

1 This is Info file ld.info, produced by Makeinfo version 1.68 from the
2 input file
3 /usr/src/gnu/usr.bin/binutils/doc/../../../../../contrib/binutils/ld/ld.texinfo.
4

5 START-INFO-DIR-ENTRY
6 * Ld: (ld). The GNU linker.
7 END-INFO-DIR-ENTRY
8

9 This file documents the GNU linker LD.
10

11 Copyright (C) 1991, 92, 93, 94, 95, 96, 97, 1998 Free Software
12 Foundation, Inc.
13

14 Permission is granted to make and distribute verbatim copies of this
15 manual provided the copyright notice and this permission notice are
16 preserved on all copies.
17

18 Permission is granted to copy and distribute modified versions of
19 this manual under the conditions for verbatim copying, provided also
20 that the entire resulting derived work is distributed under the terms
21 of a permission notice identical to this one.
22

23 Permission is granted to copy and distribute translations of this
24 manual into another language, under the above conditions for modified
25 versions.
26

27 **US**

28 File: ld.info, Node: Top, Next: Overview, Up: (dir)
29

30 Using ld
31 *****
32

33 This file documents the GNU linker ld.
34

35 * Menu:
36

37 * Overview:: Overview
38 * Invocation:: Invocation
39 * Commands:: Command Language
40
41 * Machine Dependent:: Machine Dependent Features
42
43 * BFD:: BFD
44
45 * Reporting Bugs:: Reporting Bugs
46 * MRI:: MRI Compatible Script Files
47 * Index:: Index
48

49 **US**

50 File: ld.info, Node: Overview, Next: Invocation, Prev: Top, Up: Top
51

52 Overview
53 *****
54

55 `ld' combines a number of object and archive files, relocates their
56 data and ties up symbol references. Usually the last step in compiling
57 a program is to run `ld'.
58

59 `ld' accepts Linker Command Language files written in a superset of
60 AT&T's Link Editor Command Language syntax, to provide explicit and
61 total control over the linking process.
62

63 This version of `ld' uses the general purpose BFD libraries to
64 operate on object files. This allows `ld' to read, combine, and write
65 object files in many different formats--for example, COFF or `a.out'.
66 Different formats may be linked together to produce any available kind

Amazon Ex. 1015
IPR Petition - USP 7,784,058

```

67 of object file. *Note BFD::, for more information.
68
69 Aside from its flexibility, the GNU linker is more helpful than other
70 linkers in providing diagnostic information. Many linkers abandon
71 execution immediately upon encountering an error; whenever possible,
72 `ld' continues executing, allowing you to identify other errors (or, in
73 some cases, to get an output file in spite of the error).
74
75 US
76 File: ld.info, Node: Invocation, Next: Commands, Prev: Overview, Up: Top
77
78 Invocation
79 *****
80
81 The GNU linker `ld' is meant to cover a broad range of situations,
82 and to be as compatible as possible with other linkers. As a result,
83 you have many choices to control its behavior.
84
85 * Menu:
86
87 * Options:: Command Line Options
88 * Environment:: Environment Variables
89
90 US
91 File: ld.info, Node: Options, Next: Environment, Up: Invocation
92
93 Command Line Options
94 =====
95
96 The linker supports a plethora of command-line options, but in actual
97 practice few of them are used in any particular context. For instance,
98 a frequent use of `ld' is to link standard Unix object files on a
99 standard, supported Unix system. On such a system, to link a file
100 `hello.o':
101
102 ld -o OUTPUT /lib/crt0.o hello.o -lc
103
104 This tells `ld' to produce a file called OUTPUT as the result of
105 linking the file `/lib/crt0.o' with `hello.o' and the library `libc.a',
106 which will come from the standard search directories. (See the
107 discussion of the `-l' option below.)
108
109 The command-line options to `ld' may be specified in any order, and
110 may be repeated at will. Repeating most options with a different
111 argument will either have no further effect, or override prior
112 occurrences (those further to the left on the command line) of that
113 option. Options which may be meaningfully specified more than once are
114 noted in the descriptions below.
115
116 Non-option arguments are objects files which are to be linked
117 together. They may follow, precede, or be mixed in with command-line
118 options, except that an object file argument may not be placed between
119 an option and its argument.
120
121 Usually the linker is invoked with at least one object file, but you
122 can specify other forms of binary input files using `-l', `-R', and the
123 script command language. If *no* binary input files at all are
124 specified, the linker does not produce any output, and issues the
125 message `No input files'.
126
127 If the linker can not recognize the format of an object file, it will
128 assume that it is a linker script. A script specified in this way
129 augments the main linker script used for the link (either the default
130 linker script or the one specified by using `-T'). This feature
131 permits the linker to link against a file which appears to be an object
132 or an archive, but actually merely defines some symbol values, or uses

```

```

133 `INPUT' or `GROUP' to load other objects. Note that specifying a
134 script in this way should only be used to augment the main linker
135 script; if you want to use some command that logically can only appear
136 once, such as the `SECTIONS' or `MEMORY' command, you must replace the
137 default linker script using the `-T' option. *Note Commands::.
138
139 For options whose names are a single letter, option arguments must
140 either follow the option letter without intervening whitespace, or be
141 given as separate arguments immediately following the option that
142 requires them.
143
144 For options whose names are multiple letters, either one dash or two
145 can precede the option name; for example, `--oformat' and `-oformat'
146 are equivalent. Arguments to multiple-letter options must either be
147 separated from the option name by an equals sign, or be given as
148 separate arguments immediately following the option that requires them.
149 For example, `--oformat srec' and `--oformat=srec' are equivalent.
150 Unique abbreviations of the names of multiple-letter options are
151 accepted.
152
153 `-aKEYWORD'
154     This option is supported for HP/UX compatibility. The KEYWORD
155     argument must be one of the strings `archive', `shared', or
156     `default'. `-aarchive' is functionally equivalent to `-Bstatic',
157     and the other two keywords are functionally equivalent to
158     `-Bdynamic'. This option may be used any number of times.
159
160 `--AARCHITECTURE'
161 `--architecture=ARCHITECTURE'
162     In the current release of `ld', this option is useful only for the
163     Intel 960 family of architectures. In that `ld' configuration, the
164     ARCHITECTURE argument identifies the particular architecture in
165     the 960 family, enabling some safeguards and modifying the
166     archive-library search path. *Note `ld' and the Intel 960 family:
167     i960, for details.
168
169     Future releases of `ld' may support similar functionality for
170     other architecture families.
171
172 `--b INPUT-FORMAT'
173 `--format=INPUT-FORMAT'
174     `ld' may be configured to support more than one kind of object
175     file. If your `ld' is configured this way, you can use the `-b'
176     option to specify the binary format for input object files that
177     follow this option on the command line. Even when `ld' is
178     configured to support alternative object formats, you don't
179     usually need to specify this, as `ld' should be configured to
180     expect as a default input format the most usual format on each
181     machine. INPUT-FORMAT is a text string, the name of a particular
182     format supported by the BFD libraries. (You can list the
183     available binary formats with `objdump -i'.) *Note BFD::.
184
185     You may want to use this option if you are linking files with an
186     unusual binary format. You can also use `-b' to switch formats
187     explicitly (when linking object files of different formats), by
188     including `--b INPUT-FORMAT' before each group of object files in a
189     particular format.
190
191     The default format is taken from the environment variable
192     `GNUTARGET'. *Note Environment::. You can also define the input
193     format from a script, using the command `TARGET'; see *Note Option
194     Commands::.
195
196 `--c MRI-COMMANDFILE'
197 `--mri-script=MRI-COMMANDFILE'
198     For compatibility with linkers produced by MRI, `ld' accepts script

```

```

199     files written in an alternate, restricted command language,
200     described in *Note MRI Compatible Script Files: MRI.  Introduce
201     MRI script files with the option '-c'; use the '-T' option to run
202     linker scripts written in the general-purpose 'ld' scripting
203     language.  If MRI-CMDFILE does not exist, 'ld' looks for it in the
204     directories specified by any '-L' options.
205
206     '-d'
207     '-dc'
208     '-dp'
209     These three options are equivalent; multiple forms are supported
210     for compatibility with other linkers.  They assign space to common
211     symbols even if a relocatable output file is specified (with
212     '-r').  The script command 'FORCE_COMMON_ALLOCATION' has the same
213     effect.  *Note Option Commands::.
214
215     '-e ENTRY'
216     '--entry=ENTRY'
217     Use ENTRY as the explicit symbol for beginning execution of your
218     program, rather than the default entry point.  *Note Entry Point::,
219     for a discussion of defaults and other ways of specifying the
220     entry point.
221
222     '-E'
223     '--export-dynamic'
224     When creating a dynamically linked executable, add all symbols to
225     the dynamic symbol table.  The dynamic symbol table is the set of
226     symbols which are visible from dynamic objects at run time.
227
228     If you do not use this option, the dynamic symbol table will
229     normally contain only those symbols which are referenced by some
230     dynamic object mentioned in the link.
231
232     If you use 'dlopen' to load a dynamic object which needs to refer
233     back to the symbols defined by the program, rather than some other
234     dynamic object, then you will probably need to use this option when
235     linking the program itself.
236
237     '-f'
238     '--auxiliary NAME'
239     When creating an ELF shared object, set the internal DT_AUXILIARY
240     field to the specified name.  This tells the dynamic linker that
241     the symbol table of the shared object should be used as an
242     auxiliary filter on the symbol table of the shared object NAME.
243
244     If you later link a program against this filter object, then, when
245     you run the program, the dynamic linker will see the DT_AUXILIARY
246     field.  If the dynamic linker resolves any symbols from the filter
247     object, it will first check whether there is a definition in the
248     shared object NAME.  If there is one, it will be used instead of
249     the definition in the filter object.  The shared object NAME need
250     not exist.  Thus the shared object NAME may be used to provide an
251     alternative implementation of certain functions, perhaps for
252     debugging or for machine specific performance.
253
254     This option may be specified more than once.  The DT_AUXILIARY
255     entries will be created in the order in which they appear on the
256     command line.
257
258     '-F NAME'
259     '--filter NAME'
260     When creating an ELF shared object, set the internal DT_FILTER
261     field to the specified name.  This tells the dynamic linker that
262     the symbol table of the shared object which is being created
263     should be used as a filter on the symbol table of the shared
264     object NAME.

```

```

265
266     If you later link a program against this filter object, then, when
267     you run the program, the dynamic linker will see the DT_FILTER
268     field. The dynamic linker will resolve symbols according to the
269     symbol table of the filter object as usual, but it will actually
270     link to the definitions found in the shared object NAME. Thus the
271     filter object can be used to select a subset of the symbols
272     provided by the object NAME.
273
274     Some older linkers used the '-F' option throughout a compilation
275     toolchain for specifying object-file format for both input and
276     output object files. The GNU linker uses other mechanisms for this
277     purpose: the '-b', '--format', '--oformat' options, the 'TARGET'
278     command in linker scripts, and the 'GNUTARGET' environment
279     variable. The GNU linker will ignore the '-F' option when not
280     creating an ELF shared object.
281
282     '--force-exe-suffix'
283         Make sure that an output file has a .exe suffix.
284
285         If a successfully built fully linked output file does not have a
286         '.exe' or '.dll' suffix, this option forces the linker to copy the
287         output file to one of the same name with a '.exe' suffix. This
288         option is useful when using unmodified Unix makefiles on a
289         Microsoft Windows host, since some versions of Windows won't run
290         an image unless it ends in a '.exe' suffix.
291
292     '-g'
293         Ignored. Provided for compatibility with other tools.
294
295     '-GVALUE'
296     '--gpsize=VALUE'
297         Set the maximum size of objects to be optimized using the GP
298         register to SIZE. This is only meaningful for object file formats
299         such as MIPS ECOFF which supports putting large and small objects
300         into different sections. This is ignored for other object file
301         formats.
302
303     '-hNAME'
304     '-soname=NAME'
305         When creating an ELF shared object, set the internal DT_SONAME
306         field to the specified name. When an executable is linked with a
307         shared object which has a DT_SONAME field, then when the
308         executable is run the dynamic linker will attempt to load the
309         shared object specified by the DT_SONAME field rather than the
310         using the file name given to the linker.
311
312     '-i'
313         Perform an incremental link (same as option '-r').
314
315     '-lARCHIVE'
316     '--library=ARCHIVE'
317         Add archive file ARCHIVE to the list of files to link. This
318         option may be used any number of times. 'ld' will search its
319         path-list for occurrences of 'libARCHIVE.a' for every ARCHIVE
320         specified.
321
322         On systems which support shared libraries, 'ld' may also search for
323         libraries with extensions other than '.a'. Specifically, on ELF
324         and SunOS systems, 'ld' will search a directory for a library with
325         an extension of '.so' before searching for one with an extension of
326         '.a'. By convention, a '.so' extension indicates a shared library.
327
328         The linker will search an archive only once, at the location where
329         it is specified on the command line. If the archive defines a
330         symbol which was undefined in some object which appeared before

```

```

331     the archive on the command line, the linker will include the
332     appropriate file(s) from the archive.  However, an undefined
333     symbol in an object appearing later on the command line will not
334     cause the linker to search the archive again.
335
336     See the `-' option for a way to force the linker to search
337     archives multiple times.
338
339     You may list the same archive multiple times on the command line.
340
341     This type of archive searching is standard for Unix linkers.
342     However, if you are using `ld' on AIX, note that it is different
343     from the behaviour of the AIX linker.
344
345     `-LSEARCHDIR'
346     `--library-path=SEARCHDIR'
347         Add path SEARCHDIR to the list of paths that `ld' will search for
348         archive libraries and `ld' control scripts.  You may use this
349         option any number of times.  The directories are searched in the
350         order in which they are specified on the command line.
351         Directories specified on the command line are searched before the
352         default directories.  All `-L' options apply to all `-l' options,
353         regardless of the order in which the options appear.
354
355         The default set of paths searched (without being specified with
356         `-L') depends on which emulation mode `ld' is using, and in some
357         cases also on how it was configured.  *Note Environment::.
358
359         The paths can also be specified in a link script with the
360         `SEARCH_DIR' command.  Directories specified this way are searched
361         at the point in which the linker script appears in the command
362         line.
363
364     `-mEMULATION'
365         Emulate the EMULATION linker.  You can list the available
366         emulations with the `--verbose' or `-V' options.
367
368         If the `-m' option is not used, the emulation is taken from the
369         `LDEMULATION' environment variable, if that is defined.
370
371         Otherwise, the default emulation depends upon how the linker was
372         configured.
373
374     `-M'
375     `--print-map'
376         Print a link map to the standard output.  A link map provides
377         information about the link, including the following:
378
379         * Where object files and symbols are mapped into memory.
380
381         * How common symbols are allocated.
382
383         * All archive members included in the link, with a mention of
384         the symbol which caused the archive member to be brought in.
385
386     `-n'
387     `--nmagic'
388         Set the text segment to be read only, and mark the output as
389         `NMAGIC' if possible.
390
391     `-N'
392     `--omagic'
393         Set the text and data sections to be readable and writable.  Also,
394         do not page-align the data segment.  If the output format supports
395         Unix style magic numbers, mark the output as `OMAGIC'.
396

```

```

397  '-o OUTPUT'
398  '--output=OUTPUT'
399      Use OUTPUT as the name for the program produced by 'ld'; if this
400      option is not specified, the name 'a.out' is used by default. The
401      script command 'OUTPUT' can also specify the output file name.
402
403  '-r'
404  '--relocateable'
405      Generate relocatable output--i.e., generate an output file that
406      can in turn serve as input to 'ld'. This is often called "partial
407      linking". As a side effect, in environments that support standard
408      Unix magic numbers, this option also sets the output file's magic
409      number to 'OMAGIC'. If this option is not specified, an absolute
410      file is produced. When linking C++ programs, this option *will
411      not* resolve references to constructors; to do that, use '-Ur'.
412
413      This option does the same thing as '-i'.
414
415  '-R FILENAME'
416  '--just-symbols=FILENAME'
417      Read symbol names and their addresses from FILENAME, but do not
418      relocate it or include it in the output. This allows your output
419      file to refer symbolically to absolute locations of memory defined
420      in other programs. You may use this option more than once.
421
422      For compatibility with other ELF linkers, if the '-R' option is
423      followed by a directory name, rather than a file name, it is
424      treated as the '-rpath' option.
425
426  '-s'
427  '--strip-all'
428      Omit all symbol information from the output file.
429
430  '-S'
431  '--strip-debug'
432      Omit debugger symbol information (but not all symbols) from the
433      output file.
434
435  '-t'
436  '--trace'
437      Print the names of the input files as 'ld' processes them.
438
439  '-T COMMANDFILE'
440  '--script=COMMANDFILE'
441      Read link commands from the file COMMANDFILE. These commands
442      replace 'ld''s default link script (rather than adding to it), so
443      COMMANDFILE must specify everything necessary to describe the
444      target format. You must use this option if you want to use a
445      command which can only appear once in a linker script, such as the
446      'SECTIONS' or 'MEMORY' command. *Note Commands::. If COMMANDFILE
447      does not exist, 'ld' looks for it in the directories specified by
448      any preceding '-L' options. Multiple '-T' options accumulate.
449
450  '-u SYMBOL'
451  '--undefined=SYMBOL'
452      Force SYMBOL to be entered in the output file as an undefined
453      symbol. Doing this may, for example, trigger linking of
454      additional modules from standard libraries. '-u' may be repeated
455      with different option arguments to enter additional undefined
456      symbols.
457
458  '-v'
459  '--version'
460  '-V'
461      Display the version number for 'ld'. The '-V' option also lists
462      the supported emulations.

```

```

463
464 '-x'
465 '--discard-all'
466     Delete all local symbols.
467
468 '-X'
469 '--discard-locals'
470     Delete all temporary local symbols.  For most targets, this is all
471     local symbols whose names begin with 'L'.
472
473 '-y SYMBOL'
474 '--trace-symbol=SYMBOL'
475     Print the name of each linked file in which SYMBOL appears.  This
476     option may be given any number of times.  On many systems it is
477     necessary to prepend an underscore.
478
479     This option is useful when you have an undefined symbol in your
480     link but don't know where the reference is coming from.
481
482 '-Y PATH'
483     Add PATH to the default library search path.  This option exists
484     for Solaris compatibility.
485
486 '-z KEYWORD'
487     This option is ignored for Solaris compatibility.
488
489 '-( ARCHIVES -) '
490 '--start-group ARCHIVES --end-group'
491     The ARCHIVES should be a list of archive files.  They may be
492     either explicit file names, or '-l' options.
493
494     The specified archives are searched repeatedly until no new
495     undefined references are created.  Normally, an archive is
496     searched only once in the order that it is specified on the
497     command line.  If a symbol in that archive is needed to resolve an
498     undefined symbol referred to by an object in an archive that
499     appears later on the command line, the linker would not be able to
500     resolve that reference.  By grouping the archives, they all be
501     searched repeatedly until all possible references are resolved.
502
503     Using this option has a significant performance cost.  It is best
504     to use it only when there are unavoidable circular references
505     between two or more archives.
506
507 '-assert KEYWORD'
508     This option is ignored for SunOS compatibility.
509
510 '-Bdynamic'
511 '-dy'
512 '-call_shared'
513     Link against dynamic libraries.  This is only meaningful on
514     platforms for which shared libraries are supported.  This option
515     is normally the default on such platforms.  The different variants
516     of this option are for compatibility with various systems.  You
517     may use this option multiple times on the command line: it affects
518     library searching for '-l' options which follow it.
519
520 '-Bstatic'
521 '-dn'
522 '-non_shared'
523 '-static'
524     Do not link against shared libraries.  This is only meaningful on
525     platforms for which shared libraries are supported.  The different
526     variants of this option are for compatibility with various
527     systems.  You may use this option multiple times on the command
528     line: it affects library searching for '-l' options which follow

```

```

529         it.
530
531     '-Bsymbolic'
532         When creating a shared library, bind references to global symbols
533         to the definition within the shared library, if any. Normally, it
534         is possible for a program linked against a shared library to
535         override the definition within the shared library. This option is
536         only meaningful on ELF platforms which support shared libraries.
537
538     '--cref'
539         Output a cross reference table. If a linker map file is being
540         generated, the cross reference table is printed to the map file.
541         Otherwise, it is printed on the standard output.
542
543         The format of the table is intentionally simple, so that it may be
544         easily processed by a script if necessary. The symbols are
545         printed out, sorted by name. For each symbol, a list of file
546         names is given. If the symbol is defined, the first file listed
547         is the location of the definition. The remaining files contain
548         references to the symbol.
549
550     '--defsym SYMBOL=EXPRESSION'
551         Create a global symbol in the output file, containing the absolute
552         address given by EXPRESSION. You may use this option as many
553         times as necessary to define multiple symbols in the command line.
554         A limited form of arithmetic is supported for the EXPRESSION in
555         this context: you may give a hexadecimal constant or the name of
556         an existing symbol, or use '+' and '-' to add or subtract
557         hexadecimal constants or symbols. If you need more elaborate
558         expressions, consider using the linker command language from a
559         script (*note Assignment: Symbol Definitions: Assignment.).
560         *Note:* there should be no white space between SYMBOL, the equals
561         sign ("<="), and EXPRESSION.
562
563     '--dynamic-linker FILE'
564         Set the name of the dynamic linker. This is only meaningful when
565         generating dynamically linked ELF executables. The default dynamic
566         linker is normally correct; don't use this unless you know what
567         you are doing.
568
569     '-EB'
570         Link big-endian objects. This affects the default output format.
571
572     '-EL'
573         Link little-endian objects. This affects the default output
574         format.
575
576     '--embedded-relocs'
577         This option is only meaningful when linking MIPS embedded PIC code,
578         generated by the -membedded-pic option to the GNU compiler and
579         assembler. It causes the linker to create a table which may be
580         used at runtime to relocate any data which was statically
581         initialized to pointer values. See the code in testsuite/ld-empic
582         for details.
583
584     '--help'
585         Print a summary of the command-line options on the standard output
586         and exit.
587
588     '-Map MAPFILE'
589         Print a link map to the file MAPFILE. See the description of the
590         '-M' option, above.
591
592     '--no-keep-memory'
593         'ld' normally optimizes for speed over memory usage by caching the
594         symbol tables of input files in memory. This option tells 'ld' to

```

```

595         instead optimize for memory usage, by rereading the symbol tables
596         as necessary. This may be required if `ld' runs out of memory
597         space while linking a large executable.
598
599     `--no-warn-mismatch'
600         Normally `ld' will give an error if you try to link together input
601         files that are mismatched for some reason, perhaps because they
602         have been compiled for different processors or for different
603         endiannesses. This option tells `ld' that it should silently
604         permit such possible errors. This option should only be used with
605         care, in cases when you have taken some special action that
606         ensures that the linker errors are inappropriate.
607
608     `--no-whole-archive'
609         Turn off the effect of the `--whole-archive' option for subsequent
610         archive files.
611
612     `--noinhibit-exec'
613         Retain the executable output file whenever it is still usable.
614         Normally, the linker will not produce an output file if it
615         encounters errors during the link process; it exits without
616         writing an output file when it issues any error whatsoever.
617
618     `--oformat OUTPUT-FORMAT'
619         `ld' may be configured to support more than one kind of object
620         file. If your `ld' is configured this way, you can use the
621         `--oformat' option to specify the binary format for the output
622         object file. Even when `ld' is configured to support alternative
623         object formats, you don't usually need to specify this, as `ld'
624         should be configured to produce as a default output format the most
625         usual format on each machine. OUTPUT-FORMAT is a text string, the
626         name of a particular format supported by the BFD libraries. (You
627         can list the available binary formats with `objdump -i'.) The
628         script command `OUTPUT_FORMAT' can also specify the output format,
629         but this option overrides it. *Note BFD::.
630
631     `--qmagic'
632         This option is ignored for Linux compatibility.
633
634     `--Qy'
635         This option is ignored for SVR4 compatibility.
636
637     `--relax'
638         An option with machine dependent effects. This option is only
639         supported on a few targets. *Note `ld' and the H8/300: H8/300.
640         *Note `ld' and the Intel 960 family: i960.
641
642         On some platforms, the `--relax' option performs global
643         optimizations that become possible when the linker resolves
644         addressing in the program, such as relaxing address modes and
645         synthesizing new instructions in the output object file.
646
647         On platforms where this is not supported, `--relax' is accepted,
648         but ignored.
649
650     `--retain-symbols-file FILENAME'
651         Retain *only* the symbols listed in the file FILENAME, discarding
652         all others. FILENAME is simply a flat file, with one symbol name
653         per line. This option is especially useful in environments (such
654         as VxWorks) where a large global symbol table is accumulated
655         gradually, to conserve run-time memory.
656
657         `--retain-symbols-file' does *not* discard undefined symbols, or
658         symbols needed for relocations.
659
660         You may only specify `--retain-symbols-file' once in the command

```

```

661         line. It overrides '-s' and '-S'.
662
663     '-rpath DIR'
664         Add a directory to the runtime library search path. This is used
665         when linking an ELF executable with shared objects. All '-rpath'
666         arguments are concatenated and passed to the runtime linker, which
667         uses them to locate shared objects at runtime. The '-rpath'
668         option is also used when locating shared objects which are needed
669         by shared objects explicitly included in the link; see the
670         description of the '-rpath-link' option. If '-rpath' is not used
671         when linking an ELF executable, the contents of the environment
672         variable 'LD_RUN_PATH' will be used if it is defined.
673
674         The '-rpath' option may also be used on SunOS. By default, on
675         SunOS, the linker will form a runtime search patch out of all the
676         '-L' options it is given. If a '-rpath' option is used, the
677         runtime search path will be formed exclusively using the '-rpath'
678         options, ignoring the '-L' options. This can be useful when using
679         gcc, which adds many '-L' options which may be on NFS mounted
680         filesystems.
681
682         For compatibility with other ELF linkers, if the '-R' option is
683         followed by a directory name, rather than a file name, it is
684         treated as the '-rpath' option.
685
686     '-rpath-link DIR'
687         When using ELF or SunOS, one shared library may require another.
688         This happens when an 'ld -shared' link includes a shared library
689         as one of the input files.
690
691         When the linker encounters such a dependency when doing a
692         non-shared, non-relocateable link, it will automatically try to
693         locate the required shared library and include it in the link, if
694         it is not included explicitly. In such a case, the '-rpath-link'
695         option specifies the first set of directories to search. The
696         '-rpath-link' option may specify a sequence of directory names
697         either by specifying a list of names separated by colons, or by
698         appearing multiple times.
699
700         The linker uses the following search paths to locate required
701         shared libraries.
702         1. Any directories specified by '-rpath-link' options.
703
704         2. Any directories specified by '-rpath' options. The difference
705            between '-rpath' and '-rpath-link' is that directories
706            specified by '-rpath' options are included in the executable
707            and used at runtime, whereas the '-rpath-link' option is only
708            effective at link time.
709
710         3. On an ELF system, if the '-rpath' and '-rpath-link' options
711            were not used, search the contents of the environment variable
712            'LD_RUN_PATH'.
713
714         4. On SunOS, if the '-rpath' option was not used, search any
715            directories specified using '-L' options.
716
717         5. For a native linker, the contents of the environment variable
718            'LD_LIBRARY_PATH'.
719
720         6. The default directories, normally '/lib' and '/usr/lib'.
721
722         If the required shared library is not found, the linker will issue
723         a warning and continue with the link.
724
725     '-shared'
726     '-Bshareable'

```

```

727     Create a shared library.  This is currently only supported on ELF,
728     XCOFF and SunOS platforms.  On SunOS, the linker will
729     automatically create a shared library if the '-e' option is not
730     used and there are undefined symbols in the link.
731
732     '--sort-common'
733     This option tells 'ld' to sort the common symbols by size when it
734     places them in the appropriate output sections.  First come all
735     the one byte symbols, then all the two bytes, then all the four
736     bytes, and then everything else.  This is to prevent gaps between
737     symbols due to alignment constraints.
738
739     '--split-by-file'
740     Similar to '--split-by-reloc' but creates a new output section for
741     each input file.
742
743     '--split-by-reloc COUNT'
744     Tries to creates extra sections in the output file so that no single
745     output section in the file contains more than COUNT relocations.
746     This is useful when generating huge relocatable for downloading
747     into certain real time kernels with the COFF object file format;
748     since COFF cannot represent more than 65535 relocations in a
749     single section.  Note that this will fail to work with object file
750     formats which do not support arbitrary sections.  The linker will
751     not split up individual input sections for redistribution, so if a
752     single input section contains more than COUNT relocations one
753     output section will contain that many relocations.
754
755     '--stats'
756     Compute and display statistics about the operation of the linker,
757     such as execution time and memory usage.
758
759     '--traditional-format'
760     For some targets, the output of 'ld' is different in some ways from
761     the output of some existing linker.  This switch requests 'ld' to
762     use the traditional format instead.
763
764     For example, on SunOS, 'ld' combines duplicate entries in the
765     symbol string table.  This can reduce the size of an output file
766     with full debugging information by over 30 percent.
767     Unfortunately, the SunOS 'dbx' program can not read the resulting
768     program ('gdb' has no trouble).  The '--traditional-format' switch
769     tells 'ld' to not combine duplicate entries.
770
771     '-Tbss ORG'
772     '-Tdata ORG'
773     '-Ttext ORG'
774     Use ORG as the starting address for--respectively--the 'bss',
775     'data', or the 'text' segment of the output file.  ORG must be a
776     single hexadecimal integer; for compatibility with other linkers,
777     you may omit the leading '0x' usually associated with hexadecimal
778     values.
779
780     '-Ur'
781     For anything other than C++ programs, this option is equivalent to
782     '-r': it generates relocatable output--i.e., an output file that
783     can in turn serve as input to 'ld'.  When linking C++ programs,
784     '-Ur' *does* resolve references to constructors, unlike '-r'.  It
785     does not work to use '-Ur' on files that were themselves linked
786     with '-Ur'; once the constructor table has been built, it cannot
787     be added to.  Use '-Ur' only for the last partial link, and '-r'
788     for the others.
789
790     '--verbose'
791     Display the version number for 'ld' and list the linker emulations
792     supported.  Display which input files can and cannot be opened.

```

```

793     Display the linker script if using a default builtin script.
794
795     '--version-script=VERSION-SCRIPTFILE'
796     Specify the name of a version script to the linker. This is
797     typically used when creating shared libraries to specify
798     additional information about the version heirarchy for the library
799     being created. This option is only meaningful on ELF platforms
800     which support shared libraries. *Note Version Script::.
801
802     '--warn-common'
803     Warn when a common symbol is combined with another common symbol
804     or with a symbol definition. Unix linkers allow this somewhat
805     sloppy practice, but linkers on some other operating systems do
806     not. This option allows you to find potential problems from
807     combining global symbols. Unfortunately, some C libraries use
808     this practice, so you may get some warnings about symbols in the
809     libraries as well as in your programs.
810
811     There are three kinds of global symbols, illustrated here by C
812     examples:
813
814     'int i = 1;'
815         A definition, which goes in the initialized data section of
816         the output file.
817
818     'extern int i;'
819         An undefined reference, which does not allocate space. There
820         must be either a definition or a common symbol for the
821         variable somewhere.
822
823     'int i;'
824         A common symbol. If there are only (one or more) common
825         symbols for a variable, it goes in the uninitialized data
826         area of the output file. The linker merges multiple common
827         symbols for the same variable into a single symbol. If they
828         are of different sizes, it picks the largest size. The
829         linker turns a common symbol into a declaration, if there is
830         a definition of the same variable.
831
832     The '--warn-common' option can produce five kinds of warnings.
833     Each warning consists of a pair of lines: the first describes the
834     symbol just encountered, and the second describes the previous
835     symbol encountered with the same name. One or both of the two
836     symbols will be a common symbol.
837
838     1. Turning a common symbol into a reference, because there is
839     already a definition for the symbol.
840         FILE(SECTION): warning: common of `SYMBOL'
841         overridden by definition
842         FILE(SECTION): warning: defined here
843
844     2. Turning a common symbol into a reference, because a later
845     definition for the symbol is encountered. This is the same
846     as the previous case, except that the symbols are encountered
847     in a different order.
848         FILE(SECTION): warning: definition of `SYMBOL'
849         overriding common
850         FILE(SECTION): warning: common is here
851
852     3. Merging a common symbol with a previous same-sized common
853     symbol.
854         FILE(SECTION): warning: multiple common
855         of `SYMBOL'
856         FILE(SECTION): warning: previous common is here
857
858     4. Merging a common symbol with a previous larger common symbol.

```

```

859         FILE(SECTION): warning: common of `SYMBOL'
860         overridden by larger common
861         FILE(SECTION): warning: larger common is here
862
863     5. Merging a common symbol with a previous smaller common
864        symbol. This is the same as the previous case, except that
865        the symbols are encountered in a different order.
866        FILE(SECTION): warning: common of `SYMBOL'
867        overriding smaller common
868        FILE(SECTION): warning: smaller common is here
869
870 `--warn-constructors'
871     Warn if any global constructors are used. This is only useful for
872     a few object file formats. For formats like COFF or ELF, the
873     linker can not detect the use of global constructors.
874
875 `--warn-multiple-gp'
876     Warn if multiple global pointer values are required in the output
877     file. This is only meaningful for certain processors, such as the
878     Alpha. Specifically, some processors put large-valued constants
879     in a special section. A special register (the global pointer)
880     points into the middle of this section, so that constants can be
881     loaded efficiently via a base-register relative addressing mode.
882     Since the offset in base-register relative mode is fixed and
883     relatively small (e.g., 16 bits), this limits the maximum size of
884     the constant pool. Thus, in large programs, it is often necessary
885     to use multiple global pointer values in order to be able to
886     address all possible constants. This option causes a warning to
887     be issued whenever this case occurs.
888
889 `--warn-once'
890     Only warn once for each undefined symbol, rather than once per
891     module which refers to it.
892
893 `--warn-section-align'
894     Warn if the address of an output section is changed because of
895     alignment. Typically, the alignment will be set by an input
896     section. The address will only be changed if it not explicitly
897     specified; that is, if the `SECTIONS' command does not specify a
898     start address for the section (*note SECTIONS::.).
899
900 `--whole-archive'
901     For each archive mentioned on the command line after the
902     `--whole-archive' option, include every object file in the archive
903     in the link, rather than searching the archive for the required
904     object files. This is normally used to turn an archive file into
905     a shared library, forcing every object to be included in the
906     resulting shared library. This option may be used more than once.
907
908 `--wrap SYMBOL'
909     Use a wrapper function for SYMBOL. Any undefined reference to
910     SYMBOL will be resolved to `__wrap_SYMBOL'. Any undefined
911     reference to `__real_SYMBOL' will be resolved to SYMBOL.
912
913     This can be used to provide a wrapper for a system function. The
914     wrapper function should be called `__wrap_SYMBOL'. If it wishes
915     to call the system function, it should call `__real_SYMBOL'.
916
917     Here is a trivial example:
918
919     void *
920     __wrap_malloc (int c)
921     {
922         printf ("malloc called with %ld\n", c);
923         return __real_malloc (c);
924     }

```

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 wish 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`.

US

File: ld.info, Node: Environment, Prev: Options, Up: Invocation

Environment Variables

=====

You can change the behavior of `ld` with the environment variables `GNUTARGET` and `LDEMULATION`.

`GNUTARGET` determines the input-file object format if you don't use `-b` (or its synonym `--format`). Its value should be one of the BFD names for an input format (*note BFD::). If there is no `GNUTARGET` in the environment, `ld` uses the natural format of the target. If `GNUTARGET` is set to `default` then BFD attempts to discover the input format by examining binary input files; this method often succeeds, but there are potential ambiguities, since there is no method of 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.

`LDEMULATION` determines the default emulation if you don't use the `-m` option. The emulation can affect various aspects of linker behaviour, 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.

US

File: ld.info, Node: Commands, Next: Machine Dependent, Prev: Invocation, Up: Top

Command Language

The command language provides explicit control over the link process, allowing complete specification of the mapping between the linker's input files and its output. It controls:

- * input files
- * file formats
- * output file layout
- * addresses of sections
- * placement of common blocks

You may supply a command file (also known as a linker script) to the linker either explicitly through the `-T` option, or implicitly as an ordinary file. Normally you should use the `-T` option. An implicit linker script should only be used when you want to augment, rather than replace, the default linker script; typically an implicit linker script would consist only of `INPUT` or `GROUP` commands.

```

991
992     If the linker opens a file which it cannot recognize as a supported
993 object or archive format, nor as a linker script, it reports an error.
994
995 * Menu:
996
997 * Scripts::                Linker Scripts
998 * Expressions::            Expressions
999 * MEMORY::                 MEMORY Command
1000 * SECTIONS::               SECTIONS Command
1001 * PHDRS::                  PHDRS Command
1002 * Entry Point::            The Entry Point
1003 * Version Script::          Version Script
1004 * Option Commands::         Option Commands
1005
1006 US
1007 File: ld.info,  Node: Scripts,  Next: Expressions,  Up: Commands
1008
1009 Linker Scripts
1010 =====
1011
1012     The `ld' command language is a collection of statements; some are
1013 simple keywords setting a particular option, some are used to select and
1014 group input files or name output files; and two statement types have a
1015 fundamental and pervasive impact on the linking process.
1016
1017     The most fundamental command of the `ld' command language is the
1018 `SECTIONS' command (*note SECTIONS::.). Every meaningful command
1019 script must have a `SECTIONS' command: it specifies a "picture" of the
1020 output file's layout, in varying degrees of detail. No other command
1021 is required in all cases.
1022
1023     The `MEMORY' command complements `SECTIONS' by describing the
1024 available memory in the target architecture. This command is optional;
1025 if you don't use a `MEMORY' command, `ld' assumes sufficient memory is
1026 available in a contiguous block for all output. *Note MEMORY::.
1027
1028     You may include comments in linker scripts just as in C: delimited
1029 by `/*' and `*/'. As in C, comments are syntactically equivalent to
1030 whitespace.
1031
1032 US
1033 File: ld.info,  Node: Expressions,  Next: MEMORY,  Prev: Scripts,  Up: Commands
1034
1035 Expressions
1036 =====
1037
1038     Many useful commands involve arithmetic expressions. The syntax for
1039 expressions in the command language is identical to that of C
1040 expressions, with the following features:
1041     * All expressions evaluated as integers and are of "long" or
1042       "unsigned long" type.
1043
1044     * All constants are integers.
1045
1046     * All of the C arithmetic operators are provided.
1047
1048     * You may reference, define, and create global variables.
1049
1050     * You may call special purpose built-in functions.
1051
1052 * Menu:
1053
1054 * Integers::                Integers
1055 * Symbols::                 Symbol Names
1056 * Location Counter::         The Location Counter

```

```

1057 * Operators:: Operators
1058 * Evaluation:: Evaluation
1059 * Assignment:: Assignment: Defining Symbols
1060 * Arithmetic Functions:: Built-In Functions
1061 * Semicolons:: Semicolon Usage
1062
1063 US
1064 File: ld.info, Node: Integers, Next: Symbols, Up: Expressions
1065
1066 Integers
1067 -----
1068
1069 An octal integer is `0' followed by zero or more of the octal digits
1070 (`01234567').
1071     _as_octal = 0157255;
1072
1073 A decimal integer starts with a non-zero digit followed by zero or
1074 more digits (`0123456789').
1075     _as_decimal = 57005;
1076
1077 A hexadecimal integer is `0x' or `0X' followed by one or more
1078 hexadecimal digits chosen from `0123456789abcdefABCDEF'.
1079     _as_hex = 0xdead;
1080
1081 To write a negative integer, use the prefix operator `-` (*note
1082 Operators::).
1083     _as_neg = -57005;
1084
1085 Additionally the suffixes `K' and `M' may be used to scale a
1086 constant by `1024' or `1024*1024' respectively. For example, the
1087 following all refer to the same quantity:
1088
1089     _fourk_1 = 4K;
1090     _fourk_2 = 4096;
1091     _fourk_3 = 0x1000;
1092
1093 US
1094 File: ld.info, Node: Symbols, Next: Location Counter, Prev: Integers, Up: Expressions
1095
1096 Symbol Names
1097 -----
1098
1099 Unless quoted, symbol names start with a letter, underscore, or point
1100 and may include any letters, underscores, digits, points, and hyphens.
1101 Unquoted symbol names must not conflict with any keywords. You can
1102 specify a symbol which contains odd characters or has the same name as
1103 a keyword, by surrounding the symbol name in double quotes:
1104     "SECTION" = 9;
1105     "with a space" = "also with a space" + 10;
1106
1107 Since symbols can contain many non-alphabetic characters, it is
1108 safest to delimit symbols with spaces. For example, `A-B' is one
1109 symbol, whereas `A - B' is an expression involving subtraction.
1110
1111 US
1112 File: ld.info, Node: Location Counter, Next: Operators, Prev: Symbols, Up:
Expressions
1113
1114 The Location Counter
1115 -----
1116
1117 The special linker variable "dot" `.' always contains the current
1118 output location counter. Since the `.' always refers to a location in
1119 an output section, it must always appear in an expression within a
1120 `SECTIONS' command. The `.' symbol may appear anywhere that an ordinary
1121 symbol is allowed in an expression, but its assignments have a side

```

1122 effect. Assigning a value to the `.' symbol will cause the location
1123 counter to be moved. This may be used to create holes in the output
1124 section. The location counter may never be moved backwards.

```
1125     SECTIONS
1126     {
1127         output :
1128         {
1129             file1(.text)
1130             . = . + 1000;
1131             file2(.text)
1132             . += 1000;
1133             file3(.text)
1134             } = 0x1234;
1135         }
1136
```

1137 In the previous example, `file1' is located at the beginning of the
1138 output section, then there is a 1000 byte gap. Then `file2' appears,
1139 also with a 1000 byte gap following before `file3' is loaded. The
1140 notation `= 0x1234' specifies what data to write in the gaps (*note
1141 Section Options:..).

1142 **US**

1143 File: ld.info, Node: Operators, Next: Evaluation, Prev: Location Counter, Up:
1144 Expressions

1145
1146 Operators
1147 -----

1148
1149 The linker recognizes the standard C set of arithmetic operators,
1150 with the standard bindings and precedence levels:

precedence	associativity	Operators	Notes
(highest)			
1	left	! - ~	(1)
2	left	* / %	
3	left	+ -	
4	left	>> <<	
5	left	== != > < <= >=	
6	left	&	
7	left		
8	left	&&	
9	left		
10	right	? :	
11	right	&= += -= *= /=	(2)
(lowest)			

1164 Notes: (1) Prefix operators (2) *Note Assignment:..

1165 **US**

1166 File: ld.info, Node: Evaluation, Next: Assignment, Prev: Operators, Up: Expressions

1167
1168 Evaluation
1169 -----

1170
1171
1172 The linker uses "lazy evaluation" for expressions; it only calculates
1173 an expression when absolutely necessary. The linker needs the value of
1174 the start address, and the lengths of memory regions, in order to do any
1175 linking at all; these values are computed as soon as possible when the
1176 linker reads in the command file. However, other values (such as symbol
1177 values) are not known or needed until after storage allocation. Such
1178 values are evaluated later, when other information (such as the sizes of
1179 output sections) is available for use in the symbol assignment
1180 expression.
1181

1182 **US**

1183 File: ld.info, Node: Assignment, Next: Arithmetic Functions, Prev: Evaluation, Up:
1184 Expressions

1185

1186 Assignment: Defining Symbols

1187 -----

1188
1189 You may create global symbols, and assign values (addresses) to
1190 global symbols, using any of the C assignment operators:

1191
1192 `SYMBOL = EXPRESSION ;'
1193 `SYMBOL &= EXPRESSION ;'
1194 `SYMBOL += EXPRESSION ;'
1195 `SYMBOL -= EXPRESSION ;'
1196 `SYMBOL *= EXPRESSION ;'
1197 `SYMBOL /= EXPRESSION ;'

1198 Two things distinguish assignment from other operators in `ld'
1199 expressions.

1200 * Assignment may only be used at the root of an expression; `a=b+3;'
1201 is allowed, but `a+b=3;' is an error.

1202
1203 * You must place a trailing semicolon ("<;>") at the end of an
1204 assignment statement.

1205

1206 Assignment statements may appear:

1207 * as commands in their own right in an `ld' script; or

1208

1209 * as independent statements within a `SECTIONS' command; or

1210

1211 * as part of the contents of a section definition in a `SECTIONS'
1212 command.

1213

1214 The first two cases are equivalent in effect--both define a symbol
1215 with an absolute address. The last case defines a symbol whose address
1216 is relative to a particular section (*note SECTIONS:..).

1217

1218 When a linker expression is evaluated and assigned to a variable, it
1219 is given either an absolute or a relocatable type. An absolute
1220 expression type is one in which the symbol contains the value that it
1221 will have in the output file; a relocatable expression type is one in
1222 which the value is expressed as a fixed offset from the base of a
1223 section.

1224

1225 The type of the expression is controlled by its position in the
1226 script file. A symbol assigned within a section definition is created
1227 relative to the base of the section; a symbol assigned in any other
1228 place is created as an absolute symbol. Since a symbol created within a
1229 section definition is relative to the base of the section, it will
1230 remain relocatable if relocatable output is requested. A symbol may be
1231 created with an absolute value even when assigned to within a section
1232 definition by using the absolute assignment function `ABSOLUTE'. For
1233 example, to create an absolute symbol whose address is the last byte of
1234 an output section named `.data':

```
1235     SECTIONS{ ...  
1236         .data :  
1237         {  
1238             *(.data)  
1239             _edata = ABSOLUTE(.) ;  
1240         }  
1241     ... }
```

1242

1243 The linker tries to put off the evaluation of an assignment until all
1244 the terms in the source expression are known (*note Evaluation:..). For
1245 instance, the sizes of sections cannot be known until after allocation,
1246 so assignments dependent upon these are not performed until after
1247 allocation. Some expressions, such as those depending upon the location
1248 counter "dot", `.' must be evaluated during allocation. If the result
1249 of an expression is required, but the value is not available, then an
1250 error results. For example, a script like the following

```
1251     SECTIONS { ...
```

```

1252         text 9+this_isnt_constant :
1253         { ...
1254         }
1255     ... }

```

1256
1257 will cause the error message "`Non constant expression for initial
1258 address'".

1259
1260 In some cases, it is desirable for a linker script to define a symbol
1261 only if it is referenced, and only if it is not defined by any object
1262 included in the link. For example, traditional linkers defined the
1263 symbol `etext'. However, ANSI C requires that the user be able to use
1264 `etext' as a function name without encountering an error. The
1265 `PROVIDE' keyword may be used to define a symbol, such as `etext', only
1266 if it is referenced but not defined. The syntax is `PROVIDE(SYMBOL =
1267 EXPRESSION)'.

1268 **US**

1270 File: ld.info, Node: Arithmetic Functions, Next: Semicolons, Prev: Assignment, Up:
Expressions

1271
1272 Arithmetic Functions
1273 -----

1274
1275 The command language includes a number of built-in functions for use
1276 in link script expressions.

1277 `ABSOLUTE(EXP)'

1278 Return the absolute (non-relocatable, as opposed to non-negative)
1279 value of the expression EXP. Primarily useful to assign an
1280 absolute value to a symbol within a section definition, where
1281 symbol values are normally section-relative.

1282
1283 `ADDR(SECTION)'

1284 Return the absolute address of the named SECTION. Your script must
1285 previously have defined the location of that section. In the
1286 following example, `symbol\_1' and `symbol_2' are assigned identical
1287 values:

```

1288     SECTIONS{ ...
1289         .output1 :
1290         {
1291             start_of_output_1 = ABSOLUTE(.);
1292             ...
1293         }
1294         .output :
1295         {
1296             symbol_1 = ADDR(.output1);
1297             symbol_2 = start_of_output_1;
1298         }
1299     ... }

```

1300
1301 `LOADADDR(SECTION)'

1302 Return the absolute load address of the named SECTION. This is
1303 normally the same as `ADDR', but it may be different if the `AT'
1304 keyword is used in the section definition (*note Section
1305 Options:::).

1306
1307 `ALIGN(EXP)'

1308 Return the result of the current location counter (`.') aligned to
1309 the next EXP boundary. EXP must be an expression whose value is a
1310 power of two. This is equivalent to
1311 $(. + EXP - 1) \& \sim(EXP - 1)$

1312
1313 `ALIGN' doesn't change the value of the location counter--it just
1314 does arithmetic on it. As an example, to align the output `.data'
1315 section to the next `0x2000' byte boundary after the preceding
1316 section and to set a variable within the section to the next

1317 `0x8000' boundary after the input sections:

```
1318       SECTIONS{ ...
1319       .data ALIGN(0x2000): {
1320        *(.data)
1321        variable = ALIGN(0x8000);
1322       }
1323       ... }
```

1324
1325 The first use of `ALIGN' in this example specifies the location of
1326 a section because it is used as the optional START attribute of a
1327 section definition (*note Section Options::.). The second use
1328 simply defines the value of a variable.

1329
1330 The built-in `NEXT' is closely related to `ALIGN'.

1331
1332 `DEFINED(SYMBOL)'

1333 Return 1 if SYMBOL is in the linker global symbol table and is
1334 defined, otherwise return 0. You can use this function to provide
1335 default values for symbols. For example, the following
1336 command-file fragment shows how to set a global symbol `begin' to
1337 the first location in the `.text' section--but if a symbol called
1338 `begin' already existed, its value is preserved:

```
1339       SECTIONS{ ...
1340        .text : {
1341         begin = DEFINED(begin) ? begin : . ;
1342        ...
1343       }
1344       ... }
```

1345
1346
1347 `NEXT(EXP)'

1348 Return the next unallocated address that is a multiple of EXP.
1349 This function is closely related to `ALIGN(EXP)'; unless you use
1350 the `MEMORY' command to define discontinuous memory for the output
1351 file, the two functions are equivalent.

1352
1353 `SIZEOF(SECTION)'

1354 Return the size in bytes of the named SECTION, if that section has
1355 been allocated. In the following example, `symbol_1' and
1356 `symbol_2' are assigned identical values:

```
1357       SECTIONS{ ...
1358        .output {
1359         .start = . ;
1360        ...
1361        .end = . ;
1362       }
1363       symbol_1 = .end - .start ;
1364       symbol_2 = SIZEOF(.output);
1365       ... }
```

1366
1367 `SIZEOF_HEADERS'

1368 `sizeof_headers'

1369 Return the size in bytes of the output file's headers. You can
1370 use this number as the start address of the first section, if you
1371 choose, to facilitate paging.

1372
1373 `MAX(EXP1, EXP2)'

1374 Returns the maximum of EXP1 and EXP2.

1375
1376 `MIN(EXP1, EXP2)'

1377 Returns the minimum of EXP1 and EXP2.

1378
1379 **US**

1380 File: ld.info, Node: Semicolons, Prev: Arithmetic Functions, Up: Expressions

1381
1382 Semicolons

```

1383 -----
1384
1385     Semicolons ("<;>") are required in the following places. In all
1386 other places they can appear for aesthetic reasons but are otherwise
1387 ignored.
1388
1389 `Assignment'
1390     Semicolons must appear at the end of assignment expressions.
1391     *Note Assignment::
1392
1393 `PHDRS'
1394     Semicolons must appear at the end of a `PHDRS' statement. *Note
1395 PHDRS::
1396
1397 US
1398 File: ld.info, Node: MEMORY, Next: SECTIONS, Prev: Expressions, Up: Commands
1399
1400 Memory Layout
1401 =====
1402
1403     The linker's default configuration permits allocation of all
1404 available memory. You can override this configuration by using the
1405 `MEMORY' command. The `MEMORY' command describes the location and size
1406 of blocks of memory in the target. By using it carefully, you can
1407 describe which memory regions may be used by the linker, and which
1408 memory regions it must avoid. The linker does not shuffle sections to
1409 fit into the available regions, but does move the requested sections
1410 into the correct regions and issue errors when the regions become too
1411 full.
1412
1413     A command file may contain at most one use of the `MEMORY' command;
1414 however, you can define as many blocks of memory within it as you wish.
1415 The syntax is:
1416
1417     MEMORY
1418     {
1419         NAME (ATTR) : ORIGIN = ORIGIN, LENGTH = LEN
1420         ...
1421     }
1422
1423 `NAME'
1424     is a name used internally by the linker to refer to the region. Any
1425 symbol name may be used. The region names are stored in a separate
1426 name space, and will not conflict with symbols, file names or
1427 section names. Use distinct names to specify multiple regions.
1428
1429 `(ATTR)'
1430     is an optional list of attributes that specify whether to use a
1431 particular memory to place sections that are not listed in the
1432 linker script. Valid attribute lists must be made up of the
1433 characters "`ALIRWX'" that match section attributes. If you omit
1434 the attribute list, you may omit the parentheses around it as
1435 well. The attributes currently supported are:
1436
1437     ``Letter''
1438         `Section Attribute'
1439
1440     ``R''
1441         Read-only sections.
1442
1443     ``W''
1444         Read/write sections.
1445
1446     ``X''
1447         Sections containing executable code.
1448

```

```

1449     ``A''
1450         Allocated sections.
1451
1452     ``I''
1453         Initialized sections.
1454
1455     ``L''
1456         Same as `I'.
1457
1458     ``!'
1459         Invert the sense of any of the following attributes.
1460
1461 `ORIGIN'
1462     is the start address of the region in physical memory. It is an
1463     expression that must evaluate to a constant before memory
1464     allocation is performed. The keyword `ORIGIN' may be abbreviated
1465     to `org' or `o' (but not, for example, `ORG').
1466
1467 `LEN'
1468     is the size in bytes of the region (an expression). The keyword
1469     `LENGTH' may be abbreviated to `len' or `l'.
1470
1471     For example, to specify that memory has two regions available for
1472     allocation--one starting at 0 for 256 kilobytes, and the other starting
1473     at `0x40000000' for four megabytes. The `rom' memory region will get
1474     all sections without an explicit memory register that are either
1475     read-only or contain code, while the `ram' memory region will get the
1476     sections.
1477

```

MEMORY

```

1478     {
1479         rom (rx)  : ORIGIN = 0, LENGTH = 256K
1480         ram (!rx) : org = 0x40000000, l = 4M
1481     }
1482

```

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 (*note Section Options:..). If the combined output sections directed to a region are too big for the region, the linker will issue an error message.

US

File: ld.info, Node: SECTIONS, Next: PHDRS, Prev: MEMORY, Up: Commands

Specifying Output Sections

=====

The `SECTIONS' command controls exactly where input sections are placed into output sections, their order in the output file, and to which output sections they are allocated.

You may use at most one `SECTIONS' command in a script file, but you can have as many statements within it as you wish. Statements within the `SECTIONS' command can do one of three things:

- * define the entry point;
- * assign a value to a symbol;
- * describe the placement of a named output section, and which input sections go into it.

You can also use the first two operations--defining the entry point and defining symbols--outside the `SECTIONS' command: *note Entry Point:.., and *Note Assignment:.. They are permitted here as well for your convenience in reading the script, so that symbols and the entry

1515 point can be defined at meaningful points in your output-file layout.

1516

1517 If you do not use a `SECTIONS' command, the linker places each input
1518 section into an identically named output section in the order that the
1519 sections are first encountered in the input files. If all input
1520 sections are present in the first file, for example, the order of
1521 sections in the output file will match the order in the first input
1522 file.

1523

1524 * Menu:

1525

1526 * Section Definition:: Section Definitions
1527 * Section Placement:: Section Placement
1528 * Section Data Expressions:: Section Data Expressions
1529 * Section Options:: Optional Section Attributes
1530 * Overlays:: Overlays

1531

1532 **US**

1533 File: ld.info, Node: Section Definition, Next: Section Placement, Up: SECTIONS

1534

1535 Section Definitions

1536 -----

1537

1538 The most frequently used statement in the `SECTIONS' command is the
1539 "section definition", which specifies the properties of an output
1540 section: its location, alignment, contents, fill pattern, and target
1541 memory region. Most of these specifications are optional; the simplest
1542 form of a section definition is

```
1543     SECTIONS { ...  
1544         SECNAME : {  
1545             CONTENTS  
1546         }  
1547     ... }
```

1548

1549 SECNAME is the name of the output section, and CONTENTS a specification
1550 of what goes there--for example, a list of input files or sections of
1551 input files (*note Section Placement::). The whitespace around
1552 SECNAME is required, so that the section name is unambiguous. The
1553 other whitespace shown is optional. You do need the colon `:' and the
1554 braces `{ }', however.

1555

1556 SECNAME must meet the constraints of your output format. In formats
1557 which only support a limited number of sections, such as `a.out', the
1558 name must be one of the names supported by the format (`a.out', for
1559 example, allows only `.text', `.data' or `.bss'). If the output format
1560 supports any number of sections, but with numbers and not names (as is
1561 the case for Oasys), the name should be supplied as a quoted numeric
1562 string. A section name may consist of any sequence of characters, but
1563 any name which does not conform to the standard `ld' symbol name syntax
1564 must be quoted. *Note Symbol Names: Symbols.

1565

1566 The special SECNAME `/DISCARD/' may be used to discard input
1567 sections. Any sections which are assigned to an output section named
1568 `/DISCARD/' are not included in the final link output.

1569

1570 The linker will not create output sections which do not have any
1571 contents. This is for convenience when referring to input sections that
1572 may or may not exist. For example,

```
1573     .foo { *(.foo) }
```

1574 will only create a `.foo' section in the output file if there is a
1575 `.foo' section in at least one input file.

1576

1577 **US**

1578 File: ld.info, Node: Section Placement, Next: Section Data Expressions, Prev: Section
1579 Definition, Up: SECTIONS

```

1580 Section Placement
1581 -----
1582
1583 In a section definition, you can specify the contents of an output
1584 section by listing particular input files, by listing particular
1585 input-file sections, or by a combination of the two. You can also place
1586 arbitrary data in the section, and define symbols relative to the
1587 beginning of the section.
1588
1589 The CONTENTS of a section definition may include any of the
1590 following kinds of statement. You can include as many of these as you
1591 like in a single section definition, separated from one another by
1592 whitespace.
1593
1594 `FILENAME'
1595     You may simply name a particular input file to be placed in the
1596     current output section; *all* sections from that file are placed
1597     in the current section definition. If the file name has already
1598     been mentioned in another section definition, with an explicit
1599     section name list, then only those sections which have not yet
1600     been allocated are used.
1601
1602     To specify a list of particular files by name:
1603         .data : { afile.o bfile.o cfile.o }
1604
1605     The example also illustrates that multiple statements can be
1606     included in the contents of a section definition, since each file
1607     name is a separate statement.
1608
1609 `FILENAME( SECTION )'
1610 `FILENAME( SECTION , SECTION, ... )'
1611 `FILENAME( SECTION SECTION ... )'
1612     You can name one or more sections from your input files, for
1613     insertion in the current output section. If you wish to specify a
1614     list of input-file sections inside the parentheses, separate the
1615     section names with whitespace.
1616
1617 `* (SECTION)'
1618 `* (SECTION, SECTION, ...)'
1619 `* (SECTION SECTION ...)'
1620     Instead of explicitly naming particular input files in a link
1621     control script, you can refer to *all* files from the `ld' command
1622     line: use `*' instead of a particular file name before the
1623     parenthesized input-file section list.
1624
1625     If you have already explicitly included some files by name, `*'
1626     refers to all *remaining* files--those whose places in the output
1627     file have not yet been defined.
1628
1629     For example, to copy sections `1' through `4' from an Oasys file
1630     into the `.text' section of an `a.out' file, and sections `13' and
1631     `14' into the `.data' section:
1632         SECTIONS {
1633             .text :{
1634                 *("1" "2" "3" "4")
1635             }
1636
1637             .data :{
1638                 *("13" "14")
1639             }
1640         }
1641
1642     `[ SECTION ... ]' used to be accepted as an alternate way to
1643     specify named sections from all unallocated input files. Because
1644     some operating systems (VMS) allow brackets in file names, that
1645     notation is no longer supported.

```

```

1646
1647 `FILENAME` ( COMMON ) ''
1648 `*( COMMON )'
1649     Specify where in your output file to place uninitialized data with
1650     this notation. `*(COMMON)' by itself refers to all uninitialized
1651     data from all input files (so far as it is not yet allocated);
1652     FILENAME` (COMMON)' refers to uninitialized data from a particular
1653     file. Both are special cases of the general mechanisms for
1654     specifying where to place input-file sections: `ld' permits you to
1655     refer to uninitialized data as if it were in an input-file section
1656     named `COMMON', regardless of the input file's format.
1657

```

```

1658     In any place where you may use a specific file or section name, you
1659     may also use a wildcard pattern. The linker handles wildcards much as
1660     the Unix shell does. A `*' character matches any number of characters.
1661     A `?' character matches any single character. The sequence `[CHARS]'
1662     will match a single instance of any of the CHARS; the `-' character may
1663     be used to specify a range of characters, as in `[a-z]' to match any
1664     lower case letter. A `\" character may be used to quote the following
1665     character.
1666

```

```

1667     When a file name is matched with a wildcard, the wildcard characters
1668     will not match a `/' character (used to separate directory names on
1669     Unix). A pattern consisting of a single `*' character is an exception;
1670     it will always match any file name. In a section name, the wildcard
1671     characters will match a `/' character.
1672

```

```

1673     Wildcards only match files which are explicitly specified on the
1674     command line. The linker does not search directories to expand
1675     wildcards. However, if you specify a simple file name--a name with no
1676     wildcard characters--in a linker script, and the file name is not also
1677     specified on the command line, the linker will attempt to open the file
1678     as though it appeared on the command line.
1679

```

```

1680     In the following example, the command script arranges the output file
1681     into three consecutive sections, named `.text', `.data', and `.bss',
1682     taking the input for each from the correspondingly named sections of
1683     all the input files:
1684

```

```

1685     SECTIONS {
1686         .text : { *(.text) }
1687         .data : { *(.data) }
1688         .bss : { *(.bss) *(COMMON) }
1689     }
1690

```

```

1691     The following example reads all of the sections from file `all.o'
1692     and places them at the start of output section `outputa' which starts
1693     at location `0x10000'. All of section `.input1' from file `foo.o'
1694     follows immediately, in the same output section. All of section
1695     `.input2' from `foo.o' goes into output section `outputb', followed by
1696     section `.input1' from `fool.o'. All of the remaining `.input1' and
1697     `.input2' sections from any files are written to output section
1698     `outputc'.
1699

```

```

1700     SECTIONS {
1701         outputa 0x10000 :
1702         {
1703             all.o
1704             foo.o (.input1)
1705         }
1706         outputb :
1707         {
1708             foo.o (.input2)
1709             fool.o (.input1)
1710         }
1711         outputc :

```

```

1712     {
1713     *(.input1)
1714     *(.input2)
1715     }
1716 }
1717
1718     This example shows how wildcard patterns might be used to partition
1719     files. All '.text' sections are placed in '.text', and all '.bss'
1720     sections are placed in '.bss'. For all files beginning with an upper
1721     case character, the '.data' section is placed into '.DATA'; for all
1722     other files, the '.data' section is placed into '.data'.
1723

```

```

1724     SECTIONS {
1725     .text : { *(.text) }
1726     .DATA : { [A-Z]*(.data) }
1727     .data : { *(.data) }
1728     .bss : { *(.bss) }
1729     }
1730

```

1731 **US**

1732 File: ld.info, Node: Section Data Expressions, Next: Section Options, Prev: Section Placement, Up: SECTIONS

1733 Section Data Expressions

1734 -----

1735
1736
1737 The foregoing statements arrange, in your output file, data
1738 originating from your input files. You can also place data directly in
1739 an output section from the link command script. Most of these
1740 additional statements involve expressions (*note Expressions:..).
1741 Although these statements are shown separately here for ease of
1742 presentation, no such segregation is needed within a section definition
1743 in the 'SECTIONS' command; you can intermix them freely with any of the
1744 statements we've just described.

1745 'CREATE_OBJECT_SYMBOLS'

1746
1747 Create a symbol for each input file in the current section, set to
1748 the address of the first byte of data written from that input
1749 file. For instance, with 'a.out' files it is conventional to have
1750 a symbol for each input file. You can accomplish this by defining
1751 the output '.text' section as follows:

```

1752     SECTIONS {
1753     .text 0x2020 :
1754     {
1755     CREATE_OBJECT_SYMBOLS
1756     *(.text)
1757     _etext = ALIGN(0x2000);
1758     }
1759     ...
1760     }
1761

```

1762 If 'sample.ld' is a file containing this script, and 'a.o', 'b.o',
1763 'c.o', and 'd.o' are four input files with contents like the
1764 following--

```

1765     /* a.c */
1766

```

```

1767     afunction() { }
1768     int adata=1;
1769     int abss;
1770

```

1771
1772 'ld -M -T sample.ld a.o b.o c.o d.o' would create a map like this,
1773 containing symbols matching the object file names:

```

1774     00000000 A __DYNAMIC
1775     00004020 B _abss
1776     00004000 D _adata
1777     00002020 T _afunction

```

```

1777      00004024 B _bbss
1778      00004008 D _bdata
1779      00002038 T _bfunction
1780      00004028 B _cbss
1781      00004010 D _cdata
1782      00002050 T _cfunction
1783      0000402c B _dbss
1784      00004018 D _ddata
1785      00002068 T _dfunction
1786      00004020 D _edata
1787      00004030 B _end
1788      00004000 T _etext
1789      00002020 t a.o
1790      00002038 t b.o
1791      00002050 t c.o
1792      00002068 t d.o
1793
1794 `SYMBOL = EXPRESSION ;'
1795 `SYMBOL F= EXPRESSION ;'
1796     SYMBOL is any symbol name (*note Symbols:..). "F=" refers to any
1797     of the operators `&= += -= *= /= ' which combine arithmetic and
1798     assignment.
1799
1800     When you assign a value to a symbol within a particular section
1801     definition, the value is relative to the beginning of the section
1802     (*note Assignment:..). If you write
1803
1804     SECTIONS {
1805         abs = 14 ;
1806         ...
1807         .data : { ... rel = 14 ; ... }
1808         abs2 = 14 + ADDR(.data);
1809         ...
1810     }
1811
1812     `abs' and `rel' do not have the same value; `rel' has the same
1813     value as `abs2'.
1814
1815 `BYTE(EXPRESSION)'
1816 `SHORT(EXPRESSION)'
1817 `LONG(EXPRESSION)'
1818 `QUAD(EXPRESSION)'
1819 `SQUAD(EXPRESSION)'
1820     By including one of these four statements in a section definition,
1821     you can explicitly place one, two, four, eight unsigned, or eight
1822     signed bytes (respectively) at the current address of that
1823     section. When using a 64 bit host or target, `QUAD' and `SQUAD'
1824     are the same. When both host and target are 32 bits, `QUAD' uses
1825     an unsigned 32 bit value, and `SQUAD' sign extends the value.
1826     Both will use the correct endianness when writing out the value.
1827
1828     Multiple-byte quantities are represented in whatever byte order is
1829     appropriate for the output file format (*note BFD:..).
1830
1831 `FILL(EXPRESSION)'
1832     Specify the "fill pattern" for the current section. Any otherwise
1833     unspecified regions of memory within the section (for example,
1834     regions you skip over by assigning a new value to the location
1835     counter `.' ) are filled with the two least significant bytes from
1836     the EXPRESSION argument. A `FILL' statement covers memory
1837     locations *after* the point it occurs in the section definition; by
1838     including more than one `FILL' statement, you can have different
1839     fill patterns in different parts of an output section.
1840
1841 US
1842 File: ld.info, Node: Section Options, Next: Overlays, Prev: Section Data

```

Expressions, Up: SECTIONS

Optional Section Attributes

Here is the full syntax of a section definition, including all the optional portions:

```
SECTIONS {
...
SECNAME START BLOCK(ALIGN) (NOLOAD) : AT ( LDADR )
{ CONTENTS } >REGION :PHDR =FILL
...
}
```

SECNAME and CONTENTS are required. *Note Section Definition::, and *Note Section Placement::, for details on CONTENTS. The remaining elements--START, `BLOCK(ALIGN)', `(NOLOAD)', `AT ( LDADR )', `>REGION', `:PHDR', and `=FILL'--are all optional.

`START'

You can force the output section to be loaded at a specified address by specifying START immediately following the section name. START can be represented as any expression. The following example generates section OUTPUT at location `0x40000000':

```
SECTIONS {
...
output 0x40000000: {
...
}
...
}
```

`BLOCK(ALIGN)'

You can include `BLOCK()' specification to advance the location counter `.' prior to the beginning of the section, so that the section will begin at the specified alignment. ALIGN is an expression.

`(NOLOAD)'

The `(NOLOAD)' directive will mark a section to not be loaded at run time. The linker will process the section normally, but will mark it so that a program loader will not load it into memory. For example, in the script sample below, the `ROM' section is addressed at memory location `0' and does not need to be loaded when the program is run. The contents of the `ROM' section will appear in the linker output file as usual.

```
SECTIONS {
ROM 0 (NOLOAD) : { ... }
...
}
```

`AT (LDADR)'

The expression LDADR that follows the `AT' keyword specifies the load address of the section. The default (if you do not use the `AT' keyword) is to make the load address the same as the relocation address. This feature is designed to make it easy to build a ROM image. For example, this `SECTIONS' definition creates two output sections: one called `.text', which starts at `0x1000', and one called `.mdata', which is loaded at the end of the `.text' section even though its relocation address is `0x2000'. The symbol `\_data' is defined with the value `0x2000':

```
SECTIONS
```

```

1908     {
1909     .text 0x1000 : { *(.text) _etext = . ; }
1910     .mdata 0x2000 :
1911         AT ( ADDR(.text) + SIZEOF ( .text ) )
1912         { _data = . ; *(.data); _edata = . ; }
1913     .bss 0x3000 :
1914         { _bstart = . ; *(.bss) *(COMMON) ; _bend = . ; }
1915     }
1916

```

The run-time initialization code (for C programs, usually `crt0') for use with a ROM generated this way has to include something like the following, to copy the initialized data from the ROM image to its runtime address:

```

1922     char *src = _etext;
1923     char *dst = _data;
1924
1925     /* ROM has data at end of text; copy it. */
1926     while (dst < _edata) {
1927         *dst++ = *src++;
1928     }
1929
1930     /* Zero bss */
1931     for (dst = _bstart; dst < _bend; dst++)
1932         *dst = 0;
1933

```

`>REGION'

Assign this section to a previously defined region of memory.
 *Note MEMORY::.

`:PHDR'

Assign this section to a segment described by a program header.
 *Note PHDRS::. If a section is assigned to one or more segments, then all subsequent allocated sections will be assigned to those segments as well, unless they use an explicitly `:PHDR' modifier. To prevent a section from being assigned to a segment when it would normally default to one, use `:NONE'.

`=FILL'

Including `=FILL' in a section definition specifies the initial fill value for that section. You may use any expression to specify FILL. Any unallocated holes in the current output section when written to the output file will be filled with the two least significant bytes of the value, repeated as necessary. You can also change the fill value with a `FILL' statement in the CONTENTS of a section definition.

US

File: ld.info, Node: Overlays, Prev: Section Options, Up: SECTIONS

Overlays

The `OVERLAY' command provides an easy way to describe sections which are to be loaded as part of a single memory image but are to be run at the same memory address. At run time, some sort of overlay manager will copy the overlaid sections in and out of the runtime memory address as required, perhaps by simply manipulating addressing bits. This approach can be useful, for example, when a certain region of memory is faster than another.

The `OVERLAY' command is used within a `SECTIONS' command. It appears as follows:

```

1971     OVERLAY START : [ NOCROSSREFS ] AT ( LDADDR )
1972     {
1973         SECNAME1 { CONTENTS } :PHDR =FILL

```

```

1974         SECNAME2 { CONTENTS } :PHDR =FILL
1975         ...
1976     } >REGION :PHDR =FILL
1977

```

1978 Everything is optional except 'OVERLAY' (a keyword), and each
 1979 section must have a name (SECNAME1 and SECNAME2 above). The section
 1980 definitions within the 'OVERLAY' construct are identical to those
 1981 within the general 'SECTIONS' construct (*note SECTIONS::.), except that
 1982 no addresses and no memory regions may be defined for sections within
 1983 an 'OVERLAY'.

1984
 1985 The sections are all defined with the same starting address. The
 1986 load addresses of the sections are arranged such that they are
 1987 consecutive in memory starting at the load address used for the
 1988 'OVERLAY' as a whole (as with normal section definitions, the load
 1989 address is optional, and defaults to the start address; the start
 1990 address is also optional, and defaults to '.').

1991
 1992 If the 'NOCROSSREFS' keyword is used, and there any references among
 1993 the sections, the linker will report an error. Since the sections all
 1994 run at the same address, it normally does not make sense for one
 1995 section to refer directly to another. *Note NOCROSSREFS: Option
 1996 Commands.

1997
 1998 For each section within the 'OVERLAY', the linker automatically
 1999 defines two symbols. The symbol '__load_start_SECNAME' is defined as
 2000 the starting load address of the section. The symbol
 2001 '__load_stop_SECNAME' is defined as the final load address of the
 2002 section. Any characters within SECNAME which are not legal within C
 2003 identifiers are removed. C (or assembler) code may use these symbols
 2004 to move the overlaid sections around as necessary.

2005
 2006 At the end of the overlay, the value of '.' is set to the start
 2007 address of the overlay plus the size of the largest section.

2008
 2009 Here is an example. Remember that this would appear inside a
 2010 'SECTIONS' construct.

```

2011
2012     OVERLAY 0x1000 : AT (0x4000)
2013     {
2014         .text0 { o1/*.o(.text) }
2015         .text1 { o2/*.o(.text) }
2016     }
2017

```

2018 This will define both '.text0' and '.text1' to start at address
 2019 0x1000. '.text0' will be loaded at address 0x4000, and '.text1' will
 2020 be loaded immediately after '.text0'. The following symbols will be
 2021 defined: '__load_start_text0', '__load_stop_text0',
 2022 '__load_start_text1', '__load_stop_text1'.

2023
 2024 C code to copy overlay '.text1' into the overlay area might look
 2025 like the following.

```

2026
2027     extern char __load_start_text1, __load_stop_text1;
2028     memcpy ((char *) 0x1000, &__load_start_text1,
2029             &__load_stop_text1 - &__load_start_text1);
2030

```

2031 Note that the 'OVERLAY' command is just syntactic sugar, since
 2032 everything it does can be done using the more basic commands. The above
 2033 example could have been written identically as follows.

```

2034
2035     .text0 0x1000 : AT (0x4000) { o1/*.o(.text) }
2036     __load_start_text0 = LOADADDR (.text0);
2037     __load_stop_text0 = LOADADDR (.text0) + SIZEOF (.text0);
2038     .text1 0x1000 : AT (0x4000 + SIZEOF (.text0)) { o2/*.o(.text) }
2039     __load_start_text1 = LOADADDR (.text1);

```

```

2040     __load_stop_text1 = LOADADDR (.text1) + SIZEOF (.text1);
2041     . = 0x1000 + MAX (SIZEOF (.text0), SIZEOF (.text1));
2042
2043 US
2044 File: ld.info, Node: PHDRS, Next: Entry Point, Prev: SECTIONS, Up: Commands
2045
2046 ELF Program Headers
2047 =====
2048
2049 The ELF object file format uses "program headers", which are read by
2050 the system loader and describe how the program should be loaded into
2051 memory. These program headers must be set correctly in order to run the
2052 program on a native ELF system. The linker will create reasonable
2053 program headers by default. However, in some cases, it is desirable to
2054 specify the program headers more precisely; the `PHDRS' command may be
2055 used for this purpose. When the `PHDRS' command is used, the linker
2056 will not generate any program headers itself.
2057
2058 The `PHDRS' command is only meaningful when generating an ELF output
2059 file. It is ignored in other cases. This manual does not describe the
2060 details of how the system loader interprets program headers; for more
2061 information, see the ELF ABI. The program headers of an ELF file may
2062 be displayed using the `-p' option of the `objdump' command.
2063
2064 This is the syntax of the `PHDRS' command. The words `PHDRS',
2065 `FILEHDR', `AT', and `FLAGS' are keywords.
2066
2067 PHDRS
2068 {
2069     NAME TYPE [ FILEHDR ] [ PHDRS ] [ AT ( ADDRESS ) ]
2070     [ FLAGS ( FLAGS ) ] ;
2071 }
2072
2073 The NAME is used only for reference in the `SECTIONS' command of the
2074 linker script. It does not get put into the output file.
2075
2076 Certain program header types describe segments of memory which are
2077 loaded from the file by the system loader. In the linker script, the
2078 contents of these segments are specified by directing allocated output
2079 sections to be placed in the segment. To do this, the command
2080 describing the output section in the `SECTIONS' command should use
2081 `:NAME', where NAME is the name of the program header as it appears in
2082 the `PHDRS' command. *Note Section Options::.
2083
2084 It is normal for certain sections to appear in more than one segment.
2085 This merely implies that one segment of memory contains another. This
2086 is specified by repeating `:NAME', using it once for each program
2087 header in which the section is to appear.
2088
2089 If a section is placed in one or more segments using `:NAME', then
2090 all subsequent allocated sections which do not specify `:NAME' are
2091 placed in the same segments. This is for convenience, since generally
2092 a whole set of contiguous sections will be placed in a single segment.
2093 To prevent a section from being assigned to a segment when it would
2094 normally default to one, use `:NONE'.
2095
2096 The `FILEHDR' and `PHDRS' keywords which may appear after the
2097 program header type also indicate contents of the segment of memory.
2098 The `FILEHDR' keyword means that the segment should include the ELF
2099 file header. The `PHDRS' keyword means that the segment should include
2100 the ELF program headers themselves.
2101
2102 The TYPE may be one of the following. The numbers indicate the
2103 value of the keyword.
2104
2105 `PT_NULL' (0)

```

```

2106         Indicates an unused program header.
2107
2108     `PT_LOAD' (1)
2109         Indicates that this program header describes a segment to be
2110         loaded from the file.
2111
2112     `PT_DYNAMIC' (2)
2113         Indicates a segment where dynamic linking information can be found.
2114
2115     `PT_INTERP' (3)
2116         Indicates a segment where the name of the program interpreter may
2117         be found.
2118
2119     `PT_NOTE' (4)
2120         Indicates a segment holding note information.
2121
2122     `PT_SHLIB' (5)
2123         A reserved program header type, defined but not specified by the
2124         ELF ABI.
2125
2126     `PT_PHDR' (6)
2127         Indicates a segment where the program headers may be found.
2128
2129     EXPRESSION
2130         An expression giving the numeric type of the program header. This
2131         may be used for types not defined above.
2132
2133         It is possible to specify that a segment should be loaded at a
2134         particular address in memory. This is done using an `AT' expression.
2135         This is identical to the `AT' command used in the `SECTIONS' command
2136         (*note Section Options::.). Using the `AT' command for a program
2137         header overrides any information in the `SECTIONS' command.
2138
2139         Normally the segment flags are set based on the sections. The
2140         `FLAGS' keyword may be used to explicitly specify the segment flags.
2141         The value of FLAGS must be an integer. It is used to set the `p_flags'
2142         field of the program header.
2143
2144         Here is an example of the use of `PHDRS'. This shows a typical set
2145         of program headers used on a native ELF system.
2146
2147         PHDRS
2148         {
2149             headers PT_PHDR PHDRS ;
2150             interp PT_INTERP ;
2151             text PT_LOAD FILEHDR PHDRS ;
2152             data PT_LOAD ;
2153             dynamic PT_DYNAMIC ;
2154         }
2155
2156         SECTIONS
2157         {
2158             . = SIZEOF_HEADERS;
2159             .interp : { *(.interp) } :text :interp
2160             .text : { *(.text) } :text
2161             .rodata : { *(.rodata) } /* defaults to :text */
2162             ...
2163             . = . + 0x1000; /* move to a new page in memory */
2164             .data : { *(.data) } :data
2165             .dynamic : { *(.dynamic) } :data :dynamic
2166             ...
2167         }
2168
2169 US
2170     File: ld.info, Node: Entry Point, Next: Version Script, Prev: PHDRS, Up: Commands
2171

```

2172 The Entry Point
2173 =====

2174
2175 The linker command language includes a command specifically for
2176 defining the first executable instruction in an output file (its "entry
2177 point"). Its argument is a symbol name:

2178 ENTRY(SYMBOL)

2179
2180 Like symbol assignments, the 'ENTRY' command may be placed either as
2181 an independent command in the command file, or among the section
2182 definitions within the 'SECTIONS' command--whatever makes the most
2183 sense for your layout.

2184
2185 'ENTRY' is only one of several ways of choosing the entry point.
2186 You may indicate it in any of the following ways (shown in descending
2187 order of priority: methods higher in the list override methods lower
2188 down).

2189 * the '-e' ENTRY command-line option;

2190
2191 * the 'ENTRY(SYMBOL)' command in a linker control script;

2192
2193 * the value of the symbol 'start', if present;

2194
2195 * the address of the first byte of the '.text' section, if present;

2196
2197 * The address '0'.

2198
2199 For example, you can use these rules to generate an entry point with
2200 an assignment statement: if no symbol 'start' is defined within your
2201 input files, you can simply define it, assigning it an appropriate
2202 value--

2203
2204 start = 0x2020;

2205
2206 The example shows an absolute address, but you can use any expression.
2207 For example, if your input object files use some other symbol-name
2208 convention for the entry point, you can just assign the value of
2209 whatever symbol contains the start address to 'start':

2210
2211 start = other_symbol ;

2212
2213 **US**

2214 File: ld.info, Node: Version Script, Next: Option Commands, Prev: Entry Point, Up:
Commands

2215
2216 Version Script
2217 =====

2218
2219 The linker command script includes a command specifically for
2220 specifying a version script, and is only meaningful for ELF platforms
2221 that support shared libraries. A version script can be build directly
2222 into the linker script that you are using, or you can supply the
2223 version script as just another input file to the linker at the time
2224 that you link. The command script syntax is:

2225 VERSION { version script contents }

2226 The version script can also be specified to the linker by means of
2227 the '--version-script' linker command line option. Version scripts are
2228 only meaningful when creating shared libraries.

2229
2230 The format of the version script itself is identical to that used by
2231 Sun's linker in Solaris 2.5. Versioning is done by defining a tree of
2232 version nodes with the names and interdependencies specified in the
2233 version script. The version script can specify which symbols are bound
2234 to which version nodes, and it can reduce a specified set of symbols to
2235 local scope so that they are not globally visible outside of the shared
2236 library.

```

2237
2238     The easiest way to demonstrate the version script language is with a
2239 few examples.
2240
2241     VERS_1.1 {
2242         global:
2243             fool;
2244         local:
2245             old*;
2246             original*;
2247             new*;
2248     };
2249
2250     VERS_1.2 {
2251         foo2;
2252     } VERS_1.1;
2253
2254     VERS_2.0 {
2255         bar1; bar2;
2256     } VERS_1.2;
2257

```

2258 In this example, three version nodes are defined. ``VERS_1.1'` is the
2259 first version node defined, and has no other dependencies. The symbol
2260 ``fool'` is bound to this version node, and a number of symbols that have
2261 appeared within various object files are reduced in scope to local so
2262 that they are not visible outside of the shared library.

2263
2264 Next, the node ``VERS_1.2'` is defined. It depends upon ``VERS_1.1'`.
2265 The symbol ``foo2'` is bound to this version node.

2266
2267 Finally, the node ``VERS_2.0'` is defined. It depends upon
2268 ``VERS_1.2'`. The symbols ``bar1'` and ``bar2'` are bound to this version
2269 node.

2270
2271 Symbols defined in the library which aren't specifically bound to a
2272 version node are effectively bound to an unspecified base version of the
2273 library. It is possible to bind all otherwise unspecified symbols to a
2274 given version node using ``global: *'` somewhere in the version script.

2275
2276 Lexically the names of the version nodes have no specific meaning
2277 other than what they might suggest to the person reading them. The
2278 ``2.0'` version could just as well have appeared in between ``1.1'` and
2279 ``1.2'`. However, this would be a confusing way to write a version
2280 script.

2281
2282 When you link an application against a shared library that has
2283 versioned symbols, the application itself knows which version of each
2284 symbol it requires, and it also knows which version nodes it needs from
2285 each shared library it is linked against. Thus at runtime, the dynamic
2286 loader can make a quick check to make sure that the libraries you have
2287 linked against do in fact supply all of the version nodes that the
2288 application will need to resolve all of the dynamic symbols. In this
2289 way it is possible for the dynamic linker to know with certainty that
2290 all external symbols that it needs will be resolvable without having to
2291 search for each symbol reference.

2292
2293 The symbol versioning is in effect a much more sophisticated way of
2294 doing minor version checking that SunOS does. The fundamental problem
2295 that is being addressed here is that typically references to external
2296 functions are bound on an as-needed basis, and are not all bound when
2297 the application starts up. If a shared library is out of date, a
2298 required interface may be missing; when the application tries to use
2299 that interface, it may suddenly and unexpectedly fail. With symbol
2300 versioning, the user will get a warning when they start their program if
2301 the libraries being used with the application are too old.

2303 There are several GNU extensions to Sun's versioning approach. The
2304 first of these is the ability to bind a symbol to a version node in the
2305 source file where the symbol is defined instead of in the versioning
2306 script. This was done mainly to reduce the burden on the library
2307 maintainer. This can be done by putting something like:

```
2308  
2309       __asm__ (".symver original_foo,foo@VERS_1.1");
```

2310
2311 in the C source file. This renamed the function 'original_foo' to
2312 be an alias for 'foo' bound to the version node 'VERS_1.1'. The
2313 'local:' directive can be used to prevent the symbol 'original_foo'
2314 from being exported.

2315
2316 The second GNU extension is to allow multiple versions of the same
2317 function to appear in a given shared library. In this way an
2318 incompatible change to an interface can take place without increasing
2319 the major version number of the shared library, while still allowing
2320 applications linked against the old interface to continue to function.

2321
2322 This can only be accomplished by using multiple '.symver' directives
2323 in the assembler. An example of this would be:

```
2324  
2325       __asm__ (".symver original_foo,foo@");  
2326       __asm__ (".symver old_foo,foo@VERS_1.1");  
2327       __asm__ (".symver old_fool,foo@VERS_1.2");  
2328       __asm__ (".symver new_foo,foo@@VERS_2.0");
```

2329
2330 In this example, 'foo@' represents the symbol 'foo' bound to the
2331 unspecified base version of the symbol. The source file that contains
2332 this example would define 4 C functions: 'original_foo', 'old_foo',
2333 'old_fool', and 'new_foo'.

2334
2335 When you have multiple definitions of a given symbol, there needs to
2336 be some way to specify a default version to which external references to
2337 this symbol will be bound. This can be accomplished with the
2338 'foo@@VERS_2.0' type of '.symver' directive. Only one version of a
2339 symbol can be declared 'default' in this manner - otherwise you would
2340 effectively have multiple definitions of the same symbol.

2341
2342 If you wish to bind a reference to a specific version of the symbol
2343 within the shared library, you can use the aliases of convenience (i.e.
2344 'old_foo'), or you can use the '.symver' directive to specifically bind
2345 to an external version of the function in question.

2346 **US**

2347 File: ld.info, Node: Option Commands, Prev: Version Script, Up: Commands

2348
2349
2350 Option Commands

2351 =====

2352
2353 The command language includes a number of other commands that you can
2354 use for specialized purposes. They are similar in purpose to
2355 command-line options.

2356
2357 'CONSTRUCTORS'

2358 When linking using the 'a.out' object file format, the linker uses
2359 an unusual set construct to support C++ global constructors and
2360 destructors. When linking object file formats which do not support
2361 arbitrary sections, such as 'ECOFF' and 'XCOFF', the linker will
2362 automatically recognize C++ global constructors and destructors by
2363 name. For these object file formats, the 'CONSTRUCTORS' command
2364 tells the linker where this information should be placed. The
2365 'CONSTRUCTORS' command is ignored for other object file formats.

2366
2367 The symbol '__CTOR_LIST__' marks the start of the global
2368 constructors, and the symbol '__DTOR_LIST' marks the end. The

first word in the list is the number of entries, followed by the address of each constructor or destructor, followed by a zero word. The compiler must arrange to actually run the code. For these object file formats GNU C++ calls constructors from a subroutine ``__main'`; a call to ``__main'` is automatically inserted into the startup code for ``main'`. GNU C++ runs destructors either by using ``atexit'`, or directly from the function ``exit'`.

For object file formats such as ``COFF'` or ``ELF'` which support multiple sections, GNU C++ will normally arrange to put the addresses of global constructors and destructors into the ``.ctors'` and ``.dtors'` sections. Placing the following sequence into your linker script will build the sort of table which the GNU C++ runtime code expects to see.

```
__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__ = .;
```

Normally the compiler and linker will handle these issues automatically, and you will not need to concern yourself with them. However, you may need to consider this if you are using C++ and writing your own linker scripts.

``FLOAT'`

``NOFLOAT'`

These keywords were used in some older linkers to request a particular math subroutine library. ``ld'` doesn't use the keywords, assuming instead that any necessary subroutines are in libraries specified using the general mechanisms for linking to archives; but to permit the use of scripts that were written for the older linkers, the keywords ``FLOAT'` and ``NOFLOAT'` are accepted and ignored.

``FORCE_COMMON_ALLOCATION'`

This command has the same effect as the ``-d'` command-line option: to make ``ld'` assign space to common symbols even if a relocatable output file is specified (``-r'`).

``INCLUDE FILENAME'`

Include the linker script FILENAME 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.

``INPUT (FILE, FILE, ...)'`

``INPUT (FILE FILE ...)'`

Use this command to include binary input files in the link, without including them in a particular section definition. Specify the full name for each FILE, including ``.a'` if required.

``ld'` searches for each FILE through the archive-library search path, just as for files you specify on the command line. See the description of ``-L'` in *Note Command Line Options: Options.

If you use ``-lFILE'`, ``ld'` will transform the name to ``libFILE.a'` as with the command line argument ``-l'`.

``GROUP (FILE, FILE, ...)'`

```

2435 `GROUP ( FILE FILE ... )'
2436     This command is like `INPUT', except that the named files should
2437     all be archives, and they are searched repeatedly until no new
2438     undefined references are created. See the description of `-' in
2439     *Note Command Line Options: Options.
2440
2441 `OUTPUT ( FILENAME )'
2442     Use this command to name the link output file FILENAME. The
2443     effect of `OUTPUT(FILENAME)' is identical to the effect of
2444     `-o FILENAME', which overrides it. You can use this command to
2445     supply a default output-file name other than `a.out'.
2446
2447 `OUTPUT_ARCH ( BFDNAME )'
2448     Specify a particular output machine architecture, with one of the
2449     names used by the BFD back-end routines (*note BFD:..). This
2450     command is often unnecessary; the architecture is most often set
2451     implicitly by either the system BFD configuration or as a side
2452     effect of the `OUTPUT_FORMAT' command.
2453
2454 `OUTPUT_FORMAT ( BFDNAME )'
2455     When `ld' is configured to support multiple object code formats,
2456     you can use this command to specify a particular output format.
2457     BFDNAME is one of the names used by the BFD back-end routines
2458     (*note BFD:..). The effect is identical to the effect of the
2459     `--oformat' command-line option. This selection affects only the
2460     output file; the related command `TARGET' affects primarily input
2461     files.
2462
2463 `SEARCH_DIR ( PATH )'
2464     Add PATH to the list of paths where `ld' looks for archive
2465     libraries. `SEARCH_DIR(PATH)' has the same effect as `-LPATH' on
2466     the command line.
2467
2468 `STARTUP ( FILENAME )'
2469     Ensure that FILENAME is the first input file used in the link
2470     process.
2471
2472 `TARGET ( FORMAT )'
2473     When `ld' is configured to support multiple object code formats,
2474     you can use this command to change the input-file object code
2475     format (like the command-line option `-b' or its synonym
2476     `--format'). The argument FORMAT is one of the strings used by
2477     BFD to name binary formats. If `TARGET' is specified but
2478     `OUTPUT_FORMAT' is not, the last `TARGET' argument is also used as
2479     the default format for the `ld' output file. *Note BFD:..
2480
2481     If you don't use the `TARGET' command, `ld' uses the value of the
2482     environment variable `GNUTARGET', if available, to select the
2483     output file format. If that variable is also absent, `ld' uses
2484     the default format configured for your machine in the BFD
2485     libraries.
2486
2487 `NOCROSSREFS ( SECTION SECTION ... )'
2488     This command may be used to tell `ld' to issue an error about any
2489     references among certain sections.
2490
2491     In certain types of programs, particularly on embedded systems,
2492     when one section is loaded into memory, another section will not
2493     be. Any direct references between the two sections would be
2494     errors. For example, it would be an error if code in one section
2495     called a function defined in the other section.
2496
2497     The `NOCROSSREFS' command takes a list of section names. If `ld'
2498     detects any cross references between the sections, it reports an
2499     error and returns a non-zero exit status. The `NOCROSSREFS'
2500     command uses output section names, defined in the `SECTIONS'

```

```

2501         command.  It does not use the names of input sections.
2502
2503 US
2504 File: ld.info,  Node: Machine Dependent,  Next: BFD,  Prev: Commands,  Up: Top
2505
2506 Machine Dependent Features
2507 *****
2508
2509     `ld' has additional features on some platforms; the following
2510 sections describe them.  Machines where `ld' has no additional
2511 functionality are not listed.
2512
2513 * Menu:
2514
2515 * H8/300::                `ld' and the H8/300
2516 * i960::                  `ld' and the Intel 960 family
2517
2518 US
2519 File: ld.info,  Node: H8/300,  Next: i960,  Up: Machine Dependent
2520
2521 `ld' and the H8/300
2522 =====
2523
2524     For the H8/300, `ld' can perform these global optimizations when you
2525 specify the `--relax' command-line option.
2526
2527 *relaxing address modes*
2528     `ld' finds all `jsr' and `jmp' instructions whose targets are
2529 within eight bits, and turns them into eight-bit program-counter
2530 relative `bsr' and `bra' instructions, respectively.
2531
2532 *synthesizing instructions*
2533     `ld' finds all `mov.b' instructions which use the sixteen-bit
2534 absolute address form, but refer to the top page of memory, and
2535 changes them to use the eight-bit address form.  (That is: the
2536 linker turns `mov.b `@'AA:16' into `mov.b `@'AA:8' whenever the
2537 address AA is in the top page of memory).
2538
2539 US
2540 File: ld.info,  Node: i960,  Prev: H8/300,  Up: Machine Dependent
2541
2542 `ld' and the Intel 960 family
2543 =====
2544
2545     You can use the `-AARCHITECTURE' command line option to specify one
2546 of the two-letter names identifying members of the 960 family; the
2547 option specifies the desired output target, and warns of any
2548 incompatible instructions in the input files.  It also modifies the
2549 linker's search strategy for archive libraries, to support the use of
2550 libraries specific to each particular architecture, by including in the
2551 search loop names suffixed with the string identifying the architecture.
2552
2553     For example, if your `ld' command line included `-ACA' as well as
2554 `-ltry', the linker would look (in its built-in search paths, and in
2555 any paths you specify with `-L') for a library with the names
2556
2557         try
2558         libtry.a
2559         tryca
2560         libtryca.a
2561
2562     The first two possibilities would be considered in any event; the last
2563 two are due to the use of `-ACA'.
2564
2565     You can meaningfully use `-A' more than once on a command line, since
2566 the 960 architecture family allows combination of target architectures;

```

```

2567 each use will add another pair of name variants to search for when '-l'
2568 specifies a library.
2569
2570 'ld' supports the '--relax' option for the i960 family. If you
2571 specify '--relax', 'ld' finds all 'balx' and 'calx' instructions whose
2572 targets are within 24 bits, and turns them into 24-bit program-counter
2573 relative 'bal' and 'cal' instructions, respectively. 'ld' also turns
2574 'cal' instructions into 'bal' instructions when it determines that the
2575 target subroutine is a leaf routine (that is, the target subroutine does
2576 not itself call any subroutines).
2577
2578 US
2579 File: ld.info, Node: BFD, Next: Reporting Bugs, Prev: Machine Dependent, Up: Top
2580
2581 BFD
2582 ***
2583
2584 The linker accesses object and archive files using the BFD libraries.
2585 These libraries allow the linker to use the same routines to operate on
2586 object files whatever the object file format. A different object file
2587 format can be supported simply by creating a new BFD back end and adding
2588 it to the library. To conserve runtime memory, however, the linker and
2589 associated tools are usually configured to support only a subset of the
2590 object file formats available. You can use 'objdump -i' (*note
2591 objdump: (binutils.info)objdump.) to list all the formats available for
2592 your configuration.
2593
2594 As with most implementations, BFD is a compromise between several
2595 conflicting requirements. The major factor influencing BFD design was
2596 efficiency: any time used converting between formats is time which
2597 would not have been spent had BFD not been involved. This is partly
2598 offset by abstraction payback; since BFD simplifies applications and
2599 back ends, more time and care may be spent optimizing algorithms for a
2600 greater speed.
2601
2602 One minor artifact of the BFD solution which you should bear in mind
2603 is the potential for information loss. There are two places where
2604 useful information can be lost using the BFD mechanism: during
2605 conversion and during output. *Note BFD information loss::.
2606
2607 * Menu:
2608
2609 * BFD outline:: How it works: an outline of BFD
2610
2611 US
2612 File: ld.info, Node: BFD outline, Up: BFD
2613
2614 How it works: an outline of BFD
2615 =====
2616
2617 When an object file is opened, BFD subroutines automatically
2618 determine the format of the input object file. They then build a
2619 descriptor in memory with pointers to routines that will be used to
2620 access elements of the object file's data structures.
2621
2622 As different information from the the object files is required, BFD
2623 reads from different sections of the file and processes them. For
2624 example, a very common operation for the linker is processing symbol
2625 tables. Each BFD back end provides a routine for converting between
2626 the object file's representation of symbols and an internal canonical
2627 format. When the linker asks for the symbol table of an object file, it
2628 calls through a memory pointer to the routine from the relevant BFD
2629 back end which reads and converts the table into a canonical form. The
2630 linker then operates upon the canonical form. When the link is finished
2631 and the linker writes the output file's symbol table, another BFD back
2632 end routine is called to take the newly created symbol table and

```

```

2633 convert it into the chosen output format.
2634
2635 * Menu:
2636
2637 * BFD information loss::      Information Loss
2638 * Canonical format::         The BFD canonical object-file format
2639
2640 US
2641 File: ld.info,  Node: BFD information loss,  Next: Canonical format,  Up: BFD outline
2642
2643 Information Loss
2644 -----
2645
2646 *Information can be lost during output.* The output formats
2647 supported by BFD do not provide identical facilities, and information
2648 which can be described in one form has nowhere to go in another format.
2649 One example of this is alignment information in `b.out'. There is
2650 nowhere in an `a.out' format file to store alignment information on the
2651 contained data, so when a file is linked from `b.out' and an `a.out'
2652 image is produced, alignment information will not propagate to the
2653 output file. (The linker will still use the alignment information
2654 internally, so the link is performed correctly).
2655
2656 Another example is COFF section names. COFF files may contain an
2657 unlimited number of sections, each one with a textual section name. If
2658 the target of the link is a format which does not have many sections
2659 (e.g., `a.out') or has sections without names (e.g., the Oasys format),
2660 the link cannot be done simply. You can circumvent this problem by
2661 describing the desired input-to-output section mapping with the linker
2662 command language.
2663
2664 *Information can be lost during canonicalization.* The BFD internal
2665 canonical form of the external formats is not exhaustive; there are
2666 structures in input formats for which there is no direct representation
2667 internally. This means that the BFD back ends cannot maintain all
2668 possible data richness through the transformation between external to
2669 internal and back to external formats.
2670
2671 This limitation is only a problem when an application reads one
2672 format and writes another. Each BFD back end is responsible for
2673 maintaining as much data as possible, and the internal BFD canonical
2674 form has structures which are opaque to the BFD core, and exported only
2675 to the back ends. When a file is read in one format, the canonical form
2676 is generated for BFD and the application. At the same time, the back
2677 end saves away any information which may otherwise be lost. If the data
2678 is then written back in the same format, the back end routine will be
2679 able to use the canonical form provided by the BFD core as well as the
2680 information it prepared earlier. Since there is a great deal of
2681 commonality between back ends, there is no information lost when
2682 linking or copying big endian COFF to little endian COFF, or `a.out' to
2683 `b.out'. When a mixture of formats is linked, the information is only
2684 lost from the files whose format differs from the destination.
2685
2686 US
2687 File: ld.info,  Node: Canonical format,  Prev: BFD information loss,  Up: BFD outline
2688
2689 The BFD canonical object-file format
2690 -----
2691
2692 The greatest potential for loss of information occurs when there is
2693 the least overlap between the information provided by the source
2694 format, that stored by the canonical format, and that needed by the
2695 destination format. A brief description of the canonical form may help
2696 you understand which kinds of data you can count on preserving across
2697 conversions.
2698

```

```

2699 *files*
2700     Information stored on a per-file basis includes target machine
2701     architecture, particular implementation format type, a demand
2702     pageable bit, and a write protected bit. Information like Unix
2703     magic numbers is not stored here--only the magic numbers' meaning,
2704     so a `ZMAGIC' file would have both the demand pageable bit and the
2705     write protected text bit set. The byte order of the target is
2706     stored on a per-file basis, so that big- and little-endian object
2707     files may be used with one another.
2708
2709 *sections*
2710     Each section in the input file contains the name of the section,
2711     the section's original address in the object file, size and
2712     alignment information, various flags, and pointers into other BFD
2713     data structures.
2714
2715 *symbols*
2716     Each symbol contains a pointer to the information for the object
2717     file which originally defined it, its name, its value, and various
2718     flag bits. When a BFD back end reads in a symbol table, it
2719     relocates all symbols to make them relative to the base of the
2720     section where they were defined. Doing this ensures that each
2721     symbol points to its containing section. Each symbol also has a
2722     varying amount of hidden private data for the BFD back end. Since
2723     the symbol points to the original file, the private data format
2724     for that symbol is accessible. `ld' can operate on a collection
2725     of symbols of wildly different formats without problems.
2726
2727     Normal global and simple local symbols are maintained on output,
2728     so an output file (no matter its format) will retain symbols
2729     pointing to functions and to global, static, and common variables.
2730     Some symbol information is not worth retaining; in `a.out', type
2731     information is stored in the symbol table as long symbol names.
2732     This information would be useless to most COFF debuggers; the
2733     linker has command line switches to allow users to throw it away.
2734
2735     There is one word of type information within the symbol, so if the
2736     format supports symbol type information within symbols (for
2737     example, COFF, IEEE, Oasys) and the type is simple enough to fit
2738     within one word (nearly everything but aggregates), the
2739     information will be preserved.
2740
2741 *relocation level*
2742     Each canonical BFD relocation record contains a pointer to the
2743     symbol to relocate to, the offset of the data to relocate, the
2744     section the data is in, and a pointer to a relocation type
2745     descriptor. Relocation is performed by passing messages through
2746     the relocation type descriptor and the symbol pointer. Therefore,
2747     relocations can be performed on output data using a relocation
2748     method that is only available in one of the input formats. For
2749     instance, Oasys provides a byte relocation format. A relocation
2750     record requesting this relocation type would point indirectly to a
2751     routine to perform this, so the relocation may be performed on a
2752     byte being written to a 68k COFF file, even though 68k COFF has no
2753     such relocation type.
2754
2755 *line numbers*
2756     Object formats can contain, for debugging purposes, some form of
2757     mapping between symbols, source line numbers, and addresses in the
2758     output file. These addresses have to be relocated along with the
2759     symbol information. Each symbol with an associated list of line
2760     number records points to the first record of the list. The head
2761     of a line number list consists of a pointer to the symbol, which
2762     allows finding out the address of the function whose line number
2763     is being described. The rest of the list is made up of pairs:
2764     offsets into the section and line numbers. Any format which can

```

```

2765         simply derive this information can pass it successfully between
2766         formats (COFF, IEEE and Oasys).
2767
2768 US
2769 File: ld.info,  Node: Reporting Bugs,  Next: MRI,  Prev: BFD,  Up: Top
2770
2771 Reporting Bugs
2772 *****
2773
2774     Your bug reports play an essential role in making `ld' reliable.
2775
2776     Reporting a bug may help you by bringing a solution to your problem,
2777     or it may not.  But in any case the principal function of a bug report
2778     is to help the entire community by making the next version of `ld' work
2779     better.  Bug reports are your contribution to the maintenance of `ld'.
2780
2781     In order for a bug report to serve its purpose, you must include the
2782     information that enables us to fix the bug.
2783
2784     * Menu:
2785
2786     * Bug Criteria::                Have you found a bug?
2787     * Bug Reporting::              How to report bugs
2788
2789 US
2790 File: ld.info,  Node: Bug Criteria,  Next: Bug Reporting,  Up: Reporting Bugs
2791
2792 Have you found a bug?
2793 =====
2794
2795     If you are not sure whether you have found a bug, here are some
2796     guidelines:
2797
2798     * If the linker gets a fatal signal, for any input whatever, that is
2799     a `ld' bug.  Reliable linkers never crash.
2800
2801     * If `ld' produces an error message for valid input, that is a bug.
2802
2803     * If `ld' does not produce an error message for invalid input, that
2804     may be a bug.  In the general case, the linker can not verify that
2805     object files are correct.
2806
2807     * If you are an experienced user of linkers, your suggestions for
2808     improvement of `ld' are welcome in any case.
2809
2810 US
2811 File: ld.info,  Node: Bug Reporting,  Prev: Bug Criteria,  Up: Reporting Bugs
2812
2813 How to report bugs
2814 =====
2815
2816     A number of companies and individuals offer support for GNU
2817     products.  If you obtained `ld' from a support organization, we
2818     recommend you contact that organization first.
2819
2820     You can find contact information for many support companies and
2821     individuals in the file `etc/SERVICE' in the GNU Emacs distribution.
2822
2823     In any event, we also recommend that you send bug reports for `ld'
2824     to `bug-gnu-utils@gnu.org'.
2825
2826     The fundamental principle of reporting bugs usefully is this:
2827     *report all the facts*.  If you are not sure whether to state a fact or
2828     leave it out, state it!
2829
2830     Often people omit facts because they think they know what causes the

```

2831 problem and assume that some details do not matter. Thus, you might
2832 assume that the name of a symbol you use in an example does not matter.
2833 Well, probably it does not, but one cannot be sure. Perhaps the bug is
2834 a stray memory reference which happens to fetch from the location where
2835 that name is stored in memory; perhaps, if the name were different, the
2836 contents of that location would fool the linker into doing the right
2837 thing despite the bug. Play it safe and give a specific, complete
2838 example. That is the easiest thing for you to do, and the most helpful.
2839

2840 Keep in mind that the purpose of a bug report is to enable us to fix
2841 the bug if it is new to us. Therefore, always write your bug reports
2842 on the assumption that the bug has not been reported previously.
2843

2844 Sometimes people give a few sketchy facts and ask, "Does this ring a
2845 bell?" Those bug reports are useless, and we urge everyone to *refuse
2846 to respond to them* except to chide the sender to report bugs properly.
2847

2848 To enable us to fix the bug, you should include all these things:
2849

2850 * The version of `ld'. `ld' announces it if you start it with the
2851 `--version' argument.
2852

2853 Without this, we will not know whether there is any point in
2854 looking for the bug in the current version of `ld'.
2855

2856 * Any patches you may have applied to the `ld' source, including any
2857 patches made to the `BFD' library.
2858

2859 * The type of machine you are using, and the operating system name
2860 and version number.
2861

2862 * What compiler (and its version) was used to compile `ld'--e.g.
2863 "`gcc-2.7'".
2864

2865 * The command arguments you gave the linker to link your example and
2866 observe the bug. To guarantee you will not omit something
2867 important, list them all. A copy of the Makefile (or the output
2868 from make) is sufficient.
2869

2870 If we were to try to guess the arguments, we would probably guess
2871 wrong and then we might not encounter the bug.
2872

2873 * A complete input file, or set of input files, that will reproduce
2874 the bug. It is generally most helpful to send the actual object
2875 files, uuencoded if necessary to get them through the mail system.
2876 Making them available for anonymous FTP is not as good, but may
2877 be the only reasonable choice for large object files.
2878

2879 If the source files were assembled using `gas' or compiled using
2880 `gcc', then it may be OK to send the source files rather than the
2881 object files. In this case, be sure to say exactly what version of
2882 `gas' or `gcc' was used to produce the object files. Also say how
2883 `gas' or `gcc' were configured.
2884

2885 * A description of what behavior you observe that you believe is
2886 incorrect. For example, "It gets a fatal signal."
2887

2888 Of course, if the bug is that `ld' gets a fatal signal, then we
2889 will certainly notice it. But if the bug is incorrect output, we
2890 might not notice unless it is glaringly wrong. You might as well
2891 not give us a chance to make a mistake.
2892

2893 Even if the problem you experience is a fatal signal, you should
2894 still say so explicitly. Suppose something strange is going on,
2895 such as, your copy of `ld' is out of synch, or you have
2896 encountered a bug in the C library on your system. (This has

2897 happened!) Your copy might crash and ours would not. If you told
 2898 us to expect a crash, then when ours fails to crash, we would know
 2899 that the bug was not happening for us. If you had not told us to
 2900 expect a crash, then we would not be able to draw any conclusion
 2901 from our observations.
 2902

2903 * If you wish to suggest changes to the 'ld' source, send us context
 2904 diffs, as generated by 'diff' with the '-u', '-c', or '-p' option.
 2905 Always send diffs from the old file to the new file. If you even
 2906 discuss something in the 'ld' source, refer to it by context, not
 2907 by line number.
 2908

2909 The line numbers in our development sources will not match those
 2910 in your sources. Your line numbers would convey no useful
 2911 information to us.
 2912

2913 Here are some things that are not necessary:
 2914

2915 * A description of the envelope of the bug.
 2916

2917 Often people who encounter a bug spend a lot of time investigating
 2918 which changes to the input file will make the bug go away and which
 2919 changes will not affect it.
 2920

2921 This is often time consuming and not very useful, because the way
 2922 we will find the bug is by running a single example under the
 2923 debugger with breakpoints, not by pure deduction from a series of
 2924 examples. We recommend that you save your time for something else.
 2925

2926 Of course, if you can find a simpler example to report *instead*
 2927 of the original one, that is a convenience for us. Errors in the
 2928 output will be easier to spot, running under the debugger will take
 2929 less time, and so on.
 2930

2931 However, simplification is not vital; if you do not want to do
 2932 this, report the bug anyway and send us the entire test case you
 2933 used.
 2934

2935 * A patch for the bug.
 2936

2937 A patch for the bug does help us if it is a good one. But do not
 2938 omit the necessary information, such as the test case, on the
 2939 assumption that a patch is all we need. We might see problems
 2940 with your patch and decide to fix the problem another way, or we
 2941 might not understand it at all.
 2942

2943 Sometimes with a program as complicated as 'ld' it is very hard to
 2944 construct an example that will make the program follow a certain
 2945 path through the code. If you do not send us the example, we will
 2946 not be able to construct one, so we will not be able to verify
 2947 that the bug is fixed.
 2948

2949 And if we cannot understand what bug you are trying to fix, or why
 2950 your patch should be an improvement, we will not install it. A
 2951 test case will help us to understand.
 2952

2953 * A guess about what the bug is or what it depends on.
 2954

2955 Such guesses are usually wrong. Even we cannot guess right about
 2956 such things without first using the debugger to find the facts.
 2957



2958 File: ld.info, Node: MRI, Next: Index, Prev: Reporting Bugs, Up: Top
 2959

2960 MRI Compatible Script Files
 2961

2962 *****

```

2963
2964     To aid users making the transition to GNU `ld' from the MRI linker,
2965 `ld' can use MRI compatible linker scripts as an alternative to the
2966 more general-purpose linker scripting language described in *Note
2967 Command Language: Commands. MRI compatible linker scripts have a much
2968 simpler command set than the scripting language otherwise used with
2969 `ld'. GNU `ld' supports the most commonly used MRI linker commands;
2970 these commands are described here.
2971
2972     In general, MRI scripts aren't of much use with the `a.out' object
2973 file format, since it only has three sections and MRI scripts lack some
2974 features to make use of them.
2975
2976     You can specify a file containing an MRI-compatible script using the
2977 `-c' command-line option.
2978
2979     Each command in an MRI-compatible script occupies its own line; each
2980 command line starts with the keyword that identifies the command (though
2981 blank lines are also allowed for punctuation). If a line of an
2982 MRI-compatible script begins with an unrecognized keyword, `ld' issues
2983 a warning message, but continues processing the script.
2984
2985     Lines beginning with `*' are comments.
2986
2987     You can write these commands using all upper-case letters, or all
2988 lower case; for example, `chip' is the same as `CHIP'. The following
2989 list shows only the upper-case form of each command.
2990
2991 `ABSOLUTE SECNAME'
2992 `ABSOLUTE SECNAME, SECNAME, ... SECNAME'
2993     Normally, `ld' includes in the output file all sections from all
2994 the input files. However, in an MRI-compatible script, you can
2995 use the `ABSOLUTE' command to restrict the sections that will be
2996 present in your output program. If the `ABSOLUTE' command is used
2997 at all in a script, then only the sections named explicitly in
2998 `ABSOLUTE' commands will appear in the linker output. You can
2999 still use other input sections (whatever you select on the command
3000 line, or using `LOAD') to resolve addresses in the output file.
3001
3002 `ALIAS OUT-SECNAME, IN-SECNAME'
3003     Use this command to place the data from input section IN-SECNAME
3004 in a section called OUT-SECNAME in the linker output file.
3005
3006     IN-SECNAME may be an integer.
3007
3008 `ALIGN SECNAME = EXPRESSION'
3009     Align the section called SECNAME to EXPRESSION. The EXPRESSION
3010 should be a power of two.
3011
3012 `BASE EXPRESSION'
3013     Use the value of EXPRESSION as the lowest address (other than
3014 absolute addresses) in the output file.
3015
3016 `CHIP EXPRESSION'
3017 `CHIP EXPRESSION, EXPRESSION'
3018     This command does nothing; it is accepted only for compatibility.
3019
3020 `END'
3021     This command does nothing whatever; it's only accepted for
3022 compatibility.
3023
3024 `FORMAT OUTPUT-FORMAT'
3025     Similar to the `OUTPUT_FORMAT' command in the more general linker
3026 language, but restricted to one of these output formats:
3027
3028     1. S-records, if OUTPUT-FORMAT is `S'

```

```

3029
3030     2. IEEE, if OUTPUT-FORMAT is `IEEE'
3031
3032     3. COFF (the `coff-m68k' variant in BFD), if OUTPUT-FORMAT is
3033         `COFF'
3034
3035 `LIST ANYTHING...'
3036     Print (to the standard output file) a link map, as produced by the
3037     `ld' command-line option `-M'.
3038
3039     The keyword `LIST' may be followed by anything on the same line,
3040     with no change in its effect.
3041
3042 `LOAD FILENAME'
3043 `LOAD FILENAME, FILENAME, ... FILENAME'
3044     Include one or more object file FILENAME in the link; this has the
3045     same effect as specifying FILENAME directly on the `ld' command
3046     line.
3047
3048 `NAME OUTPUT-NAME'
3049     OUTPUT-NAME is the name for the program produced by `ld'; the
3050     MRI-compatible command `NAME' is equivalent to the command-line
3051     option `-o' or the general script language command `OUTPUT'.
3052
3053 `ORDER SECNAME, SECNAME, ... SECNAME'
3054 `ORDER SECNAME SECNAME SECNAME'
3055     Normally, `ld' orders the sections in its output file in the order
3056     in which they first appear in the input files. In an
3057     MRI-compatible script, you can override this ordering with the
3058     `ORDER' command. The sections you list with `ORDER' will appear
3059     first in your output file, in the order specified.
3060
3061 `PUBLIC NAME=EXPRESSION'
3062 `PUBLIC NAME,EXPRESSION'
3063 `PUBLIC NAME EXPRESSION'
3064     Supply a value (EXPRESSION) for external symbol NAME used in the
3065     linker input files.
3066
3067 `SECT SECNAME, EXPRESSION'
3068 `SECT SECNAME=EXPRESSION'
3069 `SECT SECNAME EXPRESSION'
3070     You can use any of these three forms of the `SECT' command to
3071     specify the start address (EXPRESSION) for section SECNAME. If
3072     you have more than one `SECT' statement for the same SECNAME, only
3073     the *first* sets the start address.
3074
3075 US
3076 File: ld.info, Node: Index, Prev: MRI, Up: Top
3077
3078 Index
3079 *****
3080
3081 * Menu:
3082
3083 * ": Symbols.
3084 * *( COMMON ): Section Placement.
3085 * *(SECTION): Section Placement.
3086 * -( : Options.
3087 * --relax on i960: i960.
3088 * -AARCH: Options.
3089 * -aKEYWORD: Options.
3090 * -architecture=ARCH: Options.
3091 * -assert KEYWORD: Options.
3092 * -auxiliary: Options.
3093 * -b FORMAT: Options.
3094 * -Bdynamic: Options.

```

3095	* -Bshareable:	Options.
3096	* -Bstatic:	Options.
3097	* -Bsymbolic:	Options.
3098	* -c MRI-CMDFILE:	Options.
3099	* -call_shared:	Options.
3100	* -cref:	Options.
3101	* -d:	Options.
3102	* -dc:	Options.
3103	* -defsym SYMBOL=EXP:	Options.
3104	* -discard-all:	Options.
3105	* -discard-locals:	Options.
3106	* -dn:	Options.
3107	* -dp:	Options.
3108	* -dy:	Options.
3109	* -dynamic-linker FILE:	Options.
3110	* -E:	Options.
3111	* -e ENTRY:	Options.
3112	* -EB:	Options.
3113	* -EL:	Options.
3114	* -embedded-relocs:	Options.
3115	* -entry=ENTRY:	Options.
3116	* -export-dynamic:	Options.
3117	* -f:	Options.
3118	* -F:	Options.
3119	* -filter:	Options.
3120	* -force-exe-suffix:	Options.
3121	* -format=FORMAT:	Options.
3122	* -G:	Options.
3123	* -g:	Options.
3124	* -gpsize:	Options.
3125	* -help:	Options.
3126	* -hNAME:	Options.
3127	* -i:	Options.
3128	* -just-symbols=FILE:	Options.
3129	* -lARCHIVE:	Options.
3130	* -LDIR:	Options.
3131	* -library-path=DIR:	Options.
3132	* -library=ARCHIVE:	Options.
3133	* -M:	Options.
3134	* -m EMULATION:	Options.
3135	* -Map:	Options.
3136	* -mri-script=MRI-CMDFILE:	Options.
3137	* -N:	Options.
3138	* -n:	Options.
3139	* -nmagic:	Options.
3140	* -no-keep-memory:	Options.
3141	* -no-warn-mismatch:	Options.
3142	* -no-whole-archive:	Options.
3143	* -noinhibit-exec:	Options.
3144	* -non_shared:	Options.
3145	* -o OUTPUT:	Options.
3146	* -oformat:	Options.
3147	* -omagic:	Options.
3148	* -output=OUTPUT:	Options.
3149	* -print-map:	Options.
3150	* -qmagic:	Options.
3151	* -Qy:	Options.
3152	* -r:	Options.
3153	* -R FILE:	Options.
3154	* -relax:	Options.
3155	* -relocateable:	Options.
3156	* -rpath:	Options.
3157	* -rpath-link:	Options.
3158	* -s:	Options.
3159	* -S:	Options.
3160	* -script=SCRIPT:	Options.

3161	* -shared:	Options.
3162	* -soname=NAME:	Options.
3163	* -sort-common:	Options.
3164	* -split-by-file:	Options.
3165	* -split-by-reloc:	Options.
3166	* -static:	Options.
3167	* -stats:	Options.
3168	* -strip-all:	Options.
3169	* -strip-debug:	Options.
3170	* -t:	Options.
3171	* -T SCRIPT:	Options.
3172	* -Tbss ORG:	Options.
3173	* -Tdata ORG:	Options.
3174	* -trace:	Options.
3175	* -trace-symbol=SYMBOL:	Options.
3176	* -traditional-format:	Options.
3177	* -Ttext ORG:	Options.
3178	* -u SYMBOL:	Options.
3179	* -undefined=SYMBOL:	Options.
3180	* -Ur:	Options.
3181	* -V:	Options.
3182	* -v:	Options.
3183	* -verbose:	Options.
3184	* -version:	Options.
3185	* -version-script=VERSION-SCRIPTFILE:	Options.
3186	* -warn-comon:	Options.
3187	* -warn-constructors:	Options.
3188	* -warn-multiple-gp:	Options.
3189	* -warn-once:	Options.
3190	* -warn-section-align:	Options.
3191	* -whole-archive:	Options.
3192	* -wrap:	Options.
3193	* -X:	Options.
3194	* -x:	Options.
3195	* -Y PATH:	Options.
3196	* -y SYMBOL:	Options.
3197	* -z KEYWORD:	Options.
3198	* .:	Location Counter.
3199	* 0x:	Integers.
3200	* :PHDR:	Section Options.
3201	* ;:	Assignment.
3202	* =FILL:	Section Options.
3203	* >REGION:	Section Options.
3204	* [SECTION...], not supported:	Section Placement.
3205	* ABSOLUTE (MRI):	MRI.
3206	* absolute and relocatable symbols:	Assignment.
3207	* ABSOLUTE (EXP):	Arithmetic Functions.
3208	* ADDR (SECTION):	Arithmetic Functions.
3209	* ALIAS (MRI):	MRI.
3210	* ALIGN (MRI):	MRI.
3211	* ALIGN (EXP):	Arithmetic Functions.
3212	* aligning sections:	Section Options.
3213	* allocating memory:	MEMORY.
3214	* architectures:	Options.
3215	* archive files, from cmd line:	Options.
3216	* arithmetic:	Expressions.
3217	* arithmetic operators:	Operators.
3218	* assignment in scripts:	Assignment.
3219	* assignment, in section defn:	Section Data Expressions.
3220	* AT (LDADR):	Section Options.
3221	* back end:	BFD.
3222	* BASE (MRI):	MRI.
3223	* BFD canonical format:	Canonical format.
3224	* BFD requirements:	BFD.
3225	* big-endian objects:	Options.
3226	* binary input files:	Option Commands.

3227	* binary input format:	Options.
3228	* BLOCK(ALIGN):	Section Options.
3229	* bug criteria:	Bug Criteria.
3230	* bug reports:	Bug Reporting.
3231	* bugs in ld:	Reporting Bugs.
3232	* BYTE(EXPRESSION):	Section Data Expressions.
3233	* C++ constructors, arranging in link:	Option Commands.
3234	* CHIP (MRI):	MRI.
3235	* combining symbols, warnings on:	Options.
3236	* command files:	Commands.
3237	* command line:	Options.
3238	* commands, fundamental:	Scripts.
3239	* comments:	Scripts.
3240	* common allocation <1>:	Options.
3241	* common allocation:	Option Commands.
3242	* commons in output:	Section Placement.
3243	* compatibility, MRI:	Options.
3244	* CONSTRUCTORS:	Option Commands.
3245	* constructors:	Options.
3246	* constructors, arranging in link:	Option Commands.
3247	* contents of a section:	Section Placement.
3248	* crash of linker:	Bug Criteria.
3249	* CREATE_OBJECT_SYMBOLS:	Section Data Expressions.
3250	* cross reference table:	Options.
3251	* cross references:	Option Commands.
3252	* current output location:	Location Counter.
3253	* dbx:	Options.
3254	* decimal integers:	Integers.
3255	* default emulation:	Environment.
3256	* default input format:	Environment.
3257	* DEFINED(SYMBOL):	Arithmetic Functions.
3258	* deleting local symbols:	Options.
3259	* direct output:	Section Data Expressions.
3260	* discontinuous memory:	MEMORY.
3261	* dot:	Location Counter.
3262	* dynamic linker, from command line:	Options.
3263	* dynamic symbol table:	Options.
3264	* ELF program headers:	PHDRS.
3265	* emulation:	Options.
3266	* emulation, default:	Environment.
3267	* END (MRI):	MRI.
3268	* endianness:	Options.
3269	* entry point, defaults:	Entry Point.
3270	* entry point, from command line:	Options.
3271	* ENTRY(SYMBOL):	Entry Point.
3272	* error on valid input:	Bug Criteria.
3273	* expression evaluation order:	Evaluation.
3274	* expression syntax:	Expressions.
3275	* expression, absolute:	Arithmetic Functions.
3276	* expressions in a section:	Section Data Expressions.
3277	* fatal signal:	Bug Criteria.
3278	* FILENAME:	Section Placement.
3279	* filename symbols:	Section Data Expressions.
3280	* FILENAME(SECTION):	Section Placement.
3281	* files and sections, section defn:	Section Placement.
3282	* files, including in output sections:	Section Placement.
3283	* fill pattern, entire section:	Section Options.
3284	* FILL(EXPRESSION):	Section Data Expressions.
3285	* first input file:	Option Commands.
3286	* first instruction:	Entry Point.
3287	* FLOAT:	Option Commands.
3288	* FORCE_COMMON_ALLOCATION:	Option Commands.
3289	* FORMAT (MRI):	MRI.
3290	* format, output file:	Option Commands.
3291	* functions in expression language:	Arithmetic Functions.
3292	* fundamental script commands:	Scripts.

3293	* GNU linker:	Overview.
3294	* GNUTARGET <1>:	Environment.
3295	* GNUTARGET:	Option Commands.
3296	* GROUP (FILES):	Option Commands.
3297	* grouping input files:	Option Commands.
3298	* groups of archives:	Options.
3299	* H8/300 support:	H8/300.
3300	* header size:	Arithmetic Functions.
3301	* help:	Options.
3302	* hexadecimal integers:	Integers.
3303	* holes:	Location Counter.
3304	* holes, filling:	Section Data Expressions.
3305	* i960 support:	i960.
3306	* INCLUDE FILENAME:	Option Commands.
3307	* including a linker script:	Option Commands.
3308	* including an entire archive:	Options.
3309	* incremental link:	Options.
3310	* INPUT (FILES):	Option Commands.
3311	* input file format:	Option Commands.
3312	* input filename symbols:	Section Data Expressions.
3313	* input files, displaying:	Options.
3314	* input files, section defn:	Section Placement.
3315	* input format:	Options.
3316	* input sections to output section:	Section Placement.
3317	* integer notation:	Integers.
3318	* integer suffixes:	Integers.
3319	* internal object-file format:	Canonical format.
3320	* invalid input:	Bug Criteria.
3321	* K and M integer suffixes:	Integers.
3322	* l =:	MEMORY.
3323	* L, deleting symbols beginning:	Options.
3324	* layout of output file:	Scripts.
3325	* lazy evaluation:	Evaluation.
3326	* ld bugs, reporting:	Bug Reporting.
3327	* LDEMULATION:	Environment.
3328	* len =:	MEMORY.
3329	* LENGTH =:	MEMORY.
3330	* link map:	Options.
3331	* link-time runtime library search path:	Options.
3332	* linker crash:	Bug Criteria.
3333	* LIST (MRI):	MRI.
3334	* little-endian objects:	Options.
3335	* LOAD (MRI):	MRI.
3336	* load address, specifying:	Section Options.
3337	* LOADADDR(SECTION):	Arithmetic Functions.
3338	* loading, preventing:	Section Options.
3339	* local symbols, deleting:	Options.
3340	* location counter:	Location Counter.
3341	* LONG(EXPRESSION):	Section Data Expressions.
3342	* M and K integer suffixes:	Integers.
3343	* machine architecture, output:	Option Commands.
3344	* machine dependencies:	Machine Dependent.
3345	* MAX:	Arithmetic Functions.
3346	* MEMORY:	MEMORY.
3347	* memory region attributes:	MEMORY.
3348	* memory regions and sections:	Section Options.
3349	* memory usage:	Options.
3350	* MIN:	Arithmetic Functions.
3351	* MIPS embedded PIC code:	Options.
3352	* MRI compatibility:	MRI.
3353	* NAME (MRI):	MRI.
3354	* names:	Symbols.
3355	* naming memory regions:	MEMORY.
3356	* naming output sections:	Section Definition.
3357	* naming the output file <1>:	Options.
3358	* naming the output file:	Option Commands.

3359	* negative integers:	Integers.
3360	* NEXT(EXP):	Arithmetic Functions.
3361	* NMAGIC:	Options.
3362	* NOCROSSREFS (SECTIONS):	Option Commands.
3363	* NOFLOAT:	Option Commands.
3364	* NOLOAD:	Section Options.
3365	* Non constant expression:	Assignment.
3366	* o =:	MEMORY.
3367	* objdump -i:	BFD.
3368	* object file management:	BFD.
3369	* object files:	Options.
3370	* object formats available:	BFD.
3371	* object size:	Options.
3372	* octal integers:	Integers.
3373	* OMAGIC:	Options.
3374	* opening object files:	BFD outline.
3375	* Operators for arithmetic:	Operators.
3376	* options:	Options.
3377	* ORDER (MRI):	MRI.
3378	* org =:	MEMORY.
3379	* ORIGIN =:	MEMORY.
3380	* OUTPUT (FILENAME):	Option Commands.
3381	* output file after errors:	Options.
3382	* output file layout:	Scripts.
3383	* OUTPUT_ARCH (BFDNAME):	Option Commands.
3384	* OUTPUT_FORMAT (BFDNAME):	Option Commands.
3385	* OVERLAY:	Overlays.
3386	* overlays:	Overlays.
3387	* partial link:	Options.
3388	* path for libraries:	Option Commands.
3389	* PHDRS:	PHDRS.
3390	* precedence in expressions:	Operators.
3391	* prevent unnecessary loading:	Section Options.
3392	* program headers:	PHDRS.
3393	* program headers and sections:	Section Options.
3394	* provide:	Assignment.
3395	* PUBLIC (MRI):	MRI.
3396	* QUAD(EXPRESSION):	Section Data Expressions.
3397	* quoted symbol names:	Symbols.
3398	* read-only text:	Options.
3399	* read/write from cmd line:	Options.
3400	* regions of memory:	MEMORY.
3401	* relaxing addressing modes:	Options.
3402	* relaxing on H8/300:	H8/300.
3403	* relaxing on i960:	i960.
3404	* relocatable and absolute symbols:	Assignment.
3405	* relocatable output:	Options.
3406	* reporting bugs in ld:	Reporting Bugs.
3407	* requirements for BFD:	BFD.
3408	* retaining specified symbols:	Options.
3409	* rounding up location counter:	Arithmetic Functions.
3410	* runtime library name:	Options.
3411	* runtime library search path:	Options.
3412	* scaled integers:	Integers.
3413	* script files:	Options.
3414	* search directory, from cmd line:	Options.
3415	* search path, libraries:	Option Commands.
3416	* SEARCH_DIR (PATH):	Option Commands.
3417	* SECT (MRI):	MRI.
3418	* section address <1>:	Arithmetic Functions.
3419	* section address:	Section Options.
3420	* section alignment:	Section Options.
3421	* section alignment, warnings on:	Options.
3422	* section definition:	Section Definition.
3423	* section defn, full syntax:	Section Options.
3424	* section fill pattern:	Section Options.

3425	* section load address:	Arithmetic Functions.
3426	* section size:	Arithmetic Functions.
3427	* section start:	Section Options.
3428	* section, assigning to memory region:	Section Options.
3429	* section, assigning to program header:	Section Options.
3430	* SECTIONS:	SECTIONS.
3431	* segment origins, cmd line:	Options.
3432	* semicolon:	Assignment.
3433	* shared libraries:	Options.
3434	* SHORT(EXPRESSION):	Section Data Expressions.
3435	* SIZEOF(SECTION):	Arithmetic Functions.
3436	* sizeof_headers:	Arithmetic Functions.
3437	* SIZEOF_HEADERS:	Arithmetic Functions.
3438	* specify load address:	Section Options.
3439	* SQUAD(EXPRESSION):	Section Data Expressions.
3440	* standard Unix system:	Options.
3441	* start address, section:	Section Options.
3442	* start of execution:	Entry Point.
3443	* STARTUP (FILENAME):	Option Commands.
3444	* strip all symbols:	Options.
3445	* strip debugger symbols:	Options.
3446	* stripping all but some symbols:	Options.
3447	* suffixes for integers:	Integers.
3448	* SYMBOL = EXPRESSION ;:	Section Data Expressions.
3449	* symbol defaults:	Arithmetic Functions.
3450	* symbol definition, scripts:	Assignment.
3451	* SYMBOL F= EXPRESSION ;:	Section Data Expressions.
3452	* symbol names:	Symbols.
3453	* symbol tracing:	Options.
3454	* symbol versions:	Version Script.
3455	* symbol-only input:	Options.
3456	* symbols, from command line:	Options.
3457	* symbols, relocatable and absolute:	Assignment.
3458	* symbols, retaining selectively:	Options.
3459	* synthesizing linker:	Options.
3460	* synthesizing on H8/300:	H8/300.
3461	* TARGET (FORMAT):	Option Commands.
3462	* traditional format:	Options.
3463	* unallocated address, next:	Arithmetic Functions.
3464	* undefined symbol:	Options.
3465	* undefined symbols, warnings on:	Options.
3466	* uninitialized data:	Section Placement.
3467	* unspecified memory:	Section Data Expressions.
3468	* usage:	Options.
3469	* variables, defining:	Assignment.
3470	* verbose:	Options.
3471	* version:	Options.
3472	* version script:	Version Script.
3473	* version script, symbol versions:	Options.
3474	* VERSION {script text}:	Version Script.
3475	* versions of symbols:	Version Script.
3476	* warnings, on combining symbols:	Options.
3477	* warnings, on section alignment:	Options.
3478	* warnings, on undefined symbols:	Options.
3479	* what is this?:	Overview.

3480

3481

3482 **US**

3483 Tag Table:

3484 Node: Top**DEF**949

3485 Node: Overview**DEF**1455

3486 Node: Invocation**DEF**2568

3487 Node: Options**DEF**2976

3488 Node: Environment**DEF**40932

3489 Node: Commands**DEF**42320

3490 Node: Scripts**DEF**43667

3491 Node: ExpressionsDEF144765
3492 Node: IntegersDEF145740
3493 Node: SymbolsDEF146580
3494 Node: Location CounterDEF147326
3495 Node: OperatorsDEF148474
3496 Node: EvaluationDEF149385
3497 Node: AssignmentDEF150080
3498 Node: Arithmetic FunctionsDEF153611
3499 Node: SemicolonsDEF157529
3500 Node: MEMORYDEF157978
3501 Node: SECTIONSDEF161161
3502 Node: Section DefinitionDEF162780
3503 Node: Section PlacementDEF164776
3504 Node: Section Data ExpressionsDEF170933
3505 Node: Section OptionsDEF175041
3506 Node: OverlaysDEF179352
3507 Node: PHDRSDEF183138
3508 Node: Entry PointDEF187960
3509 Node: Version ScriptDEF189501
3510 Node: Option CommandsDEF195617
3511 Node: Machine DependentDEF1102804
3512 Node: H8/300DEF1103224
3513 Node: i960DEF1104020
3514 Node: BFDDEF1105688
3515 Node: BFD outlineDEF1107142
3516 Node: BFD information lossDEF1108427
3517 Node: Canonical formatDEF1110935
3518 Node: Reporting BugsDEF1115281
3519 Node: Bug CriteriaDEF1115974
3520 Node: Bug ReportingDEF1116668
3521 Node: MRIDEF1123459
3522 Node: IndexDEF1128075
3523 US
3524 End Tag Table
3525