 Overview

The software development tools for the Hexagon processor include the utility programs listed in Table 4-1.

Table 4-1 Software development utilities

Utility	Description
Archiver	Create, modify, and extract files from archives
Object file symbols	List symbols from object files
Object file copier	Copy and translate object files
Object file viewer	Display information from object files
Archive indexer	Generate index to archive contents
Object file size	List section sizes and total size
Object file strings	List strings from object files
Object file stripper	Remove symbols from object file
C++ filter	Filter to demangle encoded C++ symbols
Address converter	Convert addresses to file and line
ELF file viewer	Display contents of ELF format files

NOTE These utility programs are based on GNU Binutils.

4.2 Using the utilities

Utilities are started from the command line by typing the utility command name followed by one or more file names and command options. For example:

```
hexagon-objcopy myfile --p --debugging
```

Many utility options have alternate abbreviated switches defined for ease of use. These long and short forms, listed in the option descriptions as alternatives, are functionally equivalent. The standard option names can also be truncated, as long as you specify enough of the option name to uniquely identify the option.

NOTE Option names can be prefixed with either “`-`” or “`--`”.

Option files

Utility command arguments can be specified in a text file rather than on the command line. The file is specified on the command line as the argument of the special command option `@`. For example:

```
hexagon-objcopy @myoptions
```

Here the file named *myoptions* contains the command arguments for the `hexagon-copy` command. The file contents are inserted into the command line in place of the specified `@`*file* option.

NOTE Command arguments stored in argument files must be delimited by whitespace characters.

The `@`*file* option can appear in argument files – it is processed recursively.

4.3 Archiver

```
hexagon-ar [-]c[mod...] [member_name] [count] archive_file
[file...]
hexagon-ar -M [script_file]
```

The archiver utility creates, modifies, and extracts from archives. An *archive* is a single file holding a collection of other files in a structure that makes it possible to retrieve the original individual files from the archive. When a file is stored in an archive it is referred to as a *member* of the archive.

The original files' contents, mode (permissions), timestamp, owner, and group are preserved in the archive, and can be restored on extraction.

The archiver is considered a binary utility because archives of this sort are most often used as *libraries* holding commonly needed subroutines.

The archiver creates an index to the symbols defined in relocatable object modules in the archive when you specify the `s` modifier option. Once created, this index is updated in the archive whenever the archiver makes a change to its contents (save for the `q` update operation). An archive with such an index speeds up linking to the library, and allows routines in the library to call each other without regard to their placement in the archive.

You can use the commands "`hexagon-nm -s`" or "`hexagon-nm --print-armap`" to list this index table. If an archive lacks the table, another form of the archiver (Section 4.7) an be used to add just the table.

The archiver can optionally create *thin* archives, which contain only a symbol index and references to the original copies of the archive's member files. Thin archives are useful for building libraries used in local builds (where the relocatable objects are expected to remain available, and copying each object would be wasteful). Adding one or more archives to a thin archive adds the elements of the nested archive individually. Paths to a thin archive's elements are stored relative to the archive itself.

The archiver can be controlled in two ways:

- Using command-line options
- Using a script supplied via the standard input (with the `-M` option)

4.3.1 Archiver options

When you use the archiver in the UNIX style, the archiver insists on at least two arguments to execute:

- A *command* (optionally accompanied by one or more *modifiers*)
- The name of the archive file that the command will be applied to

The command and modifiers (which are shown as *`c`* and *`mod`* in the archiver command syntax) are specified by option letters which appear in the first command-line argument (which begins with a '`-`' character). For example, the argument `-dv` specifies the command option `d` and the modifier option `v`. The option letters can appear in any order within the argument.

Most archiver operations also accept additional arguments, specifying particular files or archive members to operate on.

4.3.1.1 Command options

The *c* option specifies what operation to execute; it can be any of the following, but you must specify only one of them:

`d`

Delete the specified modules from the archive. Modules are specified as file arguments; the archive is untouched if no files are specified.

If you specify the `v` modifier, the archiver lists each module as it is deleted.

`m`

Use this operation to *move* members in an archive.

The ordering of members in an archive can make a difference in how programs are linked using the library, if a symbol is defined in more than one member.

If no modifiers are used with `m`, any members you name in the file arguments are moved to the *end* of the archive; you can use the `a` or `b` modifiers to move them to a specified place instead.

`p`

Print the specified members of the archive to the standard output. If the `v` modifier is specified, show the member name before copying its contents to standard output.

If you specify no file arguments, all the files in the archive are printed.

q

Quick append. Historically, add the specified files to the end of *archive*, without checking for replacement.

The modifiers `a` and `b` do *not* affect this operation; new members are always placed at the end of the archive.

The modifier `v` makes the archiver list each file as it is appended.

Since the point of this operation is speed, the archive's symbol table index is not updated, even if it already existed; you can use the command "`hexagon-ar s`" or the archive indexer (Section 4.7) to update the symbol table index.

However, too many different systems assume quick append rebuilds the index, so the archiver implements `q` as a synonym for `r`.

r

Insert the specified files into the archive (with *replacement*). This operation differs from `q` in that any previously existing members are deleted if their names match those being added.

If one of the specified files does not exist, the archiver displays an error message, and leaves undisturbed any existing members of the archive matching that name.

By default, new members are added at the end of the archive file; but you can use one of the modifiers `a` or `b` to request placement relative to some existing member.

The modifier `v` used with this operation elicits a line of output for each file inserted, along with one of the letters `a` or `r` to indicate whether the file was appended (no old member deleted) or replaced.

t

Display a *table* listing the contents of the specified archive, or those of the specified files that are present in the archive. Normally only the archive member names are displayed; if you also want to see the modes (permissions), timestamp, owner, group, and size, you can request that by also specifying the `v` modifier.

If you do not specify a file, all files in the archive are listed.

If more than one file with the same name (say, `fie`) exists in an archive (say `b.a`), the command "`hexagon-ar -t b.a fie`" lists only the first instance; to see them all, you must generate a complete listing: in our example, "`hexagon-ar -t b.a`".

x

Extract members (named *member*) from the archive. You can use the `v` modifier with this operation, to request that the archiver list each name as it extracts it.

If you do not specify a file, all files in the archive are extracted.

NOTE Files cannot be extracted from a thin archive (Section 4.3).

4.3.1.2 Modifier options

Modifier options can immediately follow a command option, to specify variations on an operation's behavior:

`a`

Add new files *after* an existing member of the archive. If you use the modifier `a`, the name of an existing archive member must be specified as the *member_name* argument.

`b`

Add new files *before* an existing member of the archive. If you use the modifier `b`, the name of an existing archive member must be present as the *member_name* argument.

`c`

Create the archive. The specified archive file is always created if it did not exist, when you request an update. But a warning is issued unless you specify in advance that you expect to create it, by using this modifier.

`f`

Truncate names in the archive. The archiver will normally permit file names of any length. This will cause it to create archives which are not compatible with the native archiver programs on some systems. If this is a concern, the `f` modifier can be used to truncate file names when putting them in the archive.

`N`

Use the *count* parameter. This is used if there are multiple entries in the archive with the same name. Extract or delete instance *count* of the specified name from the archive.

`o`

Preserve the *original* dates of members when extracting them from the archive. If you do not specify this option, files extracted from the archive are stamped with the time of extraction.

`P`

Use the full path name when matching names in the archive. The archiver can not create an archive with a full path name (such archives are not POSIX complaint), but other archive creators can. This option will cause the archiver to match file names using a complete path name, which can be convenient when extracting a single file from an archive created by another tool.

`s`

Write an object-file index into the archive, or update an existing one, even if no other change is made to the archive. You can use this option either with any operation, or alone. Running the command "`hexagon-ar s`" on an archive is equivalent to running the archive indexer (Section 4.7) on it.

`S`

Do not generate an archive symbol table. This can speed up building a large library in several steps. The resulting archive can not be used with the linker. In order to build a symbol table, you must omit the `S` modifier on the last execution of the archiver, or you must run the archive indexer (Section 4.7) on the archive.

T

Specify the archive as a *thin* archive (Section 4.3). If the specified archive already exists and is not a thin archive, then the existing members must reside in the same directory as the archive.

u

Normally, the command "`hexagon-ar r`..." inserts all files listed into the archive. If you would like to insert *only* those of the files you list that are newer than existing members of the same names, use this modifier. The `u` option is allowed only for the operation `r` (replace). In particular, the combination `qu` is not allowed, since checking the timestamps would lose any speed advantage from the operation `q`.

v

Request the *verbose* version of an operation. Many operations display additional information, such as filenames processed, when the modifier `v` is appended.

V

Display the version number of the archiver.

4.3.2 Archiver command scripts

If you use the single command-line option `-M` with the archiver, you can control its operation with a rudimentary command language. This form of the archiver operates interactively if standard input is coming directly from a terminal. During interactive use, the archiver prompts for input (the prompt is "`ar >`"), and continues executing even after errors. If you redirect standard input to a script file, no prompts are issued, and the archiver abandons execution (with a nonzero exit code) on any error.

The the archiver command language is *not* designed to be equivalent to the command-line options; in fact, it provides somewhat less control over archives. The only purpose of the command language is to ease the transition to the archiver for developers who already have scripts written for the MRI "librarian" program.

The syntax for the archiver command language is straightforward:

- Commands are recognized in upper or lower case; for example, `LIST` is the same as `list`. In the following descriptions, commands are shown in upper case for clarity.
- A single command can appear on each line; it is the first word on the line.
- Empty lines are allowed, and have no effect.
- Comments are allowed; text after either of the characters `*` or `;` is ignored.
- Whenever you use a list of names as part of the argument to an archiver command, you can separate the individual names with either commas or blanks. Commas are shown in the explanations below, for clarity.
- `+` is used as a line continuation character; if `+` appears at the end of a line, the text on the following line is considered part of the current command.

Below are the commands used in archiver scripts, or when using the archiver interactively. Three of them have special significance:

- `OPEN` or `CREATE` specify a *current archive*, which is a temporary file required for most of the other commands.
- `SAVE` commits the changes so far specified by the script. Prior to `SAVE`, commands affect only the temporary copy of the current archive.

`ADDLIB` *`archive`*
`ADDLIB` *`archive`* `(`*`module`*`,` *`module`*`, ...` *`module`*`)`

Add all the contents of *archive* (or, if specified, each named *module* from *archive*) to the current archive.

Requires prior use of `OPEN` or `CREATE`.

`ADDMOD` *`member`*`,` *`member`*`, ...` *`member`*

Add each named *member* as a module in the current archive.

Requires prior use of `OPEN` or `CREATE`.

`CLEAR`

Discard the contents of the current archive, canceling the effect of any operations since the last `SAVE`. May be executed (with no effect) even if no current archive is specified.

`CREATE` *`archive`*

Creates an archive, and makes it the current archive (required for many other commands). The new archive is created with a temporary name; it is not actually saved as *archive* until you use `SAVE`. You can overwrite existing archives; similarly, the contents of any existing file named *archive* will not be destroyed until `SAVE`.

`DELETE` *`module`*`,` *`module`*`, ...` *`module`*

Delete the specified modules from the current archive. This is equivalent to the following command: `hexagon-ar -d` *`archive module`* `...` *`module`*

Requires prior use of `OPEN` or `CREATE`.

`DIRECTORY archive` `(`*`module`*`, ...` *`module`*`)`
`DIRECTORY archive` `(`*`module`*`, ...` *`module`*`)` *`outputfile`*

List each named *module* present in *archive*. The separate command `VERBOSE` specifies the form of the output: when verbose output is off, output is equivalent to that of "`hexagon-ar -t` *archive module*. . .". When verbose output is on, the listing is equivalent to "`hexagon-ar -tv` *archive module*. . .".

Output normally goes to the standard output stream; however, if you specify *outputfile* as a final argument, the archiver directs the output to that file.

`END`

Exit from the archiver, with a `0` exit code to indicate successful completion. This command does not save the output file; if you have changed the current archive since the last `SAVE` command, those changes are lost.

`EXTRACT` *`module`*`,` *`module`*`, ...` *`module`*

Extract each named *module* from the current archive, writing them into the current directory as separate files. Equivalent to "`hexagon-ar -x` *archive module*. . .".

Requires prior use of `OPEN` or `CREATE`.

`LIST`

Display full contents of the current archive, in "verbose" style regardless of the state of `VERBOSE`. The effect is like "`hexagon-ar tv` *archive*". (This single command is a GNU enhancement, rather than present for MRI compatibility.)

Requires prior use of `OPEN` or `CREATE`.

`OPEN` *`archive`*

Opens an existing archive for use as the current archive (required for many other commands). Any changes as the result of subsequent commands will not actually affect *archive* until you next use `SAVE`.

`REPLACE` *`module`*`,` *`module`*`, ...` *`module`*

In the current archive, replace each existing *module* (named in the `REPLACE` arguments) from files in the current working directory. To execute this command without errors, both the file, and the module in the current archive, must exist.

Requires prior use of `OPEN` or `CREATE`.

`VERBOSE`

Toggle an internal flag governing the output from `DIRECTORY`. When the flag is on, `DIRECTORY` output matches output from "`hexagon-ar -tv` ...".

`SAVE`

Commit your changes to the current archive, and actually save it as a file with the name specified in the last `CREATE` or `OPEN` command.

Requires prior use of `OPEN` or `CREATE`.

4.4 Object file symbols

```
hexagon-nm [-a|--debug-syms] [-g|--extern-only]
[-B] [-C|--demangle [style]] [-D|--dynamic]
[-S|--print-size] [-s|--print-armap]
[-A|-o|--print-file-name] [--special-syms]
[-n|-v|--numeric-sort] [-p|--no-sort] [-r|--reverse-sort]
[--size-sort] [-u|--undefined-only]
[-t radix|--radix radix] [--defined-only] [-l|--line-numbers]
[--no-demangle] [-V|--version] [--help]
[objfile...]
```

The object file symbols utility lists the symbols defined in the specified object files. If no object files are listed as arguments, the symbols utility assumes the file `a.out`.

For each symbol the symbols utility shows:

- The symbol value, in the radix selected by options (see below), or hexadecimal by default.
- The symbol type. At least the following types are used; others are, as well, depending on the object file format. If lowercase, the symbol is local; if uppercase, the symbol is global (external).

The symbols utility options are described below.

`A`
: The symbol's value is absolute, and will not be changed by further linking.

`B`
: The symbol is in the uninitialized data section (known as BSS).

`C`
: The symbol is common. Common symbols are uninitialized data. When linking, multiple common symbols can appear with the same name. If the symbol is defined anywhere, the common symbols are treated as undefined references.

`D`
: The symbol is in the initialized data section.

`G`
: The symbol is in an initialized data section for small objects. Some object file formats permit more efficient access to small data objects, such as a global int variable as opposed to a large global array.

`I`
: The symbol is an indirect reference to another symbol. This is a GNU extension to the a.out object file format which is rarely used.

`N`
: The symbol is a debugging symbol.

`R`
: The symbol is in a read only data section.

`S`
: The symbol is in an uninitialized data section for small objects.

`T`
: The symbol is in the text (code) section.

`U`
: The symbol is undefined.

`V`
: The symbol is a weak object. When a weak defined symbol is linked with a normal defined symbol, the normal defined symbol is used with no error. When a weak undefined symbol is linked and the symbol is not defined, the value of the weak symbol becomes zero with no error.

`W`
: The symbol is a weak symbol that has not been specifically tagged as a weak object symbol. When a weak defined symbol is linked with a normal defined symbol, the normal defined symbol is used with no error. When a weak undefined symbol is linked and the symbol is not defined, the value of the weak symbol becomes zero with no error.

`-`
: The symbol is a stabs symbol in an a.out object file. In this case, the next values printed are the stabs other field, the stabs desc field, and the stab type. Stabs symbols are used to hold debugging information.

?

The symbol type is unknown, or object-file-format specific.

- The symbol name.

The long and short forms of options, shown here as alternatives, are equivalent.

-A
-o
--print-file-name
Precede each symbol by the name of the input file (or archive member) in which it was found, rather than identifying the input file once only, before all of its symbols.

-a
--debug-syms
Display all symbols, even debugger-only symbols; normally these are not listed.

-B
Equivalent to `--format=bsd`.

-C
--demangle [*style*]
Decode (*demangle*) low-level symbol names into user-level names. Besides removing any initial underscore prepended by the system, this makes C++ function names readable. Different compilers have different mangling styles. The optional demangling style argument can be used to choose an appropriate demangling style for your compiler. See Section 4.11 for more information on demangling.

--no-demangle
Do not demangle low-level symbol names. This is the default.

-D
--dynamic
Display the dynamic symbols rather than the normal symbols. This is only meaningful for dynamic objects, such as certain types of shared libraries.

--defined-only
Display only defined symbols for each object file.

-g
--extern-only
Display only external symbols.

-l
--line-numbers
For each symbol, use debugging information to try to find a filename and line number. For a defined symbol, look for the line number of the address of the symbol. For an undefined symbol, look for the line number of a relocation entry which refers to the symbol. If line number information can be found, print it after the other symbol information.

-n
--numeric-sort
Sort symbols numerically by their addresses, rather than alphabetically by their names.

`-p`
`--no-sort`
Do not bother to sort the symbols in any order; print them in the order encountered.

`-r`
`--reverse-sort`
Reverse the order of the sort (whether numeric or alphabetic); let the last come first.

`-S`
`--print-size`
Print size, not the value, of defined symbols for the `bsd` output format.

`-s`
`--print-armap`
When listing symbols from archive members, include the index: a mapping (stored in the archive by the archiver or archive indexer (Section 4.7)) of which modules contain definitions for which names.

`--size-sort`
Sort symbols by size. The size is computed as the difference between the value of the symbol and the value of the symbol with the next higher value. If the `bsd` output format is used the size of the symbol is printed, rather than the value, and `-S` must be used in order both size and value to be printed.

`--special-syms`
Display symbols with target-specific special meanings. These symbols are generally used by the target for special processing, and are normally not helpful when included in normal symbol lists.

`-t` *`radix`*
`--radix` *`radix`*
Use *radix* as the radix for printing the symbol values. It must be `d` for decimal, `o` for octal, or `x` for hexadecimal.

`-u`
`--undefined-only`
Display only undefined symbols (those external to each object file).

`-V`
`--version`
Display version number of the symbols utility and exit.

`--help`
Display summary of the symbols utility command-line options.

4.5 Object file copier

```
hexagon-objcopy [-O bfdname|--output-target bfdname]
        [-S|--strip-all] [-g|--strip-debug]
        [-K symbolname|--keep-symbol symbolname]
        [-N symbolname|--strip-symbol symbolname]
        [--strip-unneeded-symbol symbolname]
        [-G symbolname|--keep-global-symbol symbolname]
        [-L symbolname|--localize-symbol symbolname]
        [--globalize-symbol (symbolname|file)]
        [-W symbolname|--weaken-symbol symbolname]
        [-w|--wildcard]
        [-x|--discard-all] [-X|--discard-locals]
        [-b num|--byte num]
        [-i interleave|--interleave interleave]
        [-j sectionname|--only-section sectionname]
        [-R sectionname|--remove-section sectionname]
        [-p|--preserve-dates]
        [--debugging]
        [--gap-fill val] [--pad-to address]
        [--set-start val] [--adjust-start incr]
        [--change-addresses incr]
        [--change-section-address section{=,+,-}val]
        [--change-section-lma section{=,+,-}val]
        [--change-section-vma section{=,+,-}val]
        [--change-warnings] [--no-change-warnings]
        [--set-section-flags section=flags]
        [--add-section sectionname=filename]
        [--rename-section oldname=newname[,flags]]
        [--change-leading-char ] [--remove-leading-char]
        [--srec-len ival ] [--srec-forceS3]
        [--redefine-sym old=new ]
        [--weaken]
        [--keep-symbols filename]
        [--strip-symbols filename]
        [--strip-unneeded-symbols filename]
        [--keep-global-symbols filename]
        [--localize-symbols filename]
        [--weaken-symbols filename]
        [--alt-machine-code index]
        [--prefix-symbols string]
        [--prefix-sections string]
        [--prefix-alloc-sections string]
        [--add-gnu-debuglink=filename]
        [--only-keep-debug]
        [--extract-symbol]
        [-v|--verbose]
        [-V|--version]
        [--help]
        infile [outfile]
```

The object file copier utility copies the contents of an object file to another. The copier utility uses the GNU BFD library (Section 3.5) to read and write the object files. It can write the destination object file in ELF, Intel hex, binary, or S-record format. The output format is controlled by the command-line option `-O`.

When the copier generates a destination object file in binary format, it essentially produces a memory dump of the contents of the input object file. All symbols and relocation information are discarded. The memory dump starts at the load address of the lowest section copied into the output file.

When generating an object file in binary or S-record formats, it may be helpful to use the `-S` option to remove sections containing debugging information. In some cases `-R` is useful to remove sections which contain information not needed by the binary file.

The copier command arguments *infile* and *outfile* specify the input and output files, respectively.

NOTE If you do not specify *outfile*, the copier creates a temporary file and destructively renames the result with the name of *infile*.

The copier options are described below.

`-O` *`bfdname`*
`--output-target` *`bfdname`*
Write the output file using the specified object file format, which can be `elf`, `ihex`, `binary`, `srec`, or `verilog` . The default is `elf`.

`-j` *`sectionname`*
`--only-section` *`sectionname`*
Copy only the named section from the input file to the output file. This option can be specified more than once. Note that using this option inappropriately may make the output file unusable.

`-R` *`sectionname`*
`--remove-section` *`sectionname`*
Remove any section named *sectionname* from the output file. This option can be specified more than once. Note that using this option inappropriately may make the output file unusable.

`-S`
`--strip-all`
Do not copy relocation and symbol information from the source file.

`-g`
`--strip-debug`
Do not copy debugging symbols from the source file.

`--strip-unneeded`
Strip all symbols that are not needed for relocation processing.

`-K` *`symbolname`*
`--keep-symbol` *`symbolname`*
Copy only symbol *symbolname* from the source file. This option can be specified more than once.

-N *symbolname*
--strip-symbol *symbolname*
Do not copy symbol *symbolname* from the source file. This option can be specified more than once.

--strip-unneeded-symbol *symbolname*
Do not copy symbol *symbolname* from the source file unless it is needed by a relocation. This option can be specified more than once.

-G *symbolname*
--keep-global-symbol *symbolname*
Keep only symbol *symbolname* global. Make all other symbols local to the file, so that they are not visible externally. This option can be specified more than once.

-L *symbolname*
--localize-symbol *symbolname*
Make symbol *symbolname* local to the file so it is not visible externally. This option can be specified more than once.

--globalize-symbol (*symbolname*|*file*)
Make symbol `symbolname` (or all the symbols `in` *`file`*) global in scope so it is visible outside the file it is defined in. This option can be specified more than once.

-W *symbolname*
--weaken-symbol *symbolname*
Make symbol *symbolname* weak. This option can be specified more than once.

-w
--wildcard
Enable the other command line options in the object file copier to use regular expressions in symbol names. The question mark (**?**), asterisk (`*`), backslash (\) and square bracket (`[]`) operators can be used anywhere in the symbol name. If the first character of the symbol name is the exclamation point (`!`) then the sense of the switch is reversed for that symbol. For example:

```
-w -W !foo -W fo*
```

This causes the object file copier to weaken all symbols that start with `fo`, except for the symbol `foo`.

-x
--discard-all
Do not copy non-global symbols from the source file.

-X
--discard-locals
Do not copy compiler-generated temporary symbols (Section 2.5.2).

-b *byte*
--byte *byte*
Keep only every *byte*th byte of the input file (header data is not affected). *byte* can be in the range from 0 to *interleave*-1, where *interleave* is specified by the `-i` or `--interleave` option, or the default of 4. This option is useful for creating files to program ROM. It is typically used with an `srec` output target.

`-i` *`interleave`*
`--interleave` *`interleave`*
: Only copy one out of every *interleave* bytes. Select which byte to copy with the `-b` or `--byte` option. The default is 4. the copier ignores this option if you do not specify either `-b` or `--byte`.

`-p`
`--preserve-dates`
: Set the access and modification dates of the output file to be the same as those of the input file.

`--debugging`
: Convert debugging information, if possible. This is not the default because only certain debugging formats are supported, and the conversion process can be time consuming.

`--gap-fill` *`val`*
: Fill gaps between sections with *val*. This operation applies to the *load address* (LMA) of the sections. It is done by increasing the size of the section with the lower address, and filling in the extra space created with *val*.

`--pad-to` *`address`*
: Pad the output file up to the load address *address*. This is done by increasing the size of the last section. The extra space is filled in with the value specified by `--gap-fill` (default zero).

`--set-start` *`val`*
: Set the start address of the new file to *val*. Not all object file formats support setting the start address.

`--change-start` *`incr`*
`--adjust-start` *`incr`*
: Change the start address by adding *incr*. Not all object file formats support setting the start address.

`--change-addresses` *`incr`*
`--adjust-vma` *`incr`*
: Change the VMA and LMA addresses of all sections, as well as the start address, by adding *incr*. Note that some object file formats do not permit section addresses to be changed arbitrarily.

NOTE This option does not relocate the sections. If the program expects sections to be loaded at a certain address, and this option is used to change the sections such that they are loaded at a different address, the program may fail.

`--change-section-address` *`section`*`{=,+,-}`*`val`*
`--adjust-section-vma` *`section`*`{=,+,-}`*`val`*
: Set or change both the VMA address and the LMA address of the specified section. If `=` is specified, the section address is set to *val*. Otherwise, *val* is added to or subtracted from the section address according to the specified + or -.

If the specified section does not exist in the input file a warning will be generated, unless `--no-change-warnings` is used.

For more information see `--change-addresses` above.

`--change-section-lma` *`section`*`{=,+,-}`*`val`*

Set or change LMA address of the specified section. The LMA address is the address where the section will be loaded into memory at program load time. If `=` is specified, the section address is set to *val*. Otherwise, *val* is added to or subtracted from the section address according to the specified operator (+ or -).

If the specified section does not exist in the input file a warning will be generated, unless `--no-change-warnings` is used.

For more information see `--change-addresses` above.

The default LMA address value is the offset between the LMA and VMA in the previous output section in the same region.

`--change-section-vma` *`section`*`{=,+,-}`*`val`*

Set or change VMA address of the specified section. The VMA address is the address where the section will be located once the program has started executing. If `=` is specified, the section address is set to *val*. Otherwise, *val* is added to or subtracted from the section address according to the specified operator (+ or -).

If the specified section does not exist in the input file a warning will be generated, unless `--no-change-warnings` is used.

For more information see `--change-addresses` above.

The default VMA address value is the offset between the LMA and VMA in the previous output section in the same region.

`--change-warnings`
`--adjust-warnings`

If `--change-section-address` or `--change-section-lma` or `--change-section-vma` is used, and the named section does not exist, issue a warning. This is the default.

`--no-change-warnings`
`--no-adjust-warnings`

Do not issue a warning if `--change-section-address` or `--adjust-section-lma` or `--adjust-section-vma` is used, even if the named section does not exist.

`--set-section-flags` *`section`*`=`*`flags`*

Set the flags for the named section. The *flags* argument is a comma separated string of flag names. The recognized names are `alloc`, `contents`, `load`, `noload`, `readonly`, `code`, `data`, `rom`, `share`, and `debug`. You can set the `contents` flag for a section which does not have contents, but it is not meaningful to clear the `contents` flag of a section which does have contents–just remove the section instead. Not all flags are meaningful for all object file formats.

`--add-section` *`sectionname`*`=`*`filename`*

Add a new section named *sectionname* while copying the file. The contents of the new section are taken from the file *filename*. The size of the section will be the size of the file. This option only works on file formats which can support sections with arbitrary names.

--rename-section *oldname*=*newname*[,*flags*]

Rename a section from *oldname* to *newname*, optionally changing the section's flags to *flags* in the process. This has the advantage over using a linker script to perform the rename in that the output stays as an object file and does not become a linked executable.

This option is particularly helpful when the input format is binary, since this will always create a section called `.data`. If for example, you wanted instead to create a section called `.rodata` containing binary data you could use the following command line to achieve it:

```
hexagon-objcopy -I binary -O output_format -B architecture \
--rename-section\
data=.rodata,alloc,load,readonly,data,contents \
input_binary_file output_object_file
```

--change-leading-char

Some object file formats use special characters at the start of symbols. The most common such character is underscore, which compilers often add before every symbol. This option tells the copier to change the leading character of every symbol when it converts between object file formats. If the object file formats use the same leading character, this option has no effect. Otherwise, it will add a character, or remove a character, or change a character, as appropriate.

--remove-leading-char

If the first character of a global symbol is a special symbol leading character used by the object file format, remove the character. The most common symbol leading character is underscore. This option will remove a leading underscore from all global symbols. This can be useful if you want to link together objects of different file formats with different conventions for symbol names. This is different from `--change-leading-char` because it always changes the symbol name when appropriate, regardless of the object file format of the output file.

--srec-len *ival*

Meaningful only for srec output. Set the maximum length of the Srecords being produced to *ival*. This length covers both address, data and crc fields.

--srec-forceS3

Meaningful only for srec output. Avoid generation of S1/S2 records, creating S3-only record format.

--redefine-sym *old*=*new*

Change the name of a symbol *old*, to *new*. This can be useful when one is trying link two things together for which you have no source, and there are name collisions.

--weaken

Change all global symbols in the file to be weak. This can be useful when building an object which will be linked against other objects using the `-R` option to the linker. This option is only effective when using an object file format which supports weak symbols.

`--keep-symbols` *`filename`*
: Apply `--keep-symbol` option to each symbol listed in the file *filename*. *filename* is simply a flat file, with one symbol name per line. Line comments can be introduced by the hash character. This option can be specified more than once.

`--strip-symbols` *`filename`*
: Apply `--strip-symbol` option to each symbol listed in the file *filename*. *filename* is simply a flat file, with one symbol name per line. Line comments can be introduced by the hash character. This option can be specified more than once.

`--strip-unneeded-symbols` *`filename`*
: Apply `--strip-unneeded-symbol` option to each symbol listed in the file *filename*. The file must be a flat file, with one symbol name per line. Line comments can be introduced by the hash character (#). This option can be specified more than once.

`--keep-global-symbols` *`filename`*
: Apply `--keep-global-symbol` option to each symbol listed in the file *filename*. *filename* is simply a flat file, with one symbol name per line. Line comments can be introduced by the hash character. This option can be specified more than once.

`--localize-symbols` *`filename`*
: Apply `--localize-symbol` option to each symbol listed in the file *filename*. *filename* is simply a flat file, with one symbol name per line. Line comments can be introduced by the hash character. This option can be specified more than once.

`--weaken-symbols` *`filename`*
: Apply `--weaken-symbol` option to each symbol listed in the file *filename*. *filename* is simply a flat file, with one symbol name per line. Line comments can be introduced by the hash character. This option can be specified more than once.

`--alt-machine-code` *`index`*
: If the output architecture has alternate machine codes, use the *index*th code instead of the default one. This is useful in case a machine is assigned an official code and the tool-chain adopts the new code, but other applications still depend on the original code being used.

`--prefix-symbols` *`string`*
: Prefix all symbols in the output file with *string*.

`--prefix-sections` *`string`*
: Prefix all section names in the output file with *string*.

`--prefix-alloc-sections` *`string`*
: Prefix all the names of all allocated sections in the output file with *string*.

`--add-gnu-debuglink=`*`filename`*
: Create a `.gnu_debuglink` section with a reference to the (presumably stripped) file *filename*, and add the section to the output file. This enables the debug information to be kept in a separate file.

`--only-keep-debug`

Strip a file, removing the contents of any sections that would not be stripped by `--strip-debug`, but leaving the debugging sections intact. All note sections are preserved in the output.

This option should be used in conjunction with the object copier utility to create a two-part executable: a small stripped binary that uses less memory space, and a larger file that includes debugging information. To create these files:

a. Link the executable as usual.

b. Run `objcopy --only-keep-debug` *`file file`*`.dbg` to create a file with the debug information (the `.dbg` extension is arbitrary).

c. Run `objcopy --strip-debug` *`file`* to create the stripped executable.

d. Run `objcopy --add-gnu-debuglink=`*`file`*`.dbg` *`file`* to add a link to the debugging info into the stripped executable.

Note that using `--only-keep-debug` is optional. Alternatively, the full file can be copied and processed with the command `strip --strip-debug`. The file specified by `--add-gnu-debuglink` can be the full executable.

NOTE Use this option only on fully-linked files.It does not make sense to use it on object files where the debugging information may be incomplete. Currently, the section `.gnu_debuglink` supports the presence of only one file with debug information, and not multiple filenames on a one-per-object-file basis.

`--extract-symbol`

Keep the file's section flags and symbols but remove all section data. Specifically, remove the contents of all sections, set the size of every section to zero, and set the file's start address to zero.

This option is used to build a `.sym` file for certain kernels. It is also useful for reducing the size of a `--just-symbols` linker input file.

`-V`
`--version`

Display version number of copier utility.

`-v`
`--verbose`

Verbose output: list all object files modified. In the case of archives, the command "`objcopy -V`" lists all members of the archive.

`--help`

Display summary of the copier command-line options.

4.6 Object file viewer

```
hexagon-objdump [-a|--archive-headers]
        [-C|--demangle [style] ]
        [-d|--disassemble]
        [-D|--disassemble-all]
        [-z|--disassemble-zeroes]
        [-EB|-EL|--endian {big|little }]
        [-f|--file-headers]
        [-F|--file-offsets]
        [--file-start-context]
        [-g|--debugging]
        [-e|--debugging-tags]
        [-h|--section-headers|--headers]
        [-j section|--section section]
        [-l|--line-numbers]
        [-S|--source]
        [--prefix=prefix]
        [--prefix-strip=level]
        [-M options|--disassembler-options options]
        [-p|--private-headers]
        [-r|--reloc]
        [-R|--dynamic-reloc]
        [-s|--full-contents]
         [-w[lLiaprmfFsoR]|--dwarf[(=rawline|=decodedline|
              =info|=abbrev|=pubnames|=aranges|=macro|=frames|
             =frames-interp|=str|=loc|=Ranges)...]
        [-G|--stabs]
        [-t|--syms]
        [-T|--dynamic-syms]
        [-x|--all-headers]
        [-w|--wide]
        [--start-address address]
        [--stop-address address]
        [--prefix-addresses]
        [--[no-]show-raw-insn]
        [--insn-width=width]
        [--adjust-vma offset]
        [--special-syms]
        [-V|--version]
        [-H|--help]
        objfile...
```

The object file viewer utility displays information about one or more object files. The options control what particular information to display. This information is mostly useful to programmers who are working on the compilation tools, as opposed to programmers who just want their program to compile and work.

The *objfile*... argument specifies the object files to be examined. When you specify archives, the viewer utility shows information on each of the member object files.

The object file viewer options are described below.

NOTE At least one of the following options must be specified: `-a,-d,-D,-f, -g,-G,-h,-H,-p,-r,-R,-S,-t,-T,-V,-x`.

-a
--archive-header
If any of the *objfile* files are archives, display the archive header information (in a format similar to `ls -l`). Besides the information you can list with the command "`hexagon-ar tv`", this option shows the object file format of each archive member.

--adjust-vma *offset*
When dumping information, first add *offset* to all the section addresses. This is useful if the section addresses do not correspond to the symbol table, which can happen when putting sections at particular addresses when using a format which can not represent section addresses, such as a.out.

-C
--demangle [*style*]
Decode (*demangle*) low-level symbol names into user-level names. Besides removing any initial underscore prepended by the system, this makes C++ function names readable. Different compilers have different mangling styles. The optional demangling style argument can be used to choose an appropriate demangling style for your compiler. See Section 4.11 for more information on demangling.

-g
--debugging
Display debugging information. This attempts to parse debugging information stored in the file and print it out using a C-like syntax. Only certain types of debugging information have been implemented. Some other types are supported by the command "`hexagon-readelf -w`". See Section 4.13.

-e
--debugging-tags
Equivalent to the `-g` option, but with the debugging information generated in a format compatible with the `ctags` tool.

-d
--disassemble
Display the assembler mnemonics for the machine instructions from *objfile*. This option only disassembles those sections which are expected to contain instructions.

-D
--disassemble-all
Like `-d`, but disassemble the contents of all sections, not just those expected to contain instructions.

--prefix-addresses
When disassembling, print the complete address on each line. This is the older disassembly format.

-EB
-EL
--endian {**big**|**little**}
Specify the endian-ness of the object files. This only affects disassembly. This can be useful when disassembling a file format which does not describe endian-ness information, such as S-records.

`-f`
`--file-headers`
Display summary information from the overall header of each of the *objfile* files.

`-F`
`--file-offsets`
When disassembling sections, whenever a symbol is displayed also display the file offset of the region of data about to be dumped. If zeroes are being skipped, then when disassembly resumes inform the user how many zeroes were skipped, and the file offset of the location from where the disassembly resumes. When dumping sections, display file offset of the location from where the dump starts.

`--file-start-context`
Specify that when displaying interlisted source code/disassembly (assumes `-S`) from a file that has not yet been displayed, extend the context to the start of the file.

`-h`
`--section-headers`
`--headers`
Display summary information from the section headers of the object file.

File segments can be relocated to nonstandard addresses, for example by using the `-Ttext`, `-Tdata`, or `-Tbss` options to the linker (Section 3.2.2). However, some object file formats, such as a.out, do not store the starting address of the file segments. In those situations, although the linker relocates the sections correctly, using the command "`objdump -h`" to list the file section headers cannot show the correct addresses. Instead, it shows the usual addresses, which are implicit for the target.

`-H`
`--help`
Display summary of viewer command-line options.

`-j` *`name`*
`--section` *`name`*
Display information only for section *name*.

`-l`
`--line-numbers`
Label the display (using debugging information) with the filename and source line numbers corresponding to the object code or relocs shown. Only useful with `-d`, `-D`, or `-r`.

`-M` *`options`*
`--disassembler-options` *`options`*
Pass target specific information to the disassembler. Only supported on some targets.

Print GPR (general-purpose register) names as appropriate for the specified ABI. By default, GPR names are selected according to the ABI of the binary being disassembled.

```
fpr-names=ABI
```

Print FPR (floating-point register) names as appropriate for the specified ABI. By default, FPR numbers are printed rather than names.

`cp0-names=`*`ARCH`*

Print CP0 (system control coprocessor; coprocessor 0) register names as appropriate for the CPU or architecture specified by *ARCH*. By default, CP0 register names are selected according to the architecture and CPU of the binary being disassembled.

`hwr-names=`*`ARCH`*

Print HWR (hardware register, used by the `rdhwr` instruction) names as appropriate for the CPU or architecture specified by *ARCH*. By default, HWR names are selected according to the architecture and CPU of the binary being disassembled.

`reg-names=`*`ABI`*

Print GPR and FPR names as appropriate for the selected ABI.

`reg-names=`*`ARCH`*

Print CPU-specific register names (CP0 register and HWR names) as appropriate for the selected CPU or architecture.

For any of the options listed above, *ABI* or *ARCH* can be specified as `numeric` to have numbers printed rather than names, for the selected types of registers. You can list the available values of *ABI* and *ARCH* using the `--help` option.

`-p`
`--private-headers`
Print information that is specific to the object file format. The exact information printed depends upon the object file format. For some object file formats, no additional information is printed.

`-r`
`--reloc`
Print the relocation entries of the file. If used with `-d` or `-D`, the relocations are printed interspersed with the disassembly.

`-R`
`--dynamic-reloc`
Print the dynamic relocation entries of the file. This is only meaningful for dynamic objects, such as certain types of shared libraries.

`-s`
`--full-contents`
Display the full contents of any sections requested.

`-W``[`**`lLiaprmfFsoR`**`]`
`--dwarf``[(=`**`rawline`**`|=`**`decodedline`**`|=`**`info`**`|`
`=`**`abbrev`**`|=`**`pubnames`**`|=`**`aranges`**`|=`**`macro`**`|=`**`frames`**`|`
`=`**`frames-interp`**`|=`**`str`**`|=`**`loc`**`|=`**`Ranges`**`)...]`
Display contents of the debug sections in the file, if any are present. If any of the optional keywords (for `--dwarf`) or corresponding letters (for `-W`) are specified, then dump only the data found in the specified sections.

`-S`
`--source`
Display source code intermixed with disassembly, if possible. Implies `-d`.

`--prefix=`*`prefix`*
Specify a prefix to prepend to the absolute paths when used with `-S`.

`--prefix-strip=`*`level`*
Indicate how many initial directory names to strip off hardwired absolute paths. This option has no effect without `--prefix=`*`prefix`*.

`--show-raw-insn`
When disassembling instructions, print the instruction in hex as well as in symbolic form. This is the default except when `--prefix-addresses` is used.

`--no-show-raw-insn`
When disassembling instructions, do not print the instruction bytes. This is the default when `--prefix-addresses` is used.

`--insn-width=`*`width`*
Specify number of bytes to display on a line when disassembling instructions.

`-G`
`--stabs`
Display the full contents of any sections requested. Display the contents of the `.stab`, `.stab.index`, and `.stab.excl` sections from an ELF file.

This option is useful only on systems in which the `.stab` debugging symbol-table entries are carried in an ELF section. In most other file formats, debugging symbol-table entries are interleaved with linkage symbols, and are visible in the `--syms` output.

`--start-address` *`address`*
Start displaying data at the specified address. This affects the output of the `-d`, `-r`, and `-s` options.

`--stop-address` *`address`*
Stop displaying data at the specified address. This affects the output of the `-d`, `-r`, and `-s` options.

`-t`
`--syms`
Print the symbol table entries of the file. This is similar to the information provided by the symbols utility program.

`-T`
`--dynamic-syms`
Print the dynamic symbol table entries of the file. This is only meaningful for dynamic objects, such as certain types of shared libraries. This is similar to the information provided by the symbols utility when given the `-D` (`--dynamic`) option.

`--special-syms`
Display symbols with target-specific special meanings. These symbols are generally used by the target for special processing, and are normally not helpful when included in normal symbol lists.

-V
--version
Print version number of the viewer.

-x
--all-headers
Display all available header information, including the symbol table and relocation entries. Using `-x` is equivalent to specifying all of `-a -f -h -r -t`.

-w
--wide
Format some lines for output devices that have more than 80 columns. Also do not truncate symbol names when they are displayed.

-z
--disassemble-zeroes
Normally the disassembly output will skip blocks of zeroes. This option directs the disassembler to disassemble those blocks, just like any other data.

4.7 Archive indexer

```
hexagon-ranlib [-vV] archive
```

The archive indexer utility generates an index to the contents of an archive and stores it in the archive. The index lists each symbol defined by a member of an archive that is a relocatable object file.

Use the command “`hexagon-nm -s`” or “`hexagon-nm --print-armap`” to list this index.

An archive with such an index speeds up linking to the library and allows routines in the library to call each other without regard to their placement in the archive.

NOTE The archive indexer is another form of the archiver (Section 4.3); running it is equivalent to executing “`hexagon-ar -s`”.

The archive indexer options are described below.

-v
-V
--version
Display version number of the archive indexer.

4.8 Object file size

```
hexagon-size
       [-A|-B|--format compatibility]
      [--help]
      [-d|-o|-x|--radix number]
      [-t|--totals]
      [-V|--version]
      [objfile...]
```

The object file size utility lists the section sizes – and the total size – for each of the object or archive files *objfile* in its argument list. By default, one line of output is generated for each object file or each module in an archive.

The *objfile*... argument specifies the object files to be examined. If none are specified, the file `a.out` will be used.

The object file size utility lists the following information:

- Total file size of the non-writable segments (`text`)
- Total file size of the writable segments (`data`)
- Total memory size of the writable segments minus total file size of the writable segments (`bss`)

The section names displayed by the object file size utility (`text`, `data`, `bss`) are not directly equivalent to the file sections described in Section 2.4. For instance, `text` can include the file sections `.text`, `.rodata`, `.tcm_code_cached`, etc. The displayed section sizes also don't include any alignment padding between the sections.

NOTE The displayed sizes will include any referenced libraries only if the specified object file has already been linked. In this case the included library sizes reflect only the functions used from the library file. However, if the library file is stored in an archive file (Section 4.3) then all the library modules in the archive will be included.

The object file size options are described below.

`-A`
`-B`
`--format` *`compatibility`*

Using one of these options, you can specify whether the output resembles the output from the equivalent size utility on System V (using `-A` or `--format sysv`) or Berkeley UNIX (using `-B` or `--format berkeley`). The default is the one-line format similar to Berkeley UNIX.

Here is an example of the Berkeley (default) format of output from the size utility:

```
$ size --format=Berkeley ranlib size
text    data    bss     dec     hex     filename
294880  81920   11592   388392  5ed28   ranlib
294880  81920   11888   388688  5ee50   size
```

Here is the same data, but displayed closer to System V conventions:

```
$ size --format=SysV ranlib size
ranlib  :
section         size         addr
.text          294880        8192
.data81920        303104
.bss 11592        385024
Total          388392


size  :
```

```
section          size         addr
.text          294880         8192
.data81920        303104
.bss 11888        385024
Total          388688
```

`--help`

Display summary of acceptable arguments and options.

`-d`
`-o`
`-x`
`--radix` *`number`*

Using one of these options, you can control whether the size of each section is specified in decimal (`-d`, or `--radix 10`); octal (`-o`, or `--radix 8`); or hexadecimal (`-x`, or `--radix 16`). In the `--radix` option, only the three values (8, 10, 16) are supported. The total size is always specified in two radices; decimal and hexadecimal for `-d` or `-x` output, or octal and hexadecimal if you're using `-o`.

`-t`
`--totals`

Display totals of all objects listed (Berkeley format listing mode only).

`-V`
`--version`

Display version number of the size utility.

4.9 Object file strings

```
hexagon-strings [-afov] [-min-len]
        [-n min-len] [--bytes min-len]
        [-t radix] [--radix radix]
        [-] [--all] [--print-file-name]
        [--help] [--version] file...
```

The object file strings utility prints the strings contained in the specified files. Strings are defined as sequences of printable characters that have the following properties:

- They are at least 4 characters long (or the number specified by a command option)
- They are followed by an unprintable character

By default, the strings utility only prints the strings from the initialized and loaded sections of object files; for other types of files, it prints the strings from the whole file.

The strings utility is mainly useful for determining the contents of non-text files.

The strings utility options are described below.

`-a`
`--all`
`-`

Do not scan only the initialized and loaded sections of object files; scan the whole files.

-f
--print-file-name
Print the name of the file before each string.

--help
Print a summary of the program usage on the standard output and exit.

-*min-len*
-n *min-len*
--bytes *min-len*
Print sequences of characters that are at least *min-len* characters long, instead of the default 4.

-o
Like the option "`-t o`". Other versions of the strings utility have `-o` act like `-t d` instead. Since we cannot be compatible with both ways, we simply chose one.

-t *radix*
--radix *radix*
Print the offset within the file before each string. The single character argument specifies the radix of the offset: `o` for octal, `x` for hexadecimal, or `d` for decimal.

-v
--version
Display version number of the strings utility.

4.10 Object file stripper

```
hexagon-strip [-s|--strip-all] [-S|-g|-d|--strip-debug]
      [-K symbolname |--keep-symbol symbolname ]
      [-N symbolname |--strip-symbol symbolname ]
      [-w|--wildcard]
      [-x|--discard-all ] [-X |--discard-locals]
      [-R sectionname |--remove-section sectionname ]
      [-o file ] [-p|--preserve-dates]
      [--only-keep-debug]
      [-v |--verbose] [-V|--version]
      [--help]
      objfile...
```

The object file stripper utility discards all symbols from object files *objfile*. The list of object files can include archives. At least one object file must be specified.

The stripper utility modifies the files named in its argument, rather than writing modified copies under different names.

The stripper utility options are listed below.

--help
Display summary of the stripper utility command-line options.

-R *sectionname*
--remove-section *sectionname*
Remove any section named *sectionname* from the output file. This option can be specified more than once. Note that using this option inappropriately may make the output file unusable.

`-s`
`--strip-all`
Remove all symbols.

`-g`
`-S`
`-d`
`--strip-debug`
Remove debugging symbols only.

`--strip-unneeded`
Remove all symbols that are not needed for relocation processing.

`-K` *`symbolname`*
`--keep-symbol` *`symbolname`*
Keep only symbol *symbolname* from the source file. This option can be specified more than once.

`-N` *`symbolname`*
`--strip-symbol` *`symbolname`*
Remove symbol *symbolname* from the source file. This option can be specified more than once, and can be combined with the command options other than `-K`.

`-w`
`--wildcard`
Enable the other command line options in the object file stripper to use regular expressions in symbol names. The question mark (**`?`**), asterisk (`*`), backslash (`\`) and square bracket (`[]`) operators can be used anywhere in the symbol name. If the first character of the symbol name is the exclamation point (`!`) then the sense of the switch is reversed for that symbol. For example:

```
-w -K !foo -K fo*
```

This causes the object file stripper to keep only the symbols that start with `fo`, except for the symbol `foo`.

`-o` *`file`*
Put the stripped output in *file*, rather than replacing the existing file. When this argument is used, only one *objfile* argument can be specified.

`-p`
`--preserve-dates`
Preserve the access and modification dates of the file.

`-x`
`--discard-all`
Remove non-global symbols.

`-X`
`--discard-locals`
Remove compiler-generated temporary symbols (Section 2.5.2)

`--only-keep-debug`

Strip a file, removing the contents of any sections that would not be stripped by `--strip-debug`, but leaving the debugging sections intact. All note sections are preserved in the output.

This option should be used in conjunction with the object copier utility to create a two-part executable: a small stripped binary that uses less memory space, and a larger file that includes debugging information. To create these files:

a. Link the executable as usual.

b. Run `objcopy --only-keep-debug` *`file file`*`.dbg` to create a file with the debug information (the `.dbg` extension is arbitrary).

c. Run `objcopy --strip-debug` *`file`* to create the stripped executable.

d. Run `objcopy --add-gnu-debuglink=`*`file`*`.dbg` *`file`* to add a link to the debugging info into the stripped executable.

Note that using `--only-keep-debug` is optional. Alternatively, the full file can be copied and processed with the command `strip --strip-debug`. The file specified by `--add-gnu-debuglink` can be the full executable.

NOTE Use this option only on fully-linked files.It does not make sense to use it on object files where the debugging information may be incomplete. Currently, the section `.gnu_debuglink` supports the presence of only one file with debug information, and not multiple filenames on a one-per-object-file basis.

`-V`
`--version`

Display version number for the stripper utility.

`-v`
`--verbose`

List all object files modified in the output. In the case of archives, the command "`strip -v`" lists all members of the archive.

4.11 C++ filter

```
hexagon-c++filt [-_|--strip-underscores]
        [-n|--no-strip-underscores]
         [-p|--no-params]
         [-t|--types]
         [-i|--no-verbose]
         [-s format |--format=format]
        [--help]  [--version]  [symbol...]
```

The C++ language supports *function overloading*, where multiple functions can be declared with the same name (provided that each function accepts parameters with different types). All C++ function names are specially encoded as low-level assembly labels (in a process known as *mangling*).

The C++ filter utility performs the inverse mapping: it decodes (*demangles*) low-level names into user-level names so the linker can keep the overloaded names from clashing.

Every alphanumeric word (consisting of letters, digits, underscores, dollars, or periods) seen in the input is a potential label. If the label decodes into a C++ name, the C++ name replaces the low-level name in the output.

You can use the C++ filter to decipher individual symbols:

```
hexagon-c++filt symbol
```

If no *symbol* arguments are specified, the C++ filter reads symbol names from the standard input and writes the demangled names to the standard output. All results are printed on the standard output.

The C++ filter options are described below.

-_
--strip-underscores
: On some systems, both the C and C++ compilers put an underscore in front of every name. For example, the C name `foo` gets the low-level name `_foo`. This option removes the initial underscore. Whether the C++ filter removes the underscore by default is target dependent.

-n
--no-strip-underscores
: Do not remove the initial underscore.

-p
--no-params
: When demangling function names, do not display the types of the function parameters.

-t
--types
: Attempt to demangle types as well as function names. By default this option is disabled because mangled types are normally only used internally in the compiler, and can be confused with non-mangled names. For instance, when treated as a mangled type name, the function `a` would be demangled to `signed char`.

`-i`
`--no-verbose`
Do not include implementation-specific details (if any) in the demangled output.

`-s` *`format`*
`--format=`*`format`*
Use demangling method performed by the specified compiler. *`format`* can be one of the following values:

`auto`

Automatic method selection based on executable format (default)

`gnu`

Demangling method used by GNU C++ compiler (g++)

`gnu-v3`

Demangling method used by GNU C++ compiler (g++) with V3+ ABI.

`--help`
Display summary of the C++ filter command options.

`--version`
Display version number of the C++ filter.

NOTE The C++ filter is a new utility, and the details of its user interface are subject to change in future releases. In particular, a command-line option may be required in the future to decode a name passed as an argument on the command line. In other words, the command:

```
hexagon-c++filt symbol
```

... may in a future release become:

```
hexagon-c++filt option symbol
```

4.12 Address converter

```
hexagon-addr2line [-e filename|--exe filename]
          [-C|--demangle]
          [-f|--functions] [-s|--basename]
          [-i|--inlines]
          [-H|--help] [-V|--version]
          [addr addr...]
```

The address converter utility translates program addresses into file names and line numbers. Given an address and an executable file, the converter utility uses the debugging information in the executable to determine which file name and line number are associated with a given address.

The executable file to use is specified with the `-e` option. The default file is `a.out`.

The converter has two modes of operation:

- In the first, hexadecimal addresses are specified on the command line, and the converter displays the file name and line number for each address.
- In the second, the converter reads hexadecimal addresses from standard input, and prints the file name and line number for each address on standard output. In this mode, the converter can be used in a pipe to convert dynamically chosen addresses.

The format of the output is `FILENAME:LINENO`. The file name and line number for each address is printed on a separate line. If the `-f` option is used, then each `FILENAME:LINENO` line is preceded by a `FUNCTIONNAME` line which is the name of the function containing the address.

If the file name or function name can not be determined, the converter will print two question marks in their place. If the line number can not be determined, it will print 0.

The address converter options are described below.

-e *filename*
--exe *filename*
Specify the name of the executable for which addresses should be translated. The default file is `a.out`.

-C
--demangle
Decode (*demangle*) low-level symbol names into user-level names. Besides removing any initial underscore prepended by the system, this makes C++ function names readable. See Section 4.11 for more information on demangling.

-f
--functions
Display function names as well as file and line number information.

-s
--basenames
Display only the base of each file name.

-i
--inlines
If an address belongs to a function that was inlined, also print the source information for all enclosing scopes back to the first non-inlined function. For example, if `main` inlines `callee1` which in turn inlines `callee2`, and the address is from `callee2`, the source information for `callee1` and `main` is printed along with the source information for `callee2`.

-H
--help
Display summary of the `addr2line` command options.

-v
--version
Display version number of `addr2line`.

4.13 ELF file viewer

```
hexagon-readelf [-a|--all]
        [-h|--file-header]
        [-l|--program-headers|--segments]
        [-S|--section-headers|--sections]
        [-g|--section-groups]
        [-t|--section-details]
        [-e|--headers]
        [-s|--syms|--symbols]
        [-n|--notes]
        [-r|--relocs]
        [-u|--unwind]
        [-d|--dynamic]
        [-V|--version-info]
        [-D|--use-dynamic]
         [-x section|--hex-dump=section]
        [-R section|--relocated-dump=section]
        [-p section|--string-dump=section]
         [-c|--archive-index]
         [-w[lLiaprmfFsoR]|--debug-dump[(=rawline|=decodedline|
             =info|=abbrev|=pubnames|=aranges|=macro|=frames|
            =frames-interp|=str|=loc|=Ranges)...]
        [-I|-histogram]
        [-v|--version]
        [-W|--wide]
        [-H|--help]
        elffile...
```

The ELF file viewer utility displays information about one or more ELF-format object files. The options control what particular information to display.

elffile... specifies one or more object files to be examined. Note that archive files containing ELF files can also be specified.

The ELF viewer options are described below.

NOTE At least one option besides `-v` or `-H` must be specified.

`-a`
`--all`
Equivalent to specifying `--file-header`, `--program-headers`, `--sections`, `--symbols`, `--relocs`, `--dynamic`, `--notes`, and `--version-info`.

`-h`
`--file-header`
Displays the information contained in the ELF header at the start of the file.

`-l`
`--program-headers`
`--segments`
Display information contained in the file segment headers, if they exist.

`-S`
`--sections`
`--section-headers`
Display information contained in the file section headers, if they exist.

`-g`
`--section-groups`
Display information contained in the file section groups, if they exist.

`-t`
`--section-details`
Display the detailed section information. Implies `-S`.

`-s`
`--symbols`
`--syms`
Display entries in symbol table section of the file, if it exists.

`-e`
`--headers`
Display all headers in the file. Equivalent to `-h -l -S`.

`-n`
`--notes`
Display contents of the NOTE segment, if it exists.

`-r`
`--relocs`
Display contents of the file’s relocation section, if it has one.

`-u`
`--unwind`
Display contents of the file’s unwind section, if it has one. Only the unwind sections for IA64 ELF files are currently supported.

`-d`
`--dynamic`
Display contents of the file’s dynamic section, if it has one.

`-V`
`--version-info`
Display contents of the version sections in the file, it they exist.

`-D`
`--use-dynamic`
When displaying symbols, this option makes the ELF viewer use the symbol table in the file’s dynamic section, rather than the one in the symbols section.

`-x` *`section`*
`--hex-dump=`*`section`*
Display the contents of the specified section as a hexadecimal dump. `section` can be either the index number for the section in the section table; or a name string that identifies all the sections with that name in the object file.

`-R` *`section`*
`--relocated-dump=`*`section`*
Display contents of the specified section as hexadecimal bytes, with the contents of the section being relocated before they are displayed. *`section`* can be either the index number for the section in the section table; or a name string which identifies all the sections with that name in the object file.

`-p` *`section`*
`--string-dump=`*`section`*
Display contents of the specified section as printable strings. *`section`* can be either the index number for the section in the section table; or a name string which identifies all the sections with that name in the object file.

`-c`
`--archive-index`
Display file symbol index information contained in the header part of the binary archives.

Note that this option performs the same function as the archiver’s `t` option (Section 4.3.1.1), but without using the BFD library.

`-w``[`**`lLiaprmfFsoR`**`]`
`--debug-dump``[(=`**`rawline`**`|=`**`decodedline`**`|=`**`info`**`|`
`=`**`abbrev`**`|=`**`pubnames`**`|=`**`aranges`**`|=`**`macro`**`|=`**`frames`**`|`
`=`**`frames-interp`**`|=`**`str`**`|=`**`loc`**`|=`**`Ranges`**`)...]`
Display contents of the debug sections in the file, if any are present. If any of the optional keywords (for `--debug-dump`) or corresponding letters (for `-w`) are specified, then dump only the data found in the specified sections.

`-I`
`--histogram`
Display a histogram of bucket list lengths when displaying the contents of the symbol tables.

`-v`
`--version`
Display version number of the ELF viewer.

`-W`
`--wide`

Don’t break output lines to fit into 80 columns. By default the ELF viewer breaks section header and segment listing lines for 64-bit ELF files so that they fit into 80 columns. This option causes the ELF viewer to print each section header resp. each segment one a single line, which is far more readable on terminals wider than 80 columns.

`-H`
`--help`

Display summary of the ELF viewer command options.

A Acknowledgments

A.1 Overview

This appendix lists the people who contributed to the development of the tools described in this document, and to the development of the documents on which this one is based. The following sections were copied from the acknowledgment sections in the original source documents.

A.2 Assembler

If you have contributed to GAS and your name isn't listed here, it is not meant as a slight. We just don't know about it. Send mail to the maintainer, and we'll correct the situation. Currently the maintainer is Ken Raeburn (email address raeburn@cygnus.com).

Dean Elsner wrote the original gnu assembler for the VAX.1

Jay Fenlason maintained GAS for a while, adding support for GDB-specific debug information and the 68k series machines, most of the preprocessing pass, and extensive changes in messages.c, input-file.c, write.c.

K. Richard Pixley maintained GAS for a while, adding various enhancements and many bug fixes, including merging support for several processors, breaking GAS up to handle multiple object file format back ends (including heavy rewrite, testing, an integration of the coff and b.out back ends), adding configuration including heavy testing and verification of cross assemblers and file splits and renaming, converted GAS to strictly ANSI C including full prototypes, added support for m680[34]0 and cpu32, did considerable work on i960 including a COFF port (including considerable amounts of reverse engineering), a SPARC opcode file rewrite, DECstation, rs6000, and hp300hpux host ports, updated "know" assertions and made them work, much other reorganization, cleanup, and lint.

Ken Raeburn wrote the high-level BFD interface code to replace most of the code in format-specific I/O modules.

The original VMS support was contributed by David L. Kashtan. Eric Youngdale has done much work with it since.

The Intel 80386 machine description was written by Eliot Dresselhaus.

Minh Tran-Le at IntelliCorp contributed some AIX 386 support.

The Motorola 88k machine description was contributed by Devon Bowen of Buffalo University and Torbjorn Granlund of the Swedish Institute of Computer Science.

Keith Knowles at the Open Software Foundation wrote the original MIPS back end (tc-mips.c, tc-mips.h), and contributed Rose format support (which hasn't been merged in yet). Ralph Campbell worked with the MIPS code to support a.out format.

Support for the Zilog Z8k and Renesas H8/300 and H8/500 processors (tc-z8k, tc-h8300, tc-h8500), and IEEE 695 object file format (obj-ieee), was written by Steve Chamberlain of Cygnus Support. Steve also modified the COFF back end to use BFD for some low-level operations, for use with the H8/300 and AMD 29k targets.

John Gilmore built the AMD 29000 support, added .include support, and simplified the configuration of which versions accept which directives. He updated the 68k machine description so that Motorola's opcodes always produced fixed-size instructions (e.g., jsr), while synthetic instructions remained shrinkable (jbsr). John fixed many bugs, including true tested cross-compilation support, and one bug in relaxation that took a week and required the proverbial one-bit fix.

Ian Lance Taylor of Cygnus Support merged the Motorola and MIT syntax for the 68k, completed support for some COFF targets (68k, i386 SVR3, and SCO UNIX), added support for MIPS ECOFF and ELF targets, wrote the initial RS/6000 and PowerPC assembler, and made a few other minor patches.

Steve Chamberlain made GAS able to generate listings.

Hewlett-Packard contributed support for the HP9000/300.

Jeff Law wrote GAS and BFD support for the native HPPA object format (SOM) along with a fairly extensive HPPA testsuite (for both SOM and ELF object formats). This work was supported by both the Center for Software Science at the University of Utah and Cygnus Support.

Support for ELF format files has been worked on by Mark Eichin of Cygnus Support (original, incomplete implementation for SPARC), Pete Hoogenboom and Jeff Law at the University of Utah (HPPA mainly), Michael Meissner of the Open Software Foundation (i386 mainly), and Ken Raeburn of Cygnus Support (sparc, and some initial 64-bit support).

Linas Vepstas added GAS support for the ESA/390 "IBM 370" architecture.

Richard Henderson rewrote the Alpha assembler. Klaus Kaempf wrote GAS and BFD support for openVMS/Alpha.

Timothy Wall, Michael Hayes, and Greg Smart contributed to the various tic* flavors.

David Heine, Sterling Augustine, Bob Wilson and John Ruttenberg from Tensilica, Inc. added support for Xtensa processors.

Several engineers at Cygnus Support have also provided many small bug fixes and configuration enhancements.

Many others have contributed large or small bugfixes and enhancements. If you have contributed significant work and are not mentioned on this list, and want to be, let us know. Some of the history has been lost; we are not intentionally leaving anyone out.

Dean Elsner, Jay Fenlason, and friends wrote the GNU assembler manual, with editing by Cygnus Support.

A.3 Linker

Steve Chamberlain and Cygnus Support wrote the GNU linker manual, with editing by Jeffrey Osier.

A.4 Utilities

Roland Pesch, Jeffrey Osier, and Cygnus Support wrote the GNU binutils manual.

B License Statements

B.1 History

Title	Year	Authors	Publisher	Network Location
Using as, the GNU assembler	1991, 1992, 1993, 1994, 1995, 1996, 1997, 1998	Dean Elsner, Jay Fenlason, and friends	Free Software Foundation, Inc.	http://www.gnu.org/software/binutils/manual/gas-2.9.1/as.html
Using ld, the GNU linker	1991, 1992, 1993, 1994, 1995, 1996, 1997, 1998	Steve Chamberlain, Cygnus Support		http://www.gnu.org/software/binutils/manual/ld-2.9.1/ld.html
The GNU Binary Utilities	1991, 1992, 1993, 1994, 1995, 1996, 1997, 1998, 2000, 2001, 2002	Roland Pesch, Jeffrey M. Osier, Cygnus Support		http://ftp.gnu.org/gnu/binutils/binutils-2.14.tar.gz
Hexagon Binutils User Guide, A GNU Manual	2006, 2007, 2008, 2009, 2010	Roland H. Pesch, Steve Chamberlain, Jay Fenlason, Jeffrey M. Osier, Dean Elsner, Cygnus Support, QUALCOMM Incorporated	QUALCOMM Incorporated	Source files supplied directly to users
Hexagon Binutils, A GNU Manual	2010	Roland H. Pesch, Steve Chamberlain, Jay Fenlason, Jeffrey M. Osier, Dean Elsner, Cygnus Support, QUALCOMM Incorporated, Qualcomm Innovation Center, Inc.	Code Aurora Forum	Source files supplied directly to users

B.2 Free software

GDB is free software, protected by the GNU General Public License (GPL). The GPL gives you the freedom to copy or adapt a licensed program—but every person getting a copy also gets with it the freedom to modify that copy (which means that they must get access to the source code), and the freedom to distribute further copies. Typical software companies use copyrights to limit your freedoms; the Free Software Foundation uses the GPL to preserve these freedoms.

Fundamentally, the General Public License is a license which says that you have these freedoms and that you cannot take these freedoms away from anyone else.

B.3 Free software needs free documentation

The biggest deficiency in the free software community today is not in the software—it is the lack of good free documentation that we can include with the free software. Many of our most important programs do not come with free reference manuals and free introductory texts. Documentation is an essential part of any software package; when an important free software package does not come with a free manual and a free tutorial, that is a major gap. We have many such gaps today.

Consider Perl, for instance. The tutorial manuals that people normally use are non-free. How did this come about? Because the authors of those manuals published them with restrictive terms—no copying, no modification, source files not available—which exclude them from the free software world.

That wasn't the first time this sort of thing happened, and it was far from the last. Many times we have heard a GNU user eagerly describe a manual that he is writing, his intended contribution to the community, only to learn that he had ruined everything by signing a publication contract to make it non-free.

Free documentation, like free software, is a matter of freedom, not price. The problem with the non-free manual is not that publishers charge a price for printed copies—that in itself is fine. (The Free Software Foundation sells printed copies of manuals, too.) The problem is the restrictions on the use of the manual. Free manuals are available in source code form, and give you permission to copy and modify. Non-free manuals do not allow this.

The criteria of freedom for a free manual are roughly the same as for free software. Redistribution (including the normal kinds of commercial redistribution) must be permitted, so that the manual can accompany every copy of the program, both on-line and on paper.

Permission for modification of the technical content is crucial too. When people modify the software, adding or changing features, if they are conscientious they will change the manual too—so they can provide accurate and clear documentation for the modified program. A manual that leaves you no choice but to write a new manual to document a changed version of the program is not really available to our community.

Some kinds of limits on the way modification is handled are acceptable. For example, requirements to preserve the original author's copyright notice, the distribution terms, or the list of authors, are ok. It is also no problem to require modified versions to include notice that they were modified. Even entire sections that may not be deleted or changed are acceptable, as long as they deal with nontechnical topics (like this one). These kinds of restrictions are acceptable because they don't obstruct the community's normal use of the manual.

However, it must be possible to modify all the technical content of the manual, and then distribute the result in all the usual media, through all the usual channels. Otherwise, the restrictions obstruct the use of the manual, it is not free, and we need another manual to replace it.

Please spread the word about this issue. Our community continues to lose manuals to proprietary publishing. If we spread the word that free software needs free reference manuals and free tutorials, perhaps the next person who wants to contribute by writing documentation will realize, before it is too late, that only free manuals contribute to the free software community.

If you are writing documentation, please insist on publishing it under the GNU Free Documentation License or another free documentation license. Remember that this decision requires your approval—you don't have to let the publisher decide. Some commercial publishers will use a free license if you insist, but they will not propose the option; it is up to you to raise the issue and say firmly that this is what you want. If the publisher you are dealing with refuses, please try other publishers. If you're not sure whether a proposed license is free, write to licensing@gnu.org.

You can encourage commercial publishers to sell more free, copylefted manuals and tutorials by buying them, and particularly by buying copies from the publishers that paid for their writing or for major improvements. Meanwhile, try to avoid buying non-free documentation at all. Check the distribution terms of a manual before you buy it, and insist that whoever seeks your business must respect your freedom. Check the history of the book, and try to reward the publishers that have paid or pay the authors to work on it.

- The Free Software Foundation maintains a list of free documentation published by other publishers, at http://www.fsf.org/doc/other-free-books.html.

B.4 GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

```
Copyright © 1989, 1991 Free Software Foundation, Inc.
59 Temple Place - Suite 330, Boston, MA 02111-1307, USA

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.
```

B.4.1 Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

B.4.2 How to apply these terms to your new programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the “copyright” line and a pointer to where the full notice is found.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) year  name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
type `show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type `show c' for details.
```

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w` and `show c`; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a “copyright disclaimer” for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the
program
`Gnomovision' (which makes passes at compilers) written by James
Hacker.
```

signature of Ty Coon, `1 April 1989`
`Ty Coon, President of Vice`

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

B.5 GNU FREE DOCUMENTATION LICENSE

Version 1.2, November 2002

```
Copyright © 2000,2001,2002 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA  02111-1307, USA

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.
```

PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The “Document”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “you”. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A “Modified Version” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “Secondary Section” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The “Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not “Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

A section “Entitled XYZ” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “Acknowledgements”, “Dedications”, “Endorsements”, or “History”.) To “Preserve the Title” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

B.5.1 ADDENDUM: How to use this license for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
  Copyright (C)  year  your name.
  Permission is granted to copy, distribute and/or modify this
document
  under the terms of the GNU Free Documentation License, Version
1.2
  or any later version published by the Free Software Foundation;
  with no Invariant Sections, no Front-Cover Texts, and no Back-
Cover
  Texts.  A copy of the license is included in the section entitled
``GNU
  Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the “with...Texts.” line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Index

- 162
- 180
-_ 184
:*phdr* 133
? 163
-(*archive* ... -) 106
(*type*) 132
@*file* 106
@*filename* 21
=*fill* 134
>*region* 133
a 158
A 162
-A 163
-a 163
-a 174
-A 179
-a 180
-a 188
-a[lhsncdg][=*filename*] 22
-a=*filename* 22
ABSOLUTE(*exp*) 115
--add-gnu-debuglink=*filename* 171
ADDLIB *archive* (*module*, *module*, ... *module*) 160
ADDLIB *archive* 160
ADDMOD *member*, *member*, ... *member* 160
--add-needed 85
ADDR(*section*) 115
address 131
--add-section *sectionname*=*filename* 169
--adjust-section-vma *section*{=,+,-}*val* 168
--adjust-start *incr* 168
--adjust-vma *incr* 168
--adjust-vma *offset* 174
--adjust-warnings 169
-a*keyword* 85
ALIGN(*align*) 115
ALIGN(*exp*, *align*) 115
ALIGN(*section_align*) 133
ALIGNOF(*section*) 116
--all 180
--all 188
--all-headers 178
--allow-multiple-definition 85
--allow-shlib-undefined 85
--alternate 20
--alt-machine-code *index* 171
--archive-header 174
--archive-index 189
AS_NEEDED (*file*, *file*, ...) 141
--as-needed 85
ASSEMBLER-INTERNAL-LOGIC-ERROR! 41
ASSERT (*expr*, *message*) 141
AT (*lma*) 132
AT>*lma_region* 132
attr 121
--auxiliary *name* 90
b 158
B 162
-B 163
-B 179
-b *byte* 167
-b *input-format* 86
--basenames 186
-Bdynamic 86
-Bgroup 86
-Bshareable 86
-Bstatic 86
-Bsymbolic 87
-Bsymbolic-functions 87
--build-id[=*style*] 87
--byte *byte* 167
--bytes *min-len* 181
c 158
C 162
-C 163
-C 174
-C 186
-c 189
-call_shared 86
--change-addresses *incr* 168
--change-leading-char 170
--change-section-address *section*{=,+,-}*val* 168
--change-section-lma *section*{=,+,-}*val* 169
--change-section-vma *section*{=,+,-}*val* 169
--change-start *incr* 168
--change-warnings 169
--check-sections 87
CLEAR 160
constraint 133
CREATE *archive* 160
--cref 87
d 156
D 162
-D 163
-D 174
-d 174
-d 180
-d 182
-d 188
-D 189
-D 23

-d 88
DATA_SEGMENT_ALIGN(*maxpagesize*, *commonpagesize*) 116
DATA_SEGMENT_END(*exp*) 116
DATA_SEGMENT_RELRO_END(*offset*, *exp*) 117
-dc 88
--debugging 168
--debugging 174
--debugging-tags 174
--debug-syms 163
--default-script *file* 88
DEFINED(*symbol*) 117
--defined-only 163
--defsym *name*=*value* 25
--defsym *symbol*=*expression* 88
DELETE *module*, *module*, ... *module* 160
--demangle [*style*] 163
--demangle [*style*] 174
--demangle 186
--demangle[=*style*] 88
DIRECTORY archive (*module*, ... *module*) 160
DIRECTORY archive (*module*, ... *module*) *output-file* 160
--disable-new-dtags 88
--disassemble 174
--disassemble-all 174
--disassembler-options *options* 175
--disassemble-zeroes 178
--discard-all 104
--discard-all 167
--discard-all 182
--discard-locals 104
--discard-locals 167
--discard-locals 182
-dn 86
-dp 88
-dT *file* 88
-dy 86
--dynamic 163
--dynamic 188
--dynamic-linker *file* 88
--dynamic-list *file* 89
--dynamic-list-cpp-new 89
--dynamic-list-cpp-typeinfo 89
--dynamic-list-data 89
--dynamic-reloc 176
--dynamic-syms 177
-e 174
-e 188
-E 89
-e *entry* 89
-e *filename* 186
-EB 174
-EB 90
--eh-frame-hdr 90
-EL 174
-EL 90
--emit-relocs 97
--enable-new-dtags 90
END 160
--endian {big|little} 174
--entry *entry* 89
--error-unresolved-symbols 90
--exclude-libs *lib* [,*lib*] 89
--exe *filename* 186
--export-dynamic 89
expr section 41
EXTERN (*symbol symbol* ...) 141
--extern-only 163
EXTRACT *module*, *module*, ... *module* 160
--extract-symbol 172
f 158
-F 175
-f 175
-f 181
-f 186
-f 25
-f 90
-F *name* 90
--fatal-warnings 23
--fatal-warnings 91
--file-header 188
--file-headers 175
--file-offsets 175
files 150
--file-start-context 175
--filter *name* 90
FORCE_COMMON_ALLOCATION 141
--force-dynamic 91
--force-exe-suffix 91
--format *compatibility* 179
--format *input-format* 86
--format=*format* 185
--full-contents 176
--functions 186
G 162
-g 163
-g 166
-g 174
-G 177
-g 182
-g 188
-g 91
-G *size* 24
-G *size* 91
-G *symbolname* 167
--gap-fill *val* 168
--gc-sections 92
--gdwarf2 24
--globalize-symbol (*symbolname*|*file*) 167
--gpsize *size* 91
GROUP (*file file* ...) 142
GROUP (*file*, *file*, ...) 142
--gstabs 24
--gstabs+ 24
-H 175
-h 175
-H 187
-h 188
-H 190
-h 20
-h *name* 92

--hash-size=*number* 92
--headers 175
--headers 188
--help 164
--help 172
--help 175
--help 180
--help 181
--help 181
--help 185
--help 187
--help 190
--help 20
--help 92
--hex-dump=*section* 189
--histogram 189
I 162
-i 185
-i 187
-I 189
-i 92
-i *interleave* 168
-I *pathname* 21
INCLUDE *file* 142
INHIBIT_COMMON_ALLOCATION 142
--inlines 187
INPUT (*file file* **...)** 142
INPUT (*file, file,* **...)** 142
INSERT [AFTER | BEFORE] *output_section* 142
--insn-width=*width* 177
--interleave *interleave* 168
-j *name* 175
-j *sectionname* 166
--just-symbols *file* 97
-K *symbolname* 166
-K *symbolname* 182
--keep-global-symbol *symbolname* 167
--keep-global-symbols *filename* 171
--keep-locals 25
--keep-symbol *symbolname* 166
--keep-symbol *symbolname* 182
--keep-symbols *filename* 171
-l 163
-l 175
-l 188
-L 25
-L *symbolname* 167
-l[:]*archive* 93
len 122
LENGTH(*memory***)** 117
--library [:]*archive* 93
--library-path *searchdir* 93
line numbers 151
--line-numbers 163
--line-numbers 175
LIST 161
--listing-cont-lines *lines* 23
--listing-lhs-width *width* 23
--listing-lhs-width2 *width* 23
--listing-rhs-width *width* 23
LOADADDR(*section***)** 117
--localize-symbol *symbolname* 167
--localize-symbols *filename* 171
-L*searchdir* 93
m 156
-M 94
-M *options* 175
-Map *mapfile* 94
-march *archname* 25
-march *archname* 94
MAX(*exp1, exp2***)** 117
-mcpu *archname* 25
-mcpu *archname* 94
--MD *filename* 21
-m*emulation* 93
--mfalign-info 20
--mfalign-warn 23
MIN(*exp1, exp2***)** 117
-min-len 181
--mno-extender 24
--mno-jumps 24
--mno-jumps-long 24
--mno-pairing 24
--mno-pairing-branch 24
--mno-pairing-duplex 24
--mpairing-info 24
-mv2 25
-mv2 94
-mv3 25
-mv3 94
-mv4 25
-mv4 94
N 158
N 162
-n 163
-n 184
-n 188
-N 94
-n 94
-n *min-len* 181
-N *symbolname* 167
-N *symbolname* 182
name 121
NEXT(*exp***)** 117
--nmagic 94
--no-add-needed 94
--no-adjust-warnings 169
--no-allow-shlib-undefined 94
--no-as-needed 95
--no-change-warnings 169
--no-check-sections 95
NOCROSSREFS (*section section* **...)** 143
--no-define-common 95
--no-demangle 163
--no-demangle 95
--no-export-dynamic 95
--no-fatal-warnings 95
--no-gc-sections 95
--noinhibit-exec 96
--no-keep-memory 95
-non_shared 86
--no-omagic 95

--no-params 184
--no-print-gc-sections 95
--no-show-raw-insn 177
--no-sort 164
-nostdlib 96
--no-strip-underscores 184
--notes 188
--no-undefined 96
--no-undefined-version 96
--no-verbose 185
--no-warn 23
--no-warn-mismatch 96
--no-whole-archive 96
--numeric-sort 163
o 158
-o 163
-o 180
-o 181
-O *bfdname* 166
-o *file* 182
-o *filename* 21
-O *level* 96
-o *output* 96
--oformat *output-format* 97
--omagic 94
--only-keep-debug 172
--only-keep-debug 183
--only-section *sectionname* 166
OPEN *archive* 161
origin 122
ORIGIN(*memory***)** 117
OUTPUT (*file*) 143
--output *output* 96
OUTPUT_ARCH (*bfdname*) 143
OUTPUT_FORMAT (*bfdname*) 143
--output-target *bfdname* 166
p 156
P 158
-p 164
-p 168
-p 176
-p 182
-p 184
-p ***section*** 189
--pad-to *address* 168
--prefix=*prefix* 177
--prefix-addresses 174
--prefix-alloc-sections *string* 171
--prefix-sections *string* 171
--prefix-strip=*level* 177
--prefix-symbols *string* 171
--preserve-dates 168
--preserve-dates 182
--print-armap 164
--print-file-name 163
--print-file-name 181
--print-gc-sections 97
--print-map 94
--print-size 164
--private-headers 176
--program-headers 188
q 157
-q 97
r 157
R 162
-r 164
-R 176
-r 176
-r 188
-R 24
-r 97
-R *file* 97
-R ***section*** 189
-R *sectionname* 166
-R *sectionname* 181
--radix *number* 180
--radix *radix* 164
--radix *radix* 181
--redefine-sym *old=new* 170
--reduce-memory-overheads 25
--reduce-memory-overheads 97
REGION_ALIAS (*alias*, *region*) 143
--reloc 176
--relocateable 97
--relocated-dump=*section* 189
relocation level 150
--relocs 188
--remove-leading-char 170
--remove-section *sectionname* 166
--remove-section *sectionname* 181
--rename-section *oldname=newname*[,*flags*] 170
REPLACE *module*, *module*, *...* *module* 161
--retain-symbols-file *file* 98
--reverse-sort 164
-rpath *dir* 98
-rpath-link *dir* 98
S 158
s 158
s 162
-S 164
-s 164
-S 166
-s 176
-S 177
-S 182
-s 182
-s 186
-S 188
-s 188
-S 99
-s 99
-s *format* 185
SAVE 161
--script *commandfile* 100
SEARCH_DIR (*path*) 143
--section *name* 175
--section-details 188
--section-groups 188
--section-headers 175
--section-headers 188
sections 150
--sections 188

--section-start *section=org* 99
SEGMENT_START(*segment, default***)** 117
--segments 188
--set-section-flags *section=flags* 169
--set-start *val* 168
-shared 86
--show-raw-insn 177
SIZEOF_HEADERS 118
sizeof_headers 118
SIZEOF(*section***)** 118
--size-sort 164
-soname *name* 92
--sort-common[=(ascending|descending)] 99
--sort-section=(name|alignment) 99
--source 177
--special-syms 164
--special-syms 177
--split-by-file [*size*] 99
--split-by-reloc [*count*] 99
--srec-forceS3 170
--srec-len *ival* 170
--stabs 177
--start-address *address* 177
--start-group *archive* **... --end-group** 106
STARTUP (*file* **)** 143
-static 86
--statistics 20
--stats 99
--stop-address *address* 177
--string-dump=*section* 189
--strip-all 166
--strip-all 182
--strip-all 99
--strip-debug 166
--strip-debug 182
--strip-debug 99
--strip-local-absolute 25
--strip-symbol *symbolname* 167
--strip-symbol *symbolname* 182
--strip-symbols *filename* 171
--strip-underscores 184
--strip-unneeded 166
--strip-unneeded 182
--strip-unneeded-symbol *symbolname* 167
--strip-unneeded-symbols *filename* 171
SUBALIGN(*subsection_align***)** 133
symbols 150
--symbols 188
--syms 177
--syms 188
--sysroot=*directory* 99
t 157
T 159
T 162
-T 177
-t 177
-t 180
-t 184
-t 188
-t 99
-T *commandfile* 100
-t *radix* 164
-t *radix* 181
TARGET (*format* **)** 144
--target-help 100
--target-help 20
-Tbss *org* 100
-tcm 100
-Tdata *org* 100
--totals 180
--trace 99
--trace-symbol *symbol* 104
--traditional-format 100
--trampolines[=(yes|no)] 100
-Ttext *org* 100
-Ttext-segment=*addr* 100
--types 184
u 159
U 162
-u 164
-u 188
-u *symbol* 100
--undefined *symbol* 100
--undefined-only 164
--unique[=*section***]** 101
--unresolved-symbols=*method* 101
--unwind 188
-Ur 101
--use-dynamic 189
-V 101
-v 101
V 159
v 159
V 162
-V 164
-V 172
-v 172
-V 178
-V 178
-v 178
-V 180
-v 181
-V 183
-v 183
-v 187
-V 189
-v 189
-v 20
--verbose | --dll-verbose 102
VERBOSE 161
--verbose 172
--verbose 183
--version 101
--version 164
--version 172
--version 178
--version 178
--version 180
--version 181
--version 183
--version 185
--version 187

--version 189
--version 20
--version-info 189
--version-script *file* 102
W 162
-w 167
-w 178
-w 182
-W 190
-W 23
-W *symbolname* 167
-W[lLiaprmfFsoR] 176
-w[lLiaprmfFsoR] 189
--warn 23
--warn-alternate-em 102
--warn-common 102
--warn-once 103
--warn-section-align 103
--warn-shared-textrel 103
--warn-unresolved-symbols 102
--weaken 170
--weaken-symbol *symbolname* 167
--weaken-symbols *filename* 171
--whole-archive 103
--wide 178
--wide 190
--wildcard 167
--wildcard 182
--wrap *symbol* 104
-X 104
-x 104
x 157
-X 167
-x 167
-x 178
-x 180
-X 182
-x 182
-x ***section*** 189
-Y *path* 104
-y *symbol* 104
-z 178
-Z 24
-z *keyword* 104

Published by Code Aurora Forum.