Using ld

The GNU linker

`ld` version 2
Version 2.15.94

Steve Chamberlain
Ian Lance Taylor

Red Hat Inc
nickc@credhat.com, doc@redhat.com
Using LD, the GNU linker
Edited by Jeffrey Osier (jeffrey@cygnus.com)

Copyright © 1991, 92, 93, 94, 95, 96, 97, 98, 99, 2000, 2001, 2002, 2003, 2004 Free Software Foundation, Inc.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with no Invariant Sections, with no Front-Cover Texts, and with no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

1 Overview

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

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

This version of `ld` uses the general purpose BFD libraries to operate on object files. This allows `ld` to read, combine, and write object files in many different formats—for example, COFF or `a.out`. Different formats may be linked together to produce any available kind of object file. See Chapter 5 [BFD], page 71, for more information.

Aside from its flexibility, the GNU linker is more helpful than other linkers in providing diagnostic information. Many linkers abandon execution immediately upon encountering an error; whenever possible, `ld` continues executing, allowing you to identify other errors (or, in some cases, to get an output file in spite of the error).

2 Invocation

The GNU linker `ld` is meant to cover a broad range of situations, and to be as compatible as possible with other linkers. As a result, you have many choices to control its behavior.

2.1 Command Line Options

The linker supports a plethora of command-line options, but in actual practice few of them are used in any particular context. For instance, a frequent use of `ld` is to link standard Unix object files on a standard, supported Unix system. On such a system, to link a file `hello.o`:

```
ld -o output /lib/crt0.o hello.o -lc
```

This tells `ld` to produce a file called *output* as the result of linking the file `/lib/crt0.o` with `hello.o` and the library `libc.a`, which will come from the standard search directories. (See the discussion of the '`-l`' option below.)

Some of the command-line options to `ld` may be specified at any point in the command line. However, options which refer to files, such as '`-l`' or '`-T`', cause the file to be read at the point at which the option appears in the command line, relative to the object files and other file options. Repeating non-file options with a different argument will either have no further effect, or override prior occurrences (those further to the left on the command line) of that option. Options which may be meaningfully specified more than once are noted in the descriptions below.

Non-option arguments are object files or archives which are to be linked together. They may follow, precede, or be mixed in with command-line options, except that an object file argument may not be placed between an option and its argument.

Usually the linker is invoked with at least one object file, but you can specify other forms of binary input files using '`-l`', '`-R`', and the script command language. If *no* binary input files at all are specified, the linker does not produce any output, and issues the message '`No input files`'.

If the linker cannot recognize the format of an object file, it will assume that it is a linker script. A script specified in this way augments the main linker script used for the link (either the default linker script or the one specified by using '`-T`'). This feature permits the linker to link against a file which appears to be an object or an archive, but actually merely defines some symbol values, or uses `INPUT` or `GROUP` to load other objects. Note that specifying a script in this way merely augments the main linker script; use the '`-T`' option to replace the default linker script entirely. See Chapter 3 [Scripts], page 31.

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

For options whose names are multiple letters, either one dash or two can precede the option name; for example, '`-trace-symbol`' and '`--trace-symbol`' are equivalent. Note—there is one exception to this rule. Multiple letter options that start with a lower case 'o' can only be preceeded by two dashes. This is to reduce confusion with the '`-o`' option. So for example '`-omagic`' sets the output file name to '`magic`' whereas '`--omagic`' sets the NMAGIC flag on the output.

Arguments to multiple-letter options must either be separated from the option name by an equals sign, or be given as separate arguments immediately following the option that requires them. For example, '`--trace-symbol foo`' and '`--trace-symbol=foo`' are equivalent. Unique abbreviations of the names of multiple-letter options are accepted.

Note—if the linker is being invoked indirectly, via a compiler driver (e.g. '`gcc`') then all the linker command line options should be prefixed by '`-Wl,`' (or whatever is appropriate for the particular compiler driver) like this:

```
gcc -Wl,--startgroup foo.o bar.o -Wl,--endgroup
```

This is important, because otherwise the compiler driver program may silently drop the linker options, resulting in a bad link.

Here is a table of the generic command line switches accepted by the GNU linker:

`-a`*keyword*

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

`-A`*architecture*
`--architecture=`*architecture*

In the current release of `ld`, this option is useful only for the Intel 960 family of architectures. In that `ld` configuration, the *architecture* argument identifies the particular architecture in the 960 family, enabling some safeguards and modifying the archive-library search path. See Section 4.2 [`ld` and the Intel 960 family], page 61, for details.

Future releases of `ld` may support similar functionality for other architecture families.

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

`ld` may be configured to support more than one kind of object file. If your `ld` is configured this way, you can use the '`-b`' option to specify the binary format for input object files that follow this option on the command line. Even when `ld` is configured to support alternative object formats, you don't usually need to specify this, as `ld` should be configured to expect as a default input format the most usual format on each machine. *input-format* is a text string, the name of a particular format supported by the BFD libraries. (You can list the available binary formats with '`objdump -i`'.) See Chapter 5 [BFD], page 71.

You may want to use this option if you are linking files with an unusual binary format. You can also use '`-b`' to switch formats explicitly (when linking object files of different formats), by including '`-b` *input-format*' before each group of object files in a particular format.

The default format is taken from the environment variable `GNUTARGET`. See Section 2.2 [Environment], page 29. You can also define the input format from a script, using the command `TARGET`; see Section 3.4.3 [Format Commands], page 34.

`-c` *MRI-commandfile*
`--mri-script=`*MRI-commandfile*
: For compatibility with linkers produced by MRI, `ld` accepts script files written in an alternate, restricted command language, described in Appendix A [MRI Compatible Script Files], page 79. Introduce MRI script files with the option '`-c`'; use the '`-T`' option to run linker scripts written in the general-purpose `ld` scripting language. If *MRI-cmdfile* does not exist, `ld` looks for it in the directories specified by any '`-L`' options.

`-d`
`-dc`
`-dp`
: These three options are equivalent; multiple forms are supported for compatibility with other linkers. They assign space to common symbols even if a relocatable output file is specified (with '`-r`'). The script command `FORCE_COMMON_ALLOCATION` has the same effect. See Section 3.4.4 [Miscellaneous Commands], page 35.

`-e` *entry*
`--entry=`*entry*
: Use *entry* as the explicit symbol for beginning execution of your program, rather than the default entry point. If there is no symbol named *entry*, the linker will try to parse *entry* as a number, and use that as the entry address (the number will be interpreted in base 10; you may use a leading '`0x`' for base 16, or a leading '`0`' for base 8). See Section 3.4.1 [Entry Point], page 33, for a discussion of defaults and other ways of specifying the entry point.

`--exclude-libs` *lib*`,`*lib*`,...`
: Specifies a list of archive libraries from which symbols should not be automatically exported. The library names may be delimited by commas or colons. Specifying `--exclude-libs ALL` excludes symbols in all archive libraries from automatic export. This option is available only for the i386 PE targeted port of the linker and for ELF targeted ports. For i386 PE, symbols explicitly listed in a .def file are still exported, regardless of this option. For ELF targeted ports, symbols affected by this option will be treated as hidden.

`-E`
`--export-dynamic`
: When creating a dynamically linked executable, add all symbols to the dynamic symbol table. The dynamic symbol table is the set of symbols which are visible from dynamic objects at run time.

  If you do not use this option, the dynamic symbol table will normally contain only those symbols which are referenced by some dynamic object mentioned in the link.

  If you use `dlopen` to load a dynamic object which needs to refer back to the symbols defined by the program, rather than some other dynamic object, then you will probably need to use this option when linking the program itself.

  You can also use the version script to control what symbols should be added to the dynamic symbol table if the output format supports it. See the description of '`--version-script`' in Section 3.9 [VERSION], page 52.

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

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

`-f`
`--auxiliary` *name*
: When creating an ELF shared object, set the internal DT_AUXILIARY field to the specified name. This tells the dynamic linker that the symbol table of the shared object should be used as an auxiliary filter on the symbol table of the shared object *name*.

  If you later link a program against this filter object, then, when you run the program, the dynamic linker will see the DT_AUXILIARY field. If the dynamic linker resolves any symbols from the filter object, it will first check whether there is a definition in the shared object *name*. If there is one, it will be used instead of the definition in the filter object. The shared object *name* need not exist. Thus the shared object *name* may be used to provide an alternative implementation of certain functions, perhaps for debugging or for machine specific performance.

  This option may be specified more than once. The DT_AUXILIARY entries will be created in the order in which they appear on the command line.

`-F` *name*
`--filter` *name*
: When creating an ELF shared object, set the internal DT_FILTER field to the specified name. This tells the dynamic linker that the symbol table of the shared object which is being created should be used as a filter on the symbol table of the shared object *name*.

  If you later link a program against this filter object, then, when you run the program, the dynamic linker will see the DT_FILTER field. The dynamic linker will resolve symbols according to the symbol table of the filter object as usual, but it will actually link to the definitions found in the shared object *name*. Thus the filter object can be used to select a subset of the symbols provided by the object *name*.

  Some older linkers used the '`-F`' option throughout a compilation toolchain for specifying object-file format for both input and output object files. The GNU linker uses other mechanisms for this purpose: the '`-b`', '`--format`', '`--oformat`' options, the `TARGET` command in linker scripts, and the `GNUTARGET` environment variable. The GNU linker will ignore the '`-F`' option when not creating an ELF shared object.

`-fini` *name*
: When creating an ELF executable or shared object, call NAME when the executable or shared object is unloaded, by setting DT_FINI to the address of the function. By default, the linker uses `_fini` as the function to call.

`-g` Ignored. Provided for compatibility with other tools.

`-G`*value*
`--gpsize=`*value*
: Set the maximum size of objects to be optimized using the GP register to *size*. This is only meaningful for object file formats such as MIPS ECOFF which

supports putting large and small objects into different sections. This is ignored for other object file formats.

`-h`*name*
`-soname=`*name*
: When creating an ELF shared object, set the internal DT_SONAME field to the specified name. When an executable is linked with a shared object which has a DT_SONAME field, then when the executable is run the dynamic linker will attempt to load the shared object specified by the DT_SONAME field rather than the using the file name given to the linker.

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

`-init` *name*
: When creating an ELF executable or shared object, call NAME when the executable or shared object is loaded, by setting DT_INIT to the address of the function. By default, the linker uses `_init` as the function to call.

`-l`*archive*
`--library=`*archive*
: Add archive file *archive* to the list of files to link. This option may be used any number of times. `ld` will search its path-list for occurrences of `lib`*archive*`.a` for every *archive* specified.

  On systems which support shared libraries, `ld` may also search for libraries with extensions other than `.a`. Specifically, on ELF and SunOS systems, `ld` will search a directory for a library with an extension of `.so` before searching for one with an extension of `.a`. By convention, a `.so` extension indicates a shared library.

  The linker will search an archive only once, at the location where it is specified on the command line. If the archive defines a symbol which was undefined in some object which appeared before the archive on the command line, the linker will include the appropriate file(s) from the archive. However, an undefined symbol in an object appearing later on the command line will not cause the linker to search the archive again.

  See the '`-(`' option for a way to force the linker to search archives multiple times.

  You may list the same archive multiple times on the command line.

  This type of archive searching is standard for Unix linkers. However, if you are using `ld` on AIX, note that it is different from the behaviour of the AIX linker.

`-L`*searchdir*
`--library-path=`*searchdir*
: Add path *searchdir* to the list of paths that `ld` will search for archive libraries and `ld` control scripts. You may use this option any number of times. The directories are searched in the order in which they are specified on the command line. Directories specified on the command line are searched before the default directories. All '`-L`' options apply to all '`-l`' options, regardless of the order in which the options appear.

If *searchdir* begins with `=`, then the `=` will be replaced by the *sysroot prefix*, a path specified when the linker is configured.

The default set of paths searched (without being specified with '`-L`') depends on which emulation mode `ld` is using, and in some cases also on how it was configured. See Section 2.2 [Environment], page 29.

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

`-m`*emulation*

Emulate the *emulation* linker. You can list the available emulations with the '`--verbose`' or '`-V`' options.

If the '`-m`' option is not used, the emulation is taken from the `LDEMULATION` environment variable, if that is defined.

Otherwise, the default emulation depends upon how the linker was configured.

`-M`
`--print-map`

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

- Where object files and symbols are mapped into memory.
- How common symbols are allocated.
- All archive members included in the link, with a mention of the symbol which caused the archive member to be brought in.

`-n`
`--nmagic` Turn off page alignment of sections, and mark the output as `NMAGIC` if possible.

`-N`
`--omagic` Set the text and data sections to be readable and writable. Also, do not page-align the data segment, and disable linking against shared libraries. If the output format supports Unix style magic numbers, mark the output as `OMAGIC`. Note: Although a writable text section is allowed for PE-COFF targets, it does not conform to the format specification published by Microsoft.

`--no-omagic`

This option negates most of the effects of the '`-N`' option. It sets the text section to be read-only, and forces the data segment to be page-aligned. Note - this option does not enable linking against shared libraries. Use '`-Bdynamic`' for this.

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

Use *output* as the name for the program produced by `ld`; if this option is not specified, the name '`a.out`' is used by default. The script command `OUTPUT` can also specify the output file name.

`-O` *level* If *level* is a numeric values greater than zero `ld` optimizes the output. This might take significantly longer and therefore probably should only be enabled for the final binary.

`-q`
`--emit-relocs`
: Leave relocation sections and contents in fully linked exececutables. Post link analysis and optimization tools may need this information in order to perform correct modifications of executables. This results in larger executables.

  This option is currently only supported on ELF platforms.

`-r`
`--relocatable`
: Generate relocatable output—i.e., generate an output file that can in turn serve as input to `ld`. This is often called *partial linking*. As a side effect, in environments that support standard Unix magic numbers, this option also sets the output file's magic number to `OMAGIC`. If this option is not specified, an absolute file is produced. When linking C++ programs, this option *will not* resolve references to constructors; to do that, use '`-Ur`'.

  When an input file does not have the same format as the output file, partial linking is only supported if that input file does not contain any relocations. Different output formats can have further restrictions; for example some `a.out`-based formats do not support partial linking with input files in other formats at all.

  This option does the same thing as '`-i`'.

`-R` *filename*
`--just-symbols=`*filename*
: Read symbol names and their addresses from *filename*, but do not relocate it or include it in the output. This allows your output file to refer symbolically to absolute locations of memory defined in other programs. You may use this option more than once.

  For compatibility with other ELF linkers, if the '`-R`' option is followed by a directory name, rather than a file name, it is treated as the '`-rpath`' option.

`-s`
`--strip-all`
: Omit all symbol information from the output file.

`-S`
`--strip-debug`
: Omit debugger symbol information (but not all symbols) from the output file.

`-t`
`--trace`
: Print the names of the input files as `ld` processes them.

`-T` *scriptfile*
`--script=`*scriptfile*
: Use *scriptfile* as the linker script. This script replaces `ld`'s default linker script (rather than adding to it), so *commandfile* must specify everything necessary to describe the output file. See Chapter 3 [Scripts], page 31. If *scriptfile* does not exist in the current directory, `ld` looks for it in the directories specified by any preceding '`-L`' options. Multiple '`-T`' options accumulate.

`-u` *symbol*
`--undefined=`*symbol*
: Force *symbol* to be entered in the output file as an undefined symbol. Doing this may, for example, trigger linking of additional modules from standard libraries. '`-u`' may be repeated with different option arguments to enter additional undefined symbols. This option is equivalent to the `EXTERN` linker script command.

`-Ur`
: For anything other than C++ programs, this option is equivalent to '`-r`': it generates relocatable output—i.e., an output file that can in turn serve as input to `ld`. When linking C++ programs, '`-Ur`' *does* resolve references to constructors, unlike '`-r`'. It does not work to use '`-Ur`' on files that were themselves linked with '`-Ur`'; once the constructor table has been built, it cannot be added to. Use '`-Ur`' only for the last partial link, and '`-r`' for the others.

`--unique[=`*SECTION*`]`
: Creates a separate output section for every input section matching *SECTION*, or if the optional wildcard *SECTION* argument is missing, for every orphan input section. An orphan section is one not specifically mentioned in a linker script. You may use this option multiple times on the command line; It prevents the normal merging of input sections with the same name, overriding output section assignments in a linker script.

`-v`
`--version`
`-V`
: Display the version number for `ld`. The '`-V`' option also lists the supported emulations.

`-x`
`--discard-all`
: Delete all local symbols.

`-X`
`--discard-locals`
: Delete all temporary local symbols. For most targets, this is all local symbols whose names begin with '`L`'.

`-y` *symbol*
`--trace-symbol=`*symbol*
: Print the name of each linked file in which *symbol* appears. This option may be given any number of times. On many systems it is necessary to prepend an underscore.

  This option is useful when you have an undefined symbol in your link but don't know where the reference is coming from.

`-Y` *path*
: Add *path* to the default library search path. This option exists for Solaris compatibility.

`-z` *keyword*
: The recognized keywords are:

'`combreloc`'
Combines multiple reloc sections and sorts them to make dynamic symbol lookup caching possible.

'`defs`' Disallows undefined symbols in object files. Undefined symbols in shared libraries are still allowed.

'`initfirst`'
This option is only meaningful when building a shared object. It marks the object so that its runtime initialization will occur before the runtime initialization of any other objects brought into the process at the same time. Similarly the runtime finalization of the object will occur after the runtime finalization of any other objects.

'`interpose`'
Marks the object that its symbol table interposes before all symbols but the primary executable.

'`loadfltr`'
Marks the object that its filters be processed immediately at runtime.

'`muldefs`' Allows multiple definitions.

'`nocombreloc`'
Disables multiple reloc sections combining.

'`nocopyreloc`'
Disables production of copy relocs.

'`nodefaultlib`'
Marks the object that the search for dependencies of this object will ignore any default library search paths.

'`nodelete`'
Marks the object shouldn't be unloaded at runtime.

'`nodlopen`'
Marks the object not available to `dlopen`.

'`nodump`' Marks the object can not be dumped by `dldump`.

'`now`' When generating an executable or shared library, mark it to tell the dynamic linker to resolve all symbols when the program is started, or when the shared library is linked to using dlopen, instead of deferring function call resolution to the point when the function is first called.

'`origin`' Marks the object may contain $ORIGIN.

Other keywords are ignored for Solaris compatibility.

`-(` *archives* `-)`
`--start-group` *archives* `--end-group`
The *archives* should be a list of archive files. They may be either explicit file names, or '`-l`' options.

The specified archives are searched repeatedly until no new undefined references are created. Normally, an archive is searched only once in the order that it is specified on the command line. If a symbol in that archive is needed to resolve an undefined symbol referred to by an object in an archive that appears later on the command line, the linker would not be able to resolve that reference. By grouping the archives, they all be searched repeatedly until all possible references are resolved.

Using this option has a significant performance cost. It is best to use it only when there are unavoidable circular references between two or more archives.

`--accept-unknown-input-arch`
`--no-accept-unknown-input-arch`
: Tells the linker to accept input files whose architecture cannot be recognised. The assumption is that the user knows what they are doing and deliberately wants to link in these unknown input files. This was the default behaviour of the linker, before release 2.14. The default behaviour from release 2.14 onwards is to reject such input files, and so the '`--accept-unknown-input-arch`' option has been added to restore the old behaviour.

`--as-needed`
`--no-as-needed`
: This option affects ELF DT_NEEDED tags for dynamic libraries mentioned on the command line after the '`--as-needed`' option. Normally, the linker will add a DT_NEEDED tag for each dynamic library mentioned on the command line, regardless of whether the library is actually needed. '`--as-needed`' causes DT_NEEDED tags to only be emitted for libraries that satisfy some reference from regular objects. '`--no-as-needed`' restores the default behaviour.

`--add-needed`
`--no-add-needed`
: This option affects the treatment of dynamic libraries from ELF DT_NEEDED tags in dynamic libraries mentioned on the command line after the '`--no-add-needed`' option. Normally, the linker will add a DT_NEEDED tag for each dynamic library from DT_NEEDED tags. '`--no-add-needed`' causes DT_NEEDED tags will never be emitted for those libraries from DT_NEEDED tags. '`--add-needed`' restores the default behaviour.

`-assert` *keyword*
: This option is ignored for SunOS compatibility.

`-Bdynamic`
`-dy`
`-call_shared`
: Link against dynamic libraries. This is only meaningful on platforms for which shared libraries are supported. This option is normally the default on such platforms. The different variants of this option are for compatibility with various systems. You may use this option multiple times on the command line: it affects library searching for '`-l`' options which follow it.

`-Bgroup` Set the `DF_1_GROUP` flag in the `DT_FLAGS_1` entry in the dynamic section. This causes the runtime linker to handle lookups in this object and its dependencies

to be performed only inside the group. '`--unresolved-symbols=report-all`' is implied. This option is only meaningful on ELF platforms which support shared libraries.

`-Bstatic`
`-dn`
`-non_shared`
`-static` Do not link against shared libraries. This is only meaningful on platforms for which shared libraries are supported. The different variants of this option are for compatibility with various systems. You may use this option multiple times on the command line: it affects library searching for '`-l`' options which follow it. This option also implies '`--unresolved-symbols=report-all`'.

`-Bsymbolic`
When creating a shared library, bind references to global symbols to the definition within the shared library, if any. Normally, it is possible for a program linked against a shared library to override the definition within the shared library. This option is only meaningful on ELF platforms which support shared libraries.

`--check-sections`
`--no-check-sections`
Asks the linker *not* to check section addresses after they have been assigned to see if there any overlaps. Normally the linker will perform this check, and if it finds any overlaps it will produce suitable error messages. The linker does know about, and does make allowances for sections in overlays. The default behaviour can be restored by using the command line switch '`--check-sections`'.

`--cref` Output a cross reference table. If a linker map file is being generated, the cross reference table is printed to the map file. Otherwise, it is printed on the standard output.

The format of the table is intentionally simple, so that it may be easily processed by a script if necessary. The symbols are printed out, sorted by name. For each symbol, a list of file names is given. If the symbol is defined, the first file listed is the location of the definition. The remaining files contain references to the symbol.

`--no-define-common`
This option inhibits the assignment of addresses to common symbols. The script command `INHIBIT_COMMON_ALLOCATION` has the same effect. See Section 3.4.4 [Miscellaneous Commands], page 35.

The '`--no-define-common`' option allows decoupling the decision to assign addresses to Common symbols from the choice of the output file type; otherwise a non-Relocatable output type forces assigning addresses to Common symbols. Using '`--no-define-common`' allows Common symbols that are referenced from a shared library to be assigned addresses only in the main program. This eliminates the unused duplicate space in the shared library, and also prevents any possible confusion over resolving to the wrong duplicate when there are many dynamic modules with specialized search paths for runtime symbol resolution.

`--defsym` *symbol=expression*
: Create a global symbol in the output file, containing the absolute address given by *expression*. You may use this option as many times as necessary to define multiple symbols in the command line. A limited form of arithmetic is supported for the *expression* in this context: you may give a hexadecimal constant or the name of an existing symbol, or use `+` and `-` to add or subtract hexadecimal constants or symbols. If you need more elaborate expressions, consider using the linker command language from a script (see Section 3.5 [Assignment: Symbol Definitions], page 36). *Note:* there should be no white space between *symbol*, the equals sign ("=="), and *expression*.

`--demangle[=`*style*`]`
`--no-demangle`
: These options control whether to demangle symbol names in error messages and other output. When the linker is told to demangle, it tries to present symbol names in a readable fashion: it strips leading underscores if they are used by the object file format, and converts C++ mangled symbol names into user readable names. Different compilers have different mangling styles. The optional demangling style argument can be used to choose an appropriate demangling style for your compiler. The linker will demangle by default unless the environment variable '`COLLECT_NO_DEMANGLE`' is set. These options may be used to override the default.

`--dynamic-linker` *file*
: Set the name of the dynamic linker. This is only meaningful when generating dynamically linked ELF executables. The default dynamic linker is normally correct; don't use this unless you know what you are doing.

`--fatal-warnings`
: Treat all warnings as errors.

`--force-exe-suffix`
: Make sure that an output file has a .exe suffix.

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

`--no-gc-sections`
`--gc-sections`
: Enable garbage collection of unused input sections. It is ignored on targets that do not support this option. This option is not compatible with '`-r`'. The default behaviour (of not performing this garbage collection) can be restored by specifying '`--no-gc-sections`' on the command line.

`--help`
: Print a summary of the command-line options on the standard output and exit.

`--target-help`
: Print a summary of all target specific options on the standard output and exit.

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

`--no-keep-memory`
: `ld` normally optimizes for speed over memory usage by caching the symbol tables of input files in memory. This option tells `ld` to instead optimize for memory usage, by rereading the symbol tables as necessary. This may be required if `ld` runs out of memory space while linking a large executable.

`--no-undefined`
`-z defs`
: Report unresolved symbol references from regular object files. This is done even if the linker is creating a non-symbolic shared library. The switch '`--[no-]allow-shlib-undefined`' controls the behaviour for reporting unresolved references found in shared libraries being linked in.

`--allow-multiple-definition`
`-z muldefs`
: Normally when a symbol is defined multiple times, the linker will report a fatal error. These options allow multiple definitions and the first definition will be used.

`--allow-shlib-undefined`
`--no-allow-shlib-undefined`
: Allows (the default) or disallows undefined symbols in shared libraries. This switch is similar to '`--no-undefined`' except that it determines the behaviour when the undefined symbols are in a shared library rather than a regular object file. It does not affect how undefined symbols in regular object files are handled.

  The reason that '`--allow-shlib-undefined`' is the default is that the shared library being specified at link time may not be the same as the one that is available at load time, so the symbols might actually be resolvable at load time. Plus there are some systems, (eg BeOS) where undefined symbols in shared libraries is normal. (The kernel patches them at load time to select which function is most appropriate for the current architecture. This is used for example to dynamically select an appropriate memset function). Apparently it is also normal for HPPA shared libraries to have undefined symbols.

`--no-undefined-version`
: Normally when a symbol has an undefined version, the linker will ignore it. This option disallows symbols with undefined version and a fatal error will be issued instead.

`--default-symver`
: Create and use a default symbol version (the soname) for unversioned exported symbols.

`--default-imported-symver`
: Create and use a default symbol version (the soname) for unversioned imported symbols.

`--no-warn-mismatch`
: Normally `ld` will give an error if you try to link together input files that are mismatched for some reason, perhaps because they have been compiled for different processors or for different endiannesses. This option tells `ld` that it should silently permit such possible errors. This option should only be used with care, in cases when you have taken some special action that ensures that the linker errors are inappropriate.

`--no-whole-archive`
: Turn off the effect of the '`--whole-archive`' option for subsequent archive files.

`--noinhibit-exec`
: Retain the executable output file whenever it is still usable. Normally, the linker will not produce an output file if it encounters errors during the link process; it exits without writing an output file when it issues any error whatsoever.

`-nostdlib`
: Only search library directories explicitly specified on the command line. Library directories specified in linker scripts (including linker scripts specified on the command line) are ignored.

`--oformat` *output-format*
: `ld` may be configured to support more than one kind of object file. If your `ld` is configured this way, you can use the '`--oformat`' option to specify the binary format for the output object file. Even when `ld` is configured to support alternative object formats, you don't usually need to specify this, as `ld` should be configured to produce as a default output format the most usual format on each machine. *output-format* is a text string, the name of a particular format supported by the BFD libraries. (You can list the available binary formats with '`objdump -i`'.) The script command `OUTPUT_FORMAT` can also specify the output format, but this option overrides it. See Chapter 5 [BFD], page 71.

`-pie`
`--pic-executable`
: Create a position independent executable. This is currently only supported on ELF platforms. Position independent executables are similar to shared libraries in that they are relocated by the dynamic linker to the virtual address the OS chooses for them (which can vary between invocations). Like normal dynamically linked executables they can be executed and symbols defined in the executable cannot be overridden by shared libraries.

`-qmagic` This option is ignored for Linux compatibility.

`-Qy` This option is ignored for SVR4 compatibility.

`--relax` An option with machine dependent effects. This option is only supported on a few targets. See Section 4.1 [`ld` and the H8/300], page 61. See Section 4.2 [`ld` and the Intel 960 family], page 61. See Section 4.9 [`ld` and Xtensa Processors], page 69.

On some platforms, the '`--relax`' option performs global optimizations that become possible when the linker resolves addressing in the program, such as

relaxing address modes and synthesizing new instructions in the output object file.

On some platforms these link time global optimizations may make symbolic debugging of the resulting executable impossible. This is known to be the case for the Matsushita MN10200 and MN10300 family of processors.

On platforms where this is not supported, '`--relax`' is accepted, but ignored.

`--retain-symbols-file` *filename*

Retain *only* the symbols listed in the file *filename*, discarding all others. *filename* is simply a flat file, with one symbol name per line. This option is especially useful in environments (such as VxWorks) where a large global symbol table is accumulated gradually, to conserve run-time memory.

'`--retain-symbols-file`' does *not* discard undefined symbols, or symbols needed for relocations.

You may only specify '`--retain-symbols-file`' once in the command line. It overrides '`-s`' and '`-S`'.

`-rpath` *dir*

Add a directory to the runtime library search path. This is used when linking an ELF executable with shared objects. All '`-rpath`' arguments are concatenated and passed to the runtime linker, which uses them to locate shared objects at runtime. The '`-rpath`' option is also used when locating shared objects which are needed by shared objects explicitly included in the link; see the description of the '`-rpath-link`' option. If '`-rpath`' is not used when linking an ELF executable, the contents of the environment variable `LD_RUN_PATH` will be used if it is defined.

The '`-rpath`' option may also be used on SunOS. By default, on SunOS, the linker will form a runtime search patch out of all the '`-L`' options it is given. If a '`-rpath`' option is used, the runtime search path will be formed exclusively using the '`-rpath`' options, ignoring the '`-L`' options. This can be useful when using gcc, which adds many '`-L`' options which may be on NFS mounted filesystems.

For compatibility with other ELF linkers, if the '`-R`' option is followed by a directory name, rather than a file name, it is treated as the '`-rpath`' option.

`-rpath-link` *DIR*

When using ELF or SunOS, one shared library may require another. This happens when an `ld -shared` link includes a shared library as one of the input files.

When the linker encounters such a dependency when doing a non-shared, non-relocatable link, it will automatically try to locate the required shared library and include it in the link, if it is not included explicitly. In such a case, the '`-rpath-link`' option specifies the first set of directories to search. The '`-rpath-link`' option may specify a sequence of directory names either by specifying a list of names separated by colons, or by appearing multiple times.

This option should be used with caution as it overrides the search path that may have been hard compiled into a shared library. In such a case it is possible to use unintentionally a different search path than the runtime linker would do.

The linker uses the following search paths to locate required shared libraries.

1. Any directories specified by '`-rpath-link`' options.
2. Any directories specified by '`-rpath`' options. The difference between '`-rpath`' and '`-rpath-link`' is that directories specified by '`-rpath`' options are included in the executable and used at runtime, whereas the '`-rpath-link`' option is only effective at link time. It is for the native linker only.
3. On an ELF system, if the '`-rpath`' and `rpath-link` options were not used, search the contents of the environment variable `LD_RUN_PATH`. It is for the native linker only.
4. On SunOS, if the '`-rpath`' option was not used, search any directories specified using '`-L`' options.
5. For a native linker, the contents of the environment variable `LD_LIBRARY_PATH`.
6. For a native ELF linker, the directories in `DT_RUNPATH` or `DT_RPATH` of a shared library are searched for shared libraries needed by it. The `DT_RPATH` entries are ignored if `DT_RUNPATH` entries exist.
7. The default directories, normally '`/lib`' and '`/usr/lib`'.
8. For a native linker on an ELF system, if the file '`/etc/ld.so.conf`' exists, the list of directories found in that file.

If the required shared library is not found, the linker will issue a warning and continue with the link.

`-shared`
`-Bshareable`
: Create a shared library. This is currently only supported on ELF, XCOFF and SunOS platforms. On SunOS, the linker will automatically create a shared library if the '`-e`' option is not used and there are undefined symbols in the link.

`--sort-common`
: This option tells `ld` to sort the common symbols by size when it places them in the appropriate output sections. First come all the one byte symbols, then all the two byte, then all the four byte, and then everything else. This is to prevent gaps between symbols due to alignment constraints.

`--sort-section name`
: This option will apply `SORT_BY_NAME` to all wildcard section patterns in the linker script.

`--sort-section alignment`
: This option will apply `SORT_BY_ALIGNMENT` to all wildcard section patterns in the linker script.

`--split-by-file [`*size*`]`
: Similar to '`--split-by-reloc`' but creates a new output section for each input file when *size* is reached. *size* defaults to a size of 1 if not given.

`--split-by-reloc [`*count*`]`
: Tries to creates extra sections in the output file so that no single output section in the file contains more than *count* relocations. This is useful when generating huge relocatable files for downloading into certain real time kernels with the COFF object file format; since COFF cannot represent more than 65535 relocations in a single section. Note that this will fail to work with object file formats which do not support arbitrary sections. The linker will not split up individual input sections for redistribution, so if a single input section contains more than *count* relocations one output section will contain that many relocations. *count* defaults to a value of 32768.

`--stats`
: Compute and display statistics about the operation of the linker, such as execution time and memory usage.

`--traditional-format`
: For some targets, the output of `ld` is different in some ways from the output of some existing linker. This switch requests `ld` to use the traditional format instead.

  For example, on SunOS, `ld` combines duplicate entries in the symbol string table. This can reduce the size of an output file with full debugging information by over 30 percent. Unfortunately, the SunOS `dbx` program can not read the resulting program (`gdb` has no trouble). The '`--traditional-format`' switch tells `ld` to not combine duplicate entries.

`--section-start` *sectionname*`=`*org*
: Locate a section in the output file at the absolute address given by *org*. You may use this option as many times as necessary to locate multiple sections in the command line. *org* must be a single hexadecimal integer; for compatibility with other linkers, you may omit the leading '`0x`' usually associated with hexadecimal values. *Note:* there should be no white space between *sectionname*, the equals sign ("⩵"), and *org*.

`-Tbss` *org*
`-Tdata` *org*
`-Ttext` *org*
: Same as –section-start, with `.bss`, `.data` or `.text` as the *sectionname*.

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

  '`ignore-all`'
  : Do not report any unresolved symbols.

  '`report-all`'
  : Report all unresolved symbols. This is the default.

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

'`ignore-in-shared-libs`'
: Report unresolved symbols that come from regular object files, but ignore them if they come from shared libraries. This can be useful when creating a dynamic binary and it is known that all the shared libraries that it should be referencing are included on the linker's command line.

The behaviour for shared libraries on their own can also be controlled by the '`--[no-]allow-shlib-undefined`' option.

Normally the linker will generate an error message for each reported unresolved symbol but the option '`--warn-unresolved-symbols`' can change this to a warning.

`--dll-verbose`
`--verbose`
: Display the version number for `ld` and list the linker emulations supported. Display which input files can and cannot be opened. Display the linker script being used by the linker.

`--version-script=`*`version-scriptfile`*
: Specify the name of a version script to the linker. This is typically used when creating shared libraries to specify additional information about the version hierarchy for the library being created. This option is only meaningful on ELF platforms which support shared libraries. See Section 3.9 [VERSION], page 52.

`--warn-common`
: Warn when a common symbol is combined with another common symbol or with a symbol definition. Unix linkers allow this somewhat sloppy practise, but linkers on some other operating systems do not. This option allows you to find potential problems from combining global symbols. Unfortunately, some C libraries use this practise, so you may get some warnings about symbols in the libraries as well as in your programs.

  There are three kinds of global symbols, illustrated here by C examples:

  '`int i = 1;`'
  : A definition, which goes in the initialized data section of the output file.

  '`extern int i;`'
  : An undefined reference, which does not allocate space. There must be either a definition or a common symbol for the variable somewhere.

  '`int i;`'
  : A common symbol. If there are only (one or more) common symbols for a variable, it goes in the uninitialized data area of the output file. The linker merges multiple common symbols for the same variable into a single symbol. If they are of different sizes, it picks the largest size. The linker turns a common symbol into a declaration, if there is a definition of the same variable.

  The '`--warn-common`' option can produce five kinds of warnings. Each warning consists of a pair of lines: the first describes the symbol just encountered, and

the second describes the previous symbol encountered with the same name. One or both of the two symbols will be a common symbol.

1. Turning a common symbol into a reference, because there is already a definition for the symbol.

```
file(section): warning: common of `symbol'
   overridden by definition
file(section): warning: defined here
```

2. Turning a common symbol into a reference, because a later definition for the symbol is encountered. This is the same as the previous case, except that the symbols are encountered in a different order.

```
file(section): warning: definition of `symbol'
   overriding common
file(section): warning: common is here
```

3. Merging a common symbol with a previous same-sized common symbol.

```
file(section): warning: multiple common
   of `symbol'
file(section): warning: previous common is here
```

4. Merging a common symbol with a previous larger common symbol.

```
file(section): warning: common of `symbol'
   overridden by larger common
file(section): warning: larger common is here
```

5. Merging a common symbol with a previous smaller common symbol. This is the same as the previous case, except that the symbols are encountered in a different order.

```
file(section): warning: common of `symbol'
   overriding smaller common
file(section): warning: smaller common is here
```

`--warn-constructors`

Warn if any global constructors are used. This is only useful for a few object file formats. For formats like COFF or ELF, the linker can not detect the use of global constructors.

`--warn-multiple-gp`

Warn if multiple global pointer values are required in the output file. This is only meaningful for certain processors, such as the Alpha. Specifically, some processors put large-valued constants in a special section. A special register (the global pointer) points into the middle of this section, so that constants can be loaded efficiently via a base-register relative addressing mode. Since the offset in base-register relative mode is fixed and relatively small (e.g., 16 bits), this limits the maximum size of the constant pool. Thus, in large programs, it is often necessary to use multiple global pointer values in order to be able to address all possible constants. This option causes a warning to be issued whenever this case occurs.

`--warn-once`

Only warn once for each undefined symbol, rather than once per module which refers to it.

`--warn-section-align`

Warn if the address of an output section is changed because of alignment. Typically, the alignment will be set by an input section. The address will only

be changed if it not explicitly specified; that is, if the `SECTIONS` command does not specify a start address for the section (see Section 3.6 [SECTIONS], page 37).

`--warn-shared-textrel`
: Warn if the linker adds a DT_TEXTREL to a shared object.

`--warn-unresolved-symbols`
: If the linker is going to report an unresolved symbol (see the option '`--unresolved-symbols`') it will normally generate an error. This option makes it generate a warning instead.

`--error-unresolved-symbols`
: This restores the linker's default behaviour of generating errors when it is reporting unresolved symbols.

`--whole-archive`
: For each archive mentioned on the command line after the '`--whole-archive`' option, include every object file in the archive in the link, rather than searching the archive for the required object files. This is normally used to turn an archive file into a shared library, forcing every object to be included in the resulting shared library. This option may be used more than once.

  Two notes when using this option from gcc: First, gcc doesn't know about this option, so you have to use '`-Wl,-whole-archive`'. Second, don't forget to use '`-Wl,-no-whole-archive`' after your list of archives, because gcc will add its own list of archives to your link and you may not want this flag to affect those as well.

`--wrap` *symbol*
: Use a wrapper function for *symbol*. Any undefined reference to *symbol* will be resolved to `__wrap_`*symbol*. Any undefined reference to `__real_`*symbol* will be resolved to *symbol*.

  This can be used to provide a wrapper for a system function. The wrapper function should be called `__wrap_`*symbol*. If it wishes to call the system function, it should call `__real_`*symbol*.

  Here is a trivial example:

```
void *
__wrap_malloc (size_t c)
{
  printf ("malloc called with %zu\n", c);
  return __real_malloc (c);
}
```

  If you link other code with this file using '`--wrap malloc`', then all calls to `malloc` will call the function `__wrap_malloc` instead. The call to `__real_malloc` in `__wrap_malloc` will call the real `malloc` function.

  You may 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`.

`--enable-new-dtags`
`--disable-new-dtags`
: This linker can create the new dynamic tags in ELF. But the older ELF systems may not understand them. If you specify '`--enable-new-dtags`', the dynamic tags will be created as needed. If you specify '`--disable-new-dtags`', no new dynamic tags will be created. By default, the new dynamic tags are not created. Note that those options are only available for ELF systems.

  Set the default size of the linker's hash tables to a prime number close to *number*. Increasing this value can reduce the length of time it takes the linker to perform its tasks, at the expense of increasing the linker's memory requirements. Similarly reducing this value can reduce the memory requirements at the expense of speed.

`--reduce-memory-overheads`
: This option reduces memory requirements at ld runtime, at the expense of linking speed. This was introduced to to select the old O(n^2) algorithm for link map file generation, rather than the new O(n) algorithm which uses about 40% more memory for symbol storage.

  Another affect of the switch is to set the default hash table size to 1021, which again saves memory at the cost of lengthening the linker's run time. This is not done however if the '`--hash-size`' switch has been used.

  The '`--reduce-memory-overheads`' switch may be also be used to enable other tradeoffs in future versions of the linker.

2.1.1 Options Specific to i386 PE Targets

The i386 PE linker supports the '`-shared`' option, which causes the output to be a dynamically linked library (DLL) instead of a normal executable. You should name the output `*.dll` when you use this option. In addition, the linker fully supports the standard `*.def` files, which may be specified on the linker command line like an object file (in fact, it should precede archives it exports symbols from, to ensure that they get linked in, just like a normal object file).

In addition to the options common to all targets, the i386 PE linker support additional command line options that are specific to the i386 PE target. Options that take values may be separated from their values by either a space or an equals sign.

`--add-stdcall-alias`
: If given, symbols with a stdcall suffix (`@`*nn*) will be exported as-is and also with the suffix stripped. [This option is specific to the i386 PE targeted port of the linker]

`--base-file` *file*
: Use *file* as the name of a file in which to save the base addresses of all the relocations needed for generating DLLs with '`dlltool`'. [This is an i386 PE specific option]

`--dll`
: Create a DLL instead of a regular executable. You may also use '`-shared`' or specify a `LIBRARY` in a given `.def` file. [This option is specific to the i386 PE targeted port of the linker]

`--enable-stdcall-fixup`
`--disable-stdcall-fixup`

If the link finds a symbol that it cannot resolve, it will attempt to do "fuzzy linking" by looking for another defined symbol that differs only in the format of the symbol name (cdecl vs stdcall) and will resolve that symbol by linking to the match. For example, the undefined symbol `_foo` might be linked to the function `_foo@12`, or the undefined symbol `_bar@16` might be linked to the function `_bar`. When the linker does this, it prints a warning, since it normally should have failed to link, but sometimes import libraries generated from third-party dlls may need this feature to be usable. If you specify '`--enable-stdcall-fixup`', this feature is fully enabled and warnings are not printed. If you specify '`--disable-stdcall-fixup`', this feature is disabled and such mismatches are considered to be errors. [This option is specific to the i386 PE targeted port of the linker]

`--export-all-symbols`

If given, all global symbols in the objects used to build a DLL will be exported by the DLL. Note that this is the default if there otherwise wouldn't be any exported symbols. When symbols are explicitly exported via DEF files or implicitly exported via function attributes, the default is to not export anything else unless this option is given. Note that the symbols `DllMain@12`, `DllEntryPoint@0`, `DllMainCRTStartup@12`, and `impure_ptr` will not be automatically exported. Also, symbols imported from other DLLs will not be re-exported, nor will symbols specifying the DLL's internal layout such as those beginning with `_head_` or ending with `_iname`. In addition, no symbols from `libgcc`, `libstd++`, `libmingw32`, or `crtX.o` will be exported. Symbols whose names begin with `__rtti_` or `__builtin_` will not be exported, to help with C++ DLLs. Finally, there is an extensive list of cygwin-private symbols that are not exported (obviously, this applies on when building DLLs for cygwin targets). These cygwin-excludes are: `_cygwin_dll_entry@12`, `_cygwin_crt0_common@8`, `_cygwin_noncygwin_dll_entry@12`, `_fmode`, `_impure_ptr`, `cygwin_attach_dll`, `cygwin_premain0`, `cygwin_premain1`, `cygwin_premain2`, `cygwin_premain3`, and `environ`. [This option is specific to the i386 PE targeted port of the linker]

`--exclude-symbols` *`symbol,symbol,...`*

Specifies a list of symbols which should not be automatically exported. The symbol names may be delimited by commas or colons. [This option is specific to the i386 PE targeted port of the linker]

`--file-alignment`

Specify the file alignment. Sections in the file will always begin at file offsets which are multiples of this number. This defaults to 512. [This option is specific to the i386 PE targeted port of the linker]

`--heap` *reserve*
`--heap` *reserve*`,`*commit*
: Specify the amount of memory to reserve (and optionally commit) to be used as heap for this program. The default is 1Mb reserved, 4K committed. [This option is specific to the i386 PE targeted port of the linker]

`--image-base` *value*
: Use *value* as the base address of your program or dll. This is the lowest memory location that will be used when your program or dll is loaded. To reduce the need to relocate and improve performance of your dlls, each should have a unique base address and not overlap any other dlls. The default is 0x400000 for executables, and 0x10000000 for dlls. [This option is specific to the i386 PE targeted port of the linker]

`--kill-at`
: If given, the stdcall suffixes (`@`*nn*) will be stripped from symbols before they are exported. [This option is specific to the i386 PE targeted port of the linker]

`--large-address-aware`
: If given, the appropriate bit in the "Charateristics" field of the COFF header is set to indicate that this executable supports virtual addresses greater than 2 gigabytes. This should be used in conjuction with the /3GB or /USERVA=*value* megabytes switch in the "[operating systems]" section of the BOOT.INI. Otherwise, this bit has no effect. [This option is specific to PE targeted ports of the linker]

`--major-image-version` *value*
: Sets the major number of the "image version". Defaults to 1. [This option is specific to the i386 PE targeted port of the linker]

`--major-os-version` *value*
: Sets the major number of the "os version". Defaults to 4. [This option is specific to the i386 PE targeted port of the linker]

`--major-subsystem-version` *value*
: Sets the major number of the "subsystem version". Defaults to 4. [This option is specific to the i386 PE targeted port of the linker]

`--minor-image-version` *value*
: Sets the minor number of the "image version". Defaults to 0. [This option is specific to the i386 PE targeted port of the linker]

`--minor-os-version` *value*
: Sets the minor number of the "os version". Defaults to 0. [This option is specific to the i386 PE targeted port of the linker]

`--minor-subsystem-version` *value*
: Sets the minor number of the "subsystem version". Defaults to 0. [This option is specific to the i386 PE targeted port of the linker]

`--output-def` *file*
: The linker will create the file *file* which will contain a DEF file corresponding to the DLL the linker is generating. This DEF file (which should be called

`*.def`) may be used to create an import library with `dlltool` or may be used as a reference to automatically or implicitly exported symbols. [This option is specific to the i386 PE targeted port of the linker]

`--out-implib` *file*
: The linker will create the file *file* which will contain an import lib corresponding to the DLL the linker is generating. This import lib (which should be called `*.dll.a` or `*.a` may be used to link clients against the generated DLL; this behaviour makes it possible to skip a separate `dlltool` import library creation step. [This option is specific to the i386 PE targeted port of the linker]

`--enable-auto-image-base`
: Automatically choose the image base for DLLs, unless one is specified using the `--image-base` argument. By using a hash generated from the dllname to create unique image bases for each DLL, in-memory collisions and relocations which can delay program execution are avoided. [This option is specific to the i386 PE targeted port of the linker]

`--disable-auto-image-base`
: Do not automatically generate a unique image base. If there is no user-specified image base (`--image-base`) then use the platform default. [This option is specific to the i386 PE targeted port of the linker]

`--dll-search-prefix` *string*
: When linking dynamically to a dll without an import library, search for `<string><basename>.dll` in preference to `lib<basename>.dll`. This behaviour allows easy distinction between DLLs built for the various "subplatforms": native, cygwin, uwin, pw, etc. For instance, cygwin DLLs typically use `--dll-search-prefix=cyg`. [This option is specific to the i386 PE targeted port of the linker]

`--enable-auto-import`
: Do sophisticated linking of `_symbol` to `__imp__symbol` for DATA imports from DLLs, and create the necessary thunking symbols when building the import libraries with those DATA exports. Note: Use of the 'auto-import' extension will cause the text section of the image file to be made writable. This does not conform to the PE-COFF format specification published by Microsoft.

  Using 'auto-import' generally will 'just work' – but sometimes you may see this message:

  "variable '<var>' can't be auto-imported. Please read the documentation for ld's `--enable-auto-import` for details."

  This message occurs when some (sub)expression accesses an address ultimately given by the sum of two constants (Win32 import tables only allow one). Instances where this may occur include accesses to member fields of struct variables imported from a DLL, as well as using a constant index into an array variable imported from a DLL. Any multiword variable (arrays, structs, long long, etc) may trigger this error condition. However, regardless of the exact data type of the offending exported variable, ld will always detect it, issue the warning, and exit.

There are several ways to address this difficulty, regardless of the data type of the exported variable:

One way is to use –enable-runtime-pseudo-reloc switch. This leaves the task of adjusting references in your client code for runtime environment, so this method works only when runtime environment supports this feature.

A second solution is to force one of the 'constants' to be a variable – that is, unknown and un-optimizable at compile time. For arrays, there are two possibilities: a) make the indexee (the array's address) a variable, or b) make the 'constant' index a variable. Thus:

```
extern type extern_array[];
extern_array[1] -->
   { volatile type *t=extern_array; t[1] }
```

or

```
extern type extern_array[];
extern_array[1] -->
   { volatile int t=1; extern_array[t] }
```

For structs (and most other multiword data types) the only option is to make the struct itself (or the long long, or the ...) variable:

```
extern struct s extern_struct;
extern_struct.field -->
   { volatile struct s *t=&extern_struct; t->field }
```

or

```
extern long long extern_ll;
extern_ll -->
  { volatile long long * local_ll=&extern_ll; *local_ll }
```

A third method of dealing with this difficulty is to abandon 'auto-import' for the offending symbol and mark it with `__declspec(dllimport)`. However, in practise that requires using compile-time #defines to indicate whether you are building a DLL, building client code that will link to the DLL, or merely building/linking to a static library. In making the choice between the various methods of resolving the 'direct address with constant offset' problem, you should consider typical real-world usage:

Original:

```
--foo.h
extern int arr[];
--foo.c
#include "foo.h"
void main(int argc, char **argv){
  printf("%d\n",arr[1]);
}
```

Solution 1:

```
--foo.h
extern int arr[];
--foo.c
```

```
#include "foo.h"
void main(int argc, char **argv){
  /* This workaround is for win32 and cygwin; do not "optimize" */
  volatile int *parr = arr;
  printf("%d\n",parr[1]);
}
```

Solution 2:

```
--foo.h
/* Note: auto-export is assumed (no __declspec(dllexport)) */
#if (defined(_WIN32) || defined(__CYGWIN__)) && \
  !(defined(FOO_BUILD_DLL) || defined(FOO_STATIC))
#define FOO_IMPORT __declspec(dllimport)
#else
#define FOO_IMPORT
#endif
extern FOO_IMPORT int arr[];
--foo.c
#include "foo.h"
void main(int argc, char **argv){
  printf("%d\n",arr[1]);
}
```

A fourth way to avoid this problem is to re-code your library to use a functional interface rather than a data interface for the offending variables (e.g. set_foo() and get_foo() accessor functions). [This option is specific to the i386 PE targeted port of the linker]

`--disable-auto-import`
: Do not attempt to do sophisticated linking of `_symbol` to `__imp__symbol` for DATA imports from DLLs. [This option is specific to the i386 PE targeted port of the linker]

`--enable-runtime-pseudo-reloc`
: If your code contains expressions described in –enable-auto-import section, that is, DATA imports from DLL with non-zero offset, this switch will create a vector of 'runtime pseudo relocations' which can be used by runtime environment to adjust references to such data in your client code. [This option is specific to the i386 PE targeted port of the linker]

`--disable-runtime-pseudo-reloc`
: Do not create pseudo relocations for non-zero offset DATA imports from DLLs. This is the default. [This option is specific to the i386 PE targeted port of the linker]

`--enable-extra-pe-debug`
: Show additional debug info related to auto-import symbol thunking. [This option is specific to the i386 PE targeted port of the linker]

`--section-alignment`
: Sets the section alignment. Sections in memory will always begin at addresses which are a multiple of this number. Defaults to 0x1000. [This option is specific to the i386 PE targeted port of the linker]

`--stack` *reserve*
`--stack` *reserve*`,`*commit*
: Specify the amount of memory to reserve (and optionally commit) to be used as stack for this program. The default is 2Mb reserved, 4K committed. [This option is specific to the i386 PE targeted port of the linker]

`--subsystem` *which*
`--subsystem` *which*`:`*major*
`--subsystem` *which*`:`*major*`.`*minor*
: Specifies the subsystem under which your program will execute. The legal values for *which* are `native`, `windows`, `console`, `posix`, and `xbox`. You may optionally set the subsystem version also. Numeric values are also accepted for *which*. [This option is specific to the i386 PE targeted port of the linker]

2.2 Environment Variables

You can change the behaviour of `ld` with the environment variables `GNUTARGET`, `LDEMULATION` and `COLLECT_NO_DEMANGLE`.

`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 (see Chapter 5 [BFD], page 71). 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.

Normally, the linker will default to demangling symbols. However, if `COLLECT_NO_DEMANGLE` is set in the environment, then it will default to not demangling symbols. This environment variable is used in a similar fashion by the `gcc` linker wrapper program. The default may be overridden by the '`--demangle`' and '`--no-demangle`' options.

3 Linker Scripts

Every link is controlled by a *linker script*. This script is written in the linker command language.

The main purpose of the linker script is to describe how the sections in the input files should be mapped into the output file, and to control the memory layout of the output file. Most linker scripts do nothing more than this. However, when necessary, the linker script can also direct the linker to perform many other operations, using the commands described below.

The linker always uses a linker script. If you do not supply one yourself, the linker will use a default script that is compiled into the linker executable. You can use the '`--verbose`' command line option to display the default linker script. Certain command line options, such as '`-r`' or '`-N`', will affect the default linker script.

You may supply your own linker script by using the '`-T`' command line option. When you do this, your linker script will replace the default linker script.

You may also use linker scripts implicitly by naming them as input files to the linker, as though they were files to be linked. See Section 3.11 [Implicit Linker Scripts], page 60.

3.1 Basic Linker Script Concepts

We need to define some basic concepts and vocabulary in order to describe the linker script language.

The linker combines input files into a single output file. The output file and each input file are in a special data format known as an *object file format*. Each file is called an *object file*. The output file is often called an *executable*, but for our purposes we will also call it an object file. Each object file has, among other things, a list of *sections*. We sometimes refer to a section in an input file as an *input section*; similarly, a section in the output file is an *output section*.

Each section in an object file has a name and a size. Most sections also have an associated block of data, known as the *section contents*. A section may be marked as *loadable*, which mean that the contents should be loaded into memory when the output file is run. A section with no contents may be *allocatable*, which means that an area in memory should be set aside, but nothing in particular should be loaded there (in some cases this memory must be zeroed out). A section which is neither loadable nor allocatable typically contains some sort of debugging information.

Every loadable or allocatable output section has two addresses. The first is the *VMA*, or virtual memory address. This is the address the section will have when the output file is run. The second is the *LMA*, or load memory address. This is the address at which the section will be loaded. In most cases the two addresses will be the same. An example of when they might be different is when a data section is loaded into ROM, and then copied into RAM when the program starts up (this technique is often used to initialize global variables in a ROM based system). In this case the ROM address would be the LMA, and the RAM address would be the VMA.

You can see the sections in an object file by using the `objdump` program with the '`-h`' option.

Every object file also has a list of *symbols*, known as the *symbol table*. A symbol may be defined or undefined. Each symbol has a name, and each defined symbol has an address,

among other information. If you compile a C or C++ program into an object file, you will get a defined symbol for every defined function and global or static variable. Every undefined function or global variable which is referenced in the input file will become an undefined symbol.

You can see the symbols in an object file by using the `nm` program, or by using the `objdump` program with the '`-t`' option.

3.2 Linker Script Format

Linker scripts are text files.

You write a linker script as a series of commands. Each command is either a keyword, possibly followed by arguments, or an assignment to a symbol. You may separate commands using semicolons. Whitespace is generally ignored.

Strings such as file or format names can normally be entered directly. If the file name contains a character such as a comma which would otherwise serve to separate file names, you may put the file name in double quotes. There is no way to use a double quote character in a file name.

You may include comments in linker scripts just as in C, delimited by '`/*`' and '`*/`'. As in C, comments are syntactically equivalent to whitespace.

3.3 Simple Linker Script Example

Many linker scripts are fairly simple.

The simplest possible linker script has just one command: '`SECTIONS`'. You use the '`SECTIONS`' command to describe the memory layout of the output file.

The '`SECTIONS`' command is a powerful command. Here we will describe a simple use of it. Let's assume your program consists only of code, initialized data, and uninitialized data. These will be in the '`.text`', '`.data`', and '`.bss`' sections, respectively. Let's assume further that these are the only sections which appear in your input files.

For this example, let's say that the code should be loaded at address 0x10000, and that the data should start at address 0x8000000. Here is a linker script which will do that:

```
SECTIONS
{
  . = 0x10000;
  .text : { *(.text) }
  . = 0x8000000;
  .data : { *(.data) }
  .bss : { *(.bss) }
}
```

You write the '`SECTIONS`' command as the keyword '`SECTIONS`', followed by a series of symbol assignments and output section descriptions enclosed in curly braces.

The first line inside the '`SECTIONS`' command of the above example sets the value of the special symbol '`.`', which is the location counter. If you do not specify the address of an output section in some other way (other ways are described later), the address is set from the current value of the location counter. The location counter is then incremented by the size of the output section. At the start of the '`SECTIONS`' command, the location counter has the value '`0`'.

The second line defines an output section, '`.text`'. The colon is required syntax which may be ignored for now. Within the curly braces after the output section name, you list the names of the input sections which should be placed into this output section. The '`*`' is a wildcard which matches any file name. The expression '`*(.text)`' means all '`.text`' input sections in all input files.

Since the location counter is '`0x10000`' when the output section '`.text`' is defined, the linker will set the address of the '`.text`' section in the output file to be '`0x10000`'.

The remaining lines define the '`.data`' and '`.bss`' sections in the output file. The linker will place the '`.data`' output section at address '`0x8000000`'. After the linker places the '`.data`' output section, the value of the location counter will be '`0x8000000`' plus the size of the '`.data`' output section. The effect is that the linker will place the '`.bss`' output section immediately after the '`.data`' output section in memory.

The linker will ensure that each output section has the required alignment, by increasing the location counter if necessary. In this example, the specified addresses for the '`.text`' and '`.data`' sections will probably satisfy any alignment constraints, but the linker may have to create a small gap between the '`.data`' and '`.bss`' sections.

That's it! That's a simple and complete linker script.

3.4 Simple Linker Script Commands

In this section we describe the simple linker script commands.

3.4.1 Setting the Entry Point

The first instruction to execute in a program is called the *entry point*. You can use the `ENTRY` linker script command to set the entry point. The argument is a symbol name:

```
ENTRY(symbol)
```

There are several ways to set the entry point. The linker will set the entry point by trying each of the following methods in order, and stopping when one of them succeeds:

- the '`-e`' *entry* command-line option;
- the `ENTRY(symbol)` command in a linker script;
- the value of the symbol `start`, if defined;
- the address of the first byte of the '`.text`' section, if present;
- The address `0`.

3.4.2 Commands Dealing with Files

Several linker script commands deal with files.

`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 ...)`

The `INPUT` command directs the linker to include the named files in the link, as though they were named on the command line.

For example, if you always want to include '`subr.o`' any time you do a link, but you can't be bothered to put it on every link command line, then you can put '`INPUT (subr.o)`' in your linker script.

In fact, if you like, you can list all of your input files in the linker script, and then invoke the linker with nothing but a '`-T`' option.

In case a *sysroot prefix* is configured, and the filename starts with the '`/`' character, and the script being processed was located inside the *sysroot prefix*, the filename will be looked for in the *sysroot prefix*. Otherwise, the linker will try to open the file in the current directory. If it is not found, the linker will search through the archive library search path. See the description of '`-L`' in Section 2.1 [Command Line Options], page 3.

If you use '`INPUT (-l`*file*`)`', `ld` will transform the name to `lib`*file*`.a`, as with the command line argument '`-l`'.

When you use the `INPUT` command in an implicit linker script, the files will be included in the link at the point at which the linker script file is included. This can affect archive searching.

`GROUP(`*file*`,` *file*`, ...)`
`GROUP(`*file* *file* `...)`

The `GROUP` command is like `INPUT`, except that the named files should all be archives, and they are searched repeatedly until no new undefined references are created. See the description of '`-(`' in Section 2.1 [Command Line Options], page 3.

`OUTPUT(`*filename*`)`

The `OUTPUT` command names the output file. Using `OUTPUT(`*filename*`)` in the linker script is exactly like using '`-o` *filename*' on the command line (see Section 2.1 [Command Line Options], page 3). If both are used, the command line option takes precedence.

You can use the `OUTPUT` command to define a default name for the output file other than the usual default of '`a.out`'.

`SEARCH_DIR(`*path*`)`

The `SEARCH_DIR` command adds *path* to the list of paths where `ld` looks for archive libraries. Using `SEARCH_DIR(`*path*`)` is exactly like using '`-L` *path*' on the command line (see Section 2.1 [Command Line Options], page 3). If both are used, then the linker will search both paths. Paths specified using the command line option are searched first.

`STARTUP(`*filename*`)`

The `STARTUP` command is just like the `INPUT` command, except that *filename* will become the first input file to be linked, as though it were specified first on the command line. This may be useful when using a system in which the entry point is always the start of the first file.

3.4.3 Commands Dealing with Object File Formats

A couple of linker script commands deal with object file formats.

`OUTPUT_FORMAT(`*bfdname*`)`
`OUTPUT_FORMAT(`*default*`, `*big*`, `*little*`)`
The `OUTPUT_FORMAT` command names the BFD format to use for the output file (see Chapter 5 [BFD], page 71). Using `OUTPUT_FORMAT(`*bfdname*`)` is exactly like using '`--oformat` *bfdname*' on the command line (see Section 2.1 [Command Line Options], page 3). If both are used, the command line option takes precedence.

You can use `OUTPUT_FORMAT` with three arguments to use different formats based on the '`-EB`' and '`-EL`' command line options. This permits the linker script to set the output format based on the desired endianness.

If neither '`-EB`' nor '`-EL`' are used, then the output format will be the first argument, *default*. If '`-EB`' is used, the output format will be the second argument, *big*. If '`-EL`' is used, the output format will be the third argument, *little*.
For example, the default linker script for the MIPS ELF target uses this command:

```
OUTPUT_FORMAT(elf32-bigmips, elf32-bigmips, elf32-littlemips)
```

This says that the default format for the output file is '`elf32-bigmips`', but if the user uses the '`-EL`' command line option, the output file will be created in the '`elf32-littlemips`' format.

`TARGET(`*bfdname*`)`
The `TARGET` command names the BFD format to use when reading input files. It affects subsequent `INPUT` and `GROUP` commands. This command is like using '`-b` *bfdname*' on the command line (see Section 2.1 [Command Line Options], page 3). If the `TARGET` command is used but `OUTPUT_FORMAT` is not, then the last `TARGET` command is also used to set the format for the output file. See Chapter 5 [BFD], page 71.

3.4.4 Other Linker Script Commands

There are a few other linker scripts commands.

`ASSERT(`*exp*`, `*message*`)`
Ensure that *exp* is non-zero. If it is zero, then exit the linker with an error code, and print *message*.

`EXTERN(`*symbol symbol ...*`)`
Force *symbol* to be entered in the output file as an undefined symbol. Doing this may, for example, trigger linking of additional modules from standard libraries. You may list several *symbol*s for each `EXTERN`, and you may use `EXTERN` multiple times. This command has the same effect as the '`-u`' command-line option.

`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`').

`INHIBIT_COMMON_ALLOCATION`
This command has the same effect as the '`--no-define-common`' command-line option: to make `ld` omit the assignment of addresses to common symbols even for a non-relocatable output file.

`NOCROSSREFS(`*section section ...*`)`

This command may be used to tell `ld` to issue an error about any references among certain output sections.

In certain types of programs, particularly on embedded systems when using overlays, when one section is loaded into memory, another section will not be. Any direct references between the two sections would be errors. For example, it would be an error if code in one section called a function defined in the other section.

The `NOCROSSREFS` command takes a list of output section names. If `ld` detects any cross references between the sections, it reports an error and returns a non-zero exit status. Note that the `NOCROSSREFS` command uses output section names, not input section names.

`OUTPUT_ARCH(`*bfdarch*`)`

Specify a particular output machine architecture. The argument is one of the names used by the BFD library (see Chapter 5 [BFD], page 71). You can see the architecture of an object file by using the `objdump` program with the '`-f`' option.

3.5 Assigning Values to Symbols

You may assign a value to a symbol in a linker script. This will define the symbol as a global symbol.

3.5.1 Simple Assignments

You may assign to a symbol using any of the C assignment operators:

symbol = *expression* ;
symbol += *expression* ;
symbol -= *expression* ;
symbol *= *expression* ;
symbol /= *expression* ;
symbol <<= *expression* ;
symbol >>= *expression* ;
symbol &= *expression* ;
symbol |= *expression* ;

The first case will define *symbol* to the value of *expression*. In the other cases, *symbol* must already be defined, and the value will be adjusted accordingly.

The special symbol name '`.`' indicates the location counter. You may only use this within a `SECTIONS` command.

The semicolon after *expression* is required.

Expressions are defined below; see Section 3.10 [Expressions], page 54.

You may write symbol assignments as commands in their own right, or as statements within a `SECTIONS` command, or as part of an output section description in a `SECTIONS` command.

The section of the symbol will be set from the section of the expression; for more information, see Section 3.10.6 [Expression Section], page 57.

Here is an example showing the three different places that symbol assignments may be used:

```
floating_point = 0;
SECTIONS
{
  .text :
    {
      *(.text)
      _etext = .;
    }
  _bdata = (. + 3) & ~ 3;
  .data : { *(.data) }
}
```

In this example, the symbol '`floating_point`' will be defined as zero. The symbol '`_etext`' will be defined as the address following the last '`.text`' input section. The symbol '`_bdata`' will be defined as the address following the '`.text`' output section aligned upward to a 4 byte boundary.

3.5.2 PROVIDE

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

Here is an example of using `PROVIDE` to define '`etext`':

```
SECTIONS
{
  .text :
    {
      *(.text)
      _etext = .;
      PROVIDE(etext = .);
    }
}
```

In this example, if the program defines '`_etext`' (with a leading underscore), the linker will give a multiple definition error. If, on the other hand, the program defines '`etext`' (with no leading underscore), the linker will silently use the definition in the program. If the program references '`etext`' but does not define it, the linker will use the definition in the linker script.

3.6 SECTIONS Command

The `SECTIONS` command tells the linker how to map input sections into output sections, and how to place the output sections in memory.

The format of the `SECTIONS` command is:

```
SECTIONS
{
  sections-command
  sections-command
  ...
}
```

Each *sections-command* may of be one of the following:

- an `ENTRY` command (see Section 3.4.1 [Entry command], page 33)
- a symbol assignment (see Section 3.5 [Assignments], page 36)
- an output section description
- an overlay description

The `ENTRY` command and symbol assignments are permitted inside the `SECTIONS` command for convenience in using the location counter in those commands. This can also make the linker script easier to understand because you can use those commands at meaningful points in the layout of the output file.

Output section descriptions and overlay descriptions are described below.

If you do not use a `SECTIONS` command in your linker script, the linker will place each input section into an identically named output section in the order that the sections are first encountered in the input files. If all input sections are present in the first file, for example, the order of sections in the output file will match the order in the first input file. The first section will be at address zero.

3.6.1 Output Section Description

The full description of an output section looks like this:

```
section [address] [(type)] :
  [AT(lma)] [SUBALIGN(subsection_align)]
  {
    output-section-command
    output-section-command
    ...
  } [>region] [AT>lma_region] [:phdr :phdr ...] [=fillexp]
```

Most output sections do not use most of the optional section attributes.

The whitespace around *section* is required, so that the section name is unambiguous. The colon and the curly braces are also required. The line breaks and other white space are optional.

Each *output-section-command* may be one of the following:

- a symbol assignment (see Section 3.5 [Assignments], page 36)
- an input section description (see Section 3.6.4 [Input Section], page 39)
- data values to include directly (see Section 3.6.5 [Output Section Data], page 42)
- a special output section keyword (see Section 3.6.6 [Output Section Keywords], page 43)

3.6.2 Output Section Name

The name of the output section is *section*. *section* must meet the constraints of your output format. In formats which only support a limited number of sections, such as `a.out`, the name must be one of the names supported by the format (`a.out`, for example, allows only '`.text`', '`.data`' or '`.bss`'). If the output format supports any number of sections, but with numbers and not names (as is the case for Oasys), the name should be supplied as a quoted numeric string. A section name may consist of any sequence of characters, but a name which contains any unusual characters such as commas must be quoted.

The output section name '`/DISCARD/`' is special; Section 3.6.7 [Output Section Discarding], page 44.

3.6.3 Output Section Address

The *address* is an expression for the VMA (the virtual memory address) of the output section. If you do not provide *address*, the linker will set it based on *region* if present, or otherwise based on the current value of the location counter.

If you provide *address*, the address of the output section will be set to precisely that. If you provide neither *address* nor *region*, then the address of the output section will be set to the current value of the location counter aligned to the alignment requirements of the output section. The alignment requirement of the output section is the strictest alignment of any input section contained within the output section.

For example,

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

and

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

are subtly different. The first will set the address of the '`.text`' output section to the current value of the location counter. The second will set it to the current value of the location counter aligned to the strictest alignment of a '`.text`' input section.

The *address* may be an arbitrary expression; Section 3.10 [Expressions], page 54. For example, if you want to align the section on a 0x10 byte boundary, so that the lowest four bits of the section address are zero, you could do something like this:

```
.text ALIGN(0x10) : { *(.text) }
```

This works because `ALIGN` returns the current location counter aligned upward to the specified value.

Specifying *address* for a section will change the value of the location counter.

3.6.4 Input Section Description

The most common output section command is an input section description.

The input section description is the most basic linker script operation. You use output sections to tell the linker how to lay out your program in memory. You use input section descriptions to tell the linker how to map the input files into your memory layout.

3.6.4.1 Input Section Basics

An input section description consists of a file name optionally followed by a list of section names in parentheses.

The file name and the section name may be wildcard patterns, which we describe further below (see Section 3.6.4.2 [Input Section Wildcards], page 40).

The most common input section description is to include all input sections with a particular name in the output section. For example, to include all input '`.text`' sections, you would write:

```
*(.text)
```

Here the '`*`' is a wildcard which matches any file name. To exclude a list of files from matching the file name wildcard, EXCLUDE_FILE may be used to match all files except the ones specified in the EXCLUDE_FILE list. For example:

```
(*(EXCLUDE_FILE (*crtend.o *otherfile.o) .ctors))
```

will cause all .ctors sections from all files except '`crtend.o`' and '`otherfile.o`' to be included.

There are two ways to include more than one section:

```
*(.text .rdata)
*(.text) *(.rdata)
```

The difference between these is the order in which the '`.text`' and '`.rdata`' input sections will appear in the output section. In the first example, they will be intermingled, appearing in the same order as they are found in the linker input. In the second example, all '`.text`' input sections will appear first, followed by all '`.rdata`' input sections.

You can specify a file name to include sections from a particular file. You would do this if one or more of your files contain special data that needs to be at a particular location in memory. For example:

```
data.o(.data)
```

If you use a file name without a list of sections, then all sections in the input file will be included in the output section. This is not commonly done, but it may by useful on occasion. For example:

```
data.o
```

When you use a file name which does not contain any wild card characters, the linker will first see if you also specified the file name on the linker command line or in an `INPUT` command. If you did not, the linker will attempt to open the file as an input file, as though it appeared on the command line. Note that this differs from an `INPUT` command, because the linker will not search for the file in the archive search path.

3.6.4.2 Input Section Wildcard Patterns

In an input section description, either the file name or the section name or both may be wildcard patterns.

The file name of '`*`' seen in many examples is a simple wildcard pattern for the file name.

The wildcard patterns are like those used by the Unix shell.

'`*`' matches any number of characters

'`?`' matches any single character

'`[`*chars*`]`' matches a single instance of any of the *chars*; the '`-`' character may be used to specify a range of characters, as in '`[a-z]`' to match any lower case letter

'`\`' quotes the following character

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

File name wildcard patterns only match files which are explicitly specified on the command line or in an `INPUT` command. The linker does not search directories to expand wildcards.

If a file name matches more than one wildcard pattern, or if a file name appears explicitly and is also matched by a wildcard pattern, the linker will use the first match in the linker script. For example, this sequence of input section descriptions is probably in error, because the '`data.o`' rule will not be used:

```
.data : { *(.data) }
.data1 : { data.o(.data) }
```

Normally, the linker will place files and sections matched by wildcards in the order in which they are seen during the link. You can change this by using the `SORT_BY_NAME` keyword, which appears before a wildcard pattern in parentheses (e.g., `SORT_BY_NAME(.text*)`).

When the SORT_BY_NAME keyword is used, the linker will sort the files or sections into ascending order by name before placing them in the output file.

SORT_BY_ALIGNMENT is very similar to SORT_BY_NAME. The difference is SORT_BY_ALIGNMENT will sort sections into ascending order by alignment before placing them in the output file.

SORT is an alias for SORT_BY_NAME.

When there are nested section sorting commands in linker script, there can be at most 1 level of nesting for section sorting commands.

1. SORT_BY_NAME (SORT_BY_ALIGNMENT (wildcard section pattern)). It will sort the input sections by name first, then by alignment if 2 sections have the same name.
2. SORT_BY_ALIGNMENT (SORT_BY_NAME (wildcard section pattern)). It will sort the input sections by alignment first, then by name if 2 sections have the same alignment.
3. SORT_BY_NAME (SORT_BY_NAME (wildcard section pattern)) is treated the same as SORT_BY_NAME (wildcard section pattern).
4. SORT_BY_ALIGNMENT (SORT_BY_ALIGNMENT (wildcard section pattern)) is treated the same as SORT_BY_ALIGNMENT (wildcard section pattern).
5. All other nested section sorting commands are invalid.

When both command line section sorting option and linker script section sorting command are used, section sorting command always takes precedence over the command line option.

If the section sorting command in linker script isn't nested, the command line option will make the section sorting command to be treated as nested sorting command.

1. SORT_BY_NAME (wildcard section pattern) with '--sort-sections alignment' is equivalent to SORT_BY_NAME (SORT_BY_ALIGNMENT (wildcard section pattern)).
2. SORT_BY_ALIGNMENT (wildcard section pattern) with '--sort-section name' is equivalent to SORT_BY_ALIGNMENT (SORT_BY_NAME (wildcard section pattern)).

If the section sorting command in linker script is nested, the command line option will be ignored.

If you ever get confused about where input sections are going, use the '-M' linker option to generate a map file. The map file shows precisely how input sections are mapped to output sections.

This example shows how wildcard patterns might be used to partition files. This linker script directs the linker to place all '.text' sections in '.text' and all '.bss' sections in '.bss'. The linker will place the '.data' section from all files beginning with an upper case character in '.DATA'; for all other files, the linker will place the '.data' section in '.data'.

```
SECTIONS {
  .text : { *(.text) }
  .DATA : { [A-Z]*(.data) }
  .data : { *(.data) }
  .bss : { *(.bss) }
}
```

3.6.4.3 Input Section for Common Symbols

A special notation is needed for common symbols, because in many object file formats common symbols do not have a particular input section. The linker treats common symbols as though they are in an input section named 'COMMON'.

You may use file names with the '`COMMON`' section just as with any other input sections. You can use this to place common symbols from a particular input file in one section while common symbols from other input files are placed in another section.
In most cases, common symbols in input files will be placed in the '`.bss`' section in the output file. For example:

```
.bss { *(.bss) *(COMMON) }
```

Some object file formats have more than one type of common symbol. For example, the MIPS ELF object file format distinguishes standard common symbols and small common symbols. In this case, the linker will use a different special section name for other types of common symbols. In the case of MIPS ELF, the linker uses '`COMMON`' for standard common symbols and '`.scommon`' for small common symbols. This permits you to map the different types of common symbols into memory at different locations.

You will sometimes see '`[COMMON]`' in old linker scripts. This notation is now considered obsolete. It is equivalent to '`*(COMMON)`'.

3.6.4.4 Input Section and Garbage Collection

When link-time garbage collection is in use ('`--gc-sections`'), it is often useful to mark sections that should not be eliminated. This is accomplished by surrounding an input section's wildcard entry with `KEEP()`, as in `KEEP(*(.init))` or `KEEP(SORT_BY_NAME(*)(.ctors))`.

3.6.4.5 Input Section Example

The following example is a complete linker script. It tells the linker to read all of the sections from file '`all.o`' and place them at the start of output section '`outputa`' which starts at location '`0x10000`'. All of section '`.input1`' from file '`foo.o`' follows immediately, in the same output section. All of section '`.input2`' from '`foo.o`' goes into output section '`outputb`', followed by section '`.input1`' from '`foo1.o`'. All of the remaining '`.input1`' and '`.input2`' sections from any files are written to output section '`outputc`'.

```
SECTIONS {
  outputa 0x10000 :
    {
    all.o
    foo.o (.input1)
    }
  outputb :
    {
    foo.o (.input2)
    foo1.o (.input1)
    }
  outputc :
    {
    *(.input1)
    *(.input2)
    }
}
```

3.6.5 Output Section Data

You can include explicit bytes of data in an output section by using `BYTE`, `SHORT`, `LONG`, `QUAD`, or `SQUAD` as an output section command. Each keyword is followed by an expression in parentheses providing the value to store (see Section 3.10 [Expressions], page 54). The value of the expression is stored at the current value of the location counter.

The BYTE, SHORT, LONG, and QUAD commands store one, two, four, and eight bytes (respectively). After storing the bytes, the location counter is incremented by the number of bytes stored.

For example, this will store the byte 1 followed by the four byte value of the symbol 'addr':

```
BYTE(1)
LONG(addr)
```

When using a 64 bit host or target, QUAD and SQUAD are the same; they both store an 8 byte, or 64 bit, value. When both host and target are 32 bits, an expression is computed as 32 bits. In this case QUAD stores a 32 bit value zero extended to 64 bits, and SQUAD stores a 32 bit value sign extended to 64 bits.

If the object file format of the output file has an explicit endianness, which is the normal case, the value will be stored in that endianness. When the object file format does not have an explicit endianness, as is true of, for example, S-records, the value will be stored in the endianness of the first input object file.

Note—these commands only work inside a section description and not between them, so the following will produce an error from the linker:

```
SECTIONS { .text : { *(.text) } LONG(1) .data : { *(.data) } }
```

whereas this will work:

```
SECTIONS { .text : { *(.text) ; LONG(1) } .data : { *(.data) } }
```

You may use the FILL command to set the fill pattern for the current section. It is followed by an expression in parentheses. Any otherwise unspecified regions of memory within the section (for example, gaps left due to the required alignment of input sections) are filled with the value of the expression, repeated as necessary. A FILL statement covers memory locations after the point at which it occurs in the section definition; by including more than one FILL statement, you can have different fill patterns in different parts of an output section.

This example shows how to fill unspecified regions of memory with the value '0x90':

```
FILL(0x90909090)
```

The FILL command is similar to the '*=fillexp*' output section attribute, but it only affects the part of the section following the FILL command, rather than the entire section. If both are used, the FILL command takes precedence. See Section 3.6.8.6 [Output Section Fill], page 47, for details on the fill expression.

3.6.6 Output Section Keywords

There are a couple of keywords which can appear as output section commands.

CREATE_OBJECT_SYMBOLS

> The command tells the linker to create a symbol for each input file. The name of each symbol will be the name of the corresponding input file. The section of each symbol will be the output section in which the CREATE_OBJECT_SYMBOLS command appears.
>
> This is conventional for the a.out object file format. It is not normally used for any other object file format.

CONSTRUCTORS

> When linking using the a.out object file format, the linker uses an unusual set construct to support C++ global constructors and destructors. When linking

object file formats which do not support arbitrary sections, such as ECOFF and XCOFF, the linker will automatically recognize C++ global constructors and destructors by name. For these object file formats, the `CONSTRUCTORS` command tells the linker to place constructor information in the output section where the `CONSTRUCTORS` command appears. The `CONSTRUCTORS` command is ignored for other object file formats.

The symbol `__CTOR_LIST__` marks the start of the global 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++ normally calls constructors from a subroutine `__main`; a call to `__main` is automatically inserted into the startup code for `main`. GNU C++ normally runs destructors either by using `atexit`, or directly from the function `exit`.

For object file formats such as `COFF` or `ELF` which support arbitrary section names, 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__ = .;
```

If you are using the GNU C++ support for initialization priority, which provides some control over the order in which global constructors are run, you must sort the constructors at link time to ensure that they are executed in the correct order. When using the `CONSTRUCTORS` command, use '`SORT_BY_NAME(CONSTRUCTORS)`' instead. When using the `.ctors` and `.dtors` sections, use '`*(SORT_BY_NAME(.ctors))`' and '`*(SORT_BY_NAME(.dtors))`' instead of just '`*(.ctors)`' and '`*(.dtors)`'.

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.

3.6.7 Output Section Discarding

The linker will not create output section which do not have any contents. This is for convenience when referring to input sections that may or may not be present in any of the input files. For example:

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

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

If you use anything other than an input section description as an output section command, such as a symbol assignment, then the output section will always be created, even if there are no matching input sections.

The special output section name '`/DISCARD/`' may be used to discard input sections. Any input sections which are assigned to an output section named '`/DISCARD/`' are not included in the output file.

3.6.8 Output Section Attributes

We showed above that the full description of an output section looked like this:

```
section [address] [(type)] :
  [AT(lma)] [SUBALIGN(subsection_align)]
  {
    output-section-command
    output-section-command
    ...
  } [>region] [AT>lma_region] [:phdr :phdr ...] [=fillexp]
```

We've already described *section*, *address*, and *output-section-command*. In this section we will describe the remaining section attributes.

3.6.8.1 Output Section Type

Each output section may have a type. The type is a keyword in parentheses. The following types are defined:

`NOLOAD` The section should be marked as not loadable, so that it will not be loaded into memory when the program is run.

`DSECT`
`COPY`
`INFO`
`OVERLAY` These type names are supported for backward compatibility, and are rarely used. They all have the same effect: the section should be marked as not allocatable, so that no memory is allocated for the section when the program is run.

The linker normally sets the attributes of an output section based on the input sections which map into it. You can override this by using the section type. 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) : { ... }
  ...
}
```

3.6.8.2 Output Section LMA

Every section has a virtual address (VMA) and a load address (LMA); see Section 3.1 [Basic Script Concepts], page 31. The address expression which may appear in an output section description sets the VMA (see Section 3.6.3 [Output Section Address], page 39).

The linker will normally set the LMA equal to the VMA. You can change that by using the `AT` keyword. The expression *lma* that follows the `AT` keyword specifies the load address of the section.

Alternatively, with '`AT>lma_region`' expression, you may specify a memory region for the section's load address. See Section 3.7 [MEMORY], page 48. Note that if the section has not had a VMA assigned to it then the linker will use the *lma_region* as the VMA region as well. See Section 3.6.8.4 [Output Section Region], page 46.

This feature is designed to make it easy to build a ROM image. For example, the following linker script creates three output sections: one called '`.text`', which starts at `0x1000`, one called '`.mdata`', which is loaded at the end of the '`.text`' section even though its VMA is `0x2000`, and one called '`.bss`' to hold uninitialized data at address `0x3000`. The symbol `_data` is defined with the value `0x2000`, which shows that the location counter holds the VMA value, not the LMA value.

```
SECTIONS
  {
  .text 0x1000 : { *(.text) _etext = . ; }
  .mdata 0x2000 :
    AT ( ADDR (.text) + SIZEOF (.text) )
    { _data = . ; *(.data); _edata = . ;  }
  .bss 0x3000 :
    { _bstart = . ;  *(.bss) *(COMMON) ; _bend = . ;}
}
```

The run-time initialization code for use with a program generated with this linker script would include something like the following, to copy the initialized data from the ROM image to its runtime address. Notice how this code takes advantage of the symbols defined by the linker script.

```
extern char _etext, _data, _edata, _bstart, _bend;
char *src = &_etext;
char *dst = &_data;

/* ROM has data at end of text; copy it. */
while (dst < &_edata) {
  *dst++ = *src++;
}

/* Zero bss */
for (dst = &_bstart; dst< &_bend; dst++)
  *dst = 0;
```

3.6.8.3 Forced Input Alignment

You can force input section alignment within an output section by using SUBALIGN. The value specified overrides any alignment given by input sections, whether larger or smaller.

3.6.8.4 Output Section Region

You can assign a section to a previously defined region of memory by using '`>region`'. See Section 3.7 [MEMORY], page 48.

Here is a simple example:

```
MEMORY { rom : ORIGIN = 0x1000, LENGTH = 0x1000 }
SECTIONS { ROM : { *(.text) } >rom }
```

3.6.8.5 Output Section Phdr

You can assign a section to a previously defined program segment by using '`:phdr`'. See Section 3.8 [PHDRS], page 50. 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. You can use `:NONE` to tell the linker to not put the section in any segment at all.
Here is a simple example:

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

3.6.8.6 Output Section Fill

You can set the fill pattern for an entire section by using '`=fillexp`'. *fillexp* is an expression (see Section 3.10 [Expressions], page 54). Any otherwise unspecified regions of memory within the output section (for example, gaps left due to the required alignment of input sections) will be filled with the value, repeated as necessary. If the fill expression is a simple hex number, ie. a string of hex digit starting with '`0x`' and without a trailing '`k`' or '`M`', then an arbitrarily long sequence of hex digits can be used to specify the fill pattern; Leading zeros become part of the pattern too. For all other cases, including extra parentheses or a unary `+`, the fill pattern is the four least significant bytes of the value of the expression. In all cases, the number is big-endian.

You can also change the fill value with a `FILL` command in the output section commands; (see Section 3.6.5 [Output Section Data], page 42).
Here is a simple example:

```
SECTIONS { .text : { *(.text) } =0x90909090 }
```

3.6.9 Overlay Description

An overlay description 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.
Overlays are described using the `OVERLAY` command. The `OVERLAY` command is used within a `SECTIONS` command, like an output section description. The full syntax of the `OVERLAY` command is as follows:

```
OVERLAY [start] : [NOCROSSREFS] [AT ( ldaddr )]
  {
    secname1
      {
        output-section-command
        output-section-command
        ...
      } [:phdr...] [=fill]
    secname2
      {
        output-section-command
        output-section-command
        ...
      } [:phdr...] [=fill]
    ...
  } [>region] [:phdr...] [=fill]
```

Everything is optional except `OVERLAY` (a keyword), and each section must have a name (*secname1* and *secname2* above). The section definitions within the `OVERLAY` construct are identical to those within the general `SECTIONS` contruct (see Section 3.6 [SECTIONS],

page 37), except that no addresses and no memory regions may be defined for sections within an `OVERLAY`.

The sections are all defined with the same starting address. The load addresses of the sections are arranged such that they are consecutive in memory starting at the load address used for the `OVERLAY` as a whole (as with normal section definitions, the load address is optional, and defaults to the start address; the start address is also optional, and defaults to the current value of the location counter).

If the `NOCROSSREFS` keyword is used, and there any references among the sections, the linker will report an error. Since the sections all run at the same address, it normally does not make sense for one section to refer directly to another. See Section 3.4.4 [Miscellaneous Commands], page 35.

For each section within the `OVERLAY`, the linker automatically defines two symbols. The symbol `__load_start_`*secname* is defined as the starting load address of the section. The symbol `__load_stop_`*secname* is defined as the final load address of the section. Any characters within *secname* which are not legal within C identifiers are removed. C (or assembler) code may use these symbols to move the overlaid sections around as necessary.

At the end of the overlay, the value of the location counter is set to the start address of the overlay plus the size of the largest section.

Here is an example. Remember that this would appear inside a `SECTIONS` construct.

```
OVERLAY 0x1000 : AT (0x4000)
 {
   .text0 { o1/*.o(.text) }
   .text1 { o2/*.o(.text) }
 }
```

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

C code to copy overlay `.text1` into the overlay area might look like the following.

```
extern char __load_start_text1, __load_stop_text1;
memcpy ((char *) 0x1000, &__load_start_text1,
        &__load_stop_text1 - &__load_start_text1);
```

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

```
.text0 0x1000 : AT (0x4000) { o1/*.o(.text) }
__load_start_text0 = LOADADDR (.text0);
__load_stop_text0 = LOADADDR (.text0) + SIZEOF (.text0);
.text1 0x1000 : AT (0x4000 + SIZEOF (.text0)) { o2/*.o(.text) }
__load_start_text1 = LOADADDR (.text1);
__load_stop_text1 = LOADADDR (.text1) + SIZEOF (.text1);
. = 0x1000 + MAX (SIZEOF (.text0), SIZEOF (.text1));
```

3.7 MEMORY Command

The linker's default configuration permits allocation of all available memory. You can override this by using the `MEMORY` command.

The `MEMORY` command describes the location and size of blocks of memory in the target. You can use it to describe which memory regions may be used by the linker, and which memory regions it must avoid. You can then assign sections to particular memory regions. The linker will set section addresses based on the memory regions, and will warn about regions that become too full. The linker will not shuffle sections around to fit into the available regions.

A linker script may contain at most one use of the `MEMORY` command. However, you can define as many blocks of memory within it as you wish. The syntax is:

```
MEMORY
  {
    name [(attr)] : ORIGIN = origin, LENGTH = len
    ...
  }
```

The *name* is a name used in the linker script to refer to the region. The region name has no meaning outside of the linker script. Region names are stored in a separate name space, and will not conflict with symbol names, file names, or section names. Each memory region must have a distinct name.

The *attr* string is an optional list of attributes that specify whether to use a particular memory region for an input section which is not explicitly mapped in the linker script. As described in Section 3.6 [SECTIONS], page 37, if you do not specify an output section for some input section, the linker will create an output section with the same name as the input section. If you define region attributes, the linker will use them to select the memory region for the output section that it creates.

The *attr* string must consist only of the following characters:

‘`R`’ Read-only section

‘`W`’ Read/write section

‘`X`’ Executable section

‘`A`’ Allocatable section

‘`I`’ Initialized section

‘`L`’ Same as ‘`I`’

‘`!`’ Invert the sense of any of the preceding attributes

If a unmapped section matches any of the listed attributes other than ‘`!`’, it will be placed in the memory region. The ‘`!`’ attribute reverses this test, so that an unmapped section will be placed in the memory region only if it does not match any of the listed attributes.

The *origin* is an numerical expression for the start address of the memory region. The expression must evaluate to a constant and it cannot involve any symbols. The keyword `ORIGIN` may be abbreviated to `org` or `o` (but not, for example, `ORG`).

The *len* is an expression for the size in bytes of the memory region. As with the *origin* expression, the expression must be numerical only and must evaluate to a constant. The keyword `LENGTH` may be abbreviated to `len` or `l`.

In the following example, we specify that there are two memory regions available for allocation: one starting at ‘`0`’ for 256 kilobytes, and the other starting at ‘`0x40000000`’ for four megabytes. The linker will place into the ‘`rom`’ memory region every section which is not

explicitly mapped into a memory region, and is either read-only or executable. The linker will place other sections which are not explicitly mapped into a memory region into the '`ram`' memory region.

```
MEMORY
  {
    rom (rx)  : ORIGIN = 0, LENGTH = 256K
    ram (!rx) : org = 0x40000000, l = 4M
  }
```

Once you define a memory region, you can direct the linker to place specific output sections into that memory region by using the '`>region`' output section attribute. For example, if you have a memory region named '`mem`', you would use '`>mem`' in the output section definition. See Section 3.6.8.4 [Output Section Region], page 46. If no address was specified for the output section, the linker will set the address to the next available address within the memory region. If the combined output sections directed to a memory region are too large for the region, the linker will issue an error message.

It is possible to access the origin and length of a memory in an expression via the `ORIGIN(memory)` and `LENGTH(memory)` functions:

```
_fstack = ORIGIN(ram) + LENGTH(ram) - 4;
```

3.8 PHDRS Command

The ELF object file format uses *program headers*, also knows as *segments*. The program headers describe how the program should be loaded into memory. You can print them out by using the `objdump` program with the '`-p`' option.

When you run an ELF program on a native ELF system, the system loader reads the program headers in order to figure out how to load the program. This will only work if the program headers are set correctly. This manual does not describe the details of how the system loader interprets program headers; for more information, see the ELF ABI.

The linker will create reasonable program headers by default. However, in some cases, you may need to specify the program headers more precisely. You may use the `PHDRS` command for this purpose. When the linker sees the `PHDRS` command in the linker script, it will not create any program headers other than the ones specified.

The linker only pays attention to the `PHDRS` command when generating an ELF output file. In other cases, the linker will simply ignore `PHDRS`.

This is the syntax of the `PHDRS` command. The words `PHDRS`, `FILEHDR`, `AT`, and `FLAGS` are keywords.

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

The *name* is used only for reference in the `SECTIONS` command of the linker script. It is not put into the output file. Program header names are stored in a separate name space, and will not conflict with symbol names, file names, or section names. Each program header must have a distinct name.

Certain program header types describe segments of memory which the system loader will load from the file. In the linker script, you specify the contents of these segments by placing

allocatable output sections in the segments. You use the '`:phdr`' output section attribute to place a section in a particular segment. See Section 3.6.8.5 [Output Section Phdr], page 46.

It is normal to put certain sections in more than one segment. This merely implies that one segment of memory contains another. You may repeat '`:phdr`', using it once for each segment which should contain the section.

If you place a section in one or more segments using '`:phdr`', then the linker will place all subsequent allocatable sections which do not specify '`:phdr`' in the same segments. This is for convenience, since generally a whole set of contiguous sections will be placed in a single segment. You can use `:NONE` to override the default segment and tell the linker to not put the section in any segment at all.

You may use the `FILEHDR` and `PHDRS` keywords appear after the program header type to further describe the contents of the segment. The `FILEHDR` keyword means that the segment should include the ELF file header. The `PHDRS` keyword means that the segment should include the ELF program headers themselves.

The *type* may be one of the following. The numbers indicate the value of the keyword.

`PT_NULL` (0)
: Indicates an unused program header.

`PT_LOAD` (1)
: Indicates that this program header describes a segment to be loaded from the file.

`PT_DYNAMIC` (2)
: Indicates a segment where dynamic linking information can be found.

`PT_INTERP` (3)
: Indicates a segment where the name of the program interpreter may be found.

`PT_NOTE` (4)
: Indicates a segment holding note information.

`PT_SHLIB` (5)
: A reserved program header type, defined but not specified by the ELF ABI.

`PT_PHDR` (6)
: Indicates a segment where the program headers may be found.

expression
: An expression giving the numeric type of the program header. This may be used for types not defined above.

You can specify that a segment should be loaded at a particular address in memory by using an `AT` expression. This is identical to the `AT` command used as an output section attribute (see Section 3.6.8.2 [Output Section LMA], page 45). The `AT` command for a program header overrides the output section attribute.

The linker will normally set the segment flags based on the sections which comprise the segment. You may use the `FLAGS` keyword to explicitly specify the segment flags. The value of *flags* must be an integer. It is used to set the `p_flags` field of the program header.

Here is an example of `PHDRS`. This shows a typical set of program headers used on a native ELF system.

```
PHDRS
{
  headers PT_PHDR PHDRS ;
  interp PT_INTERP ;
  text PT_LOAD FILEHDR PHDRS ;
  data PT_LOAD ;
  dynamic PT_DYNAMIC ;
}

SECTIONS
{
  . = SIZEOF_HEADERS;
  .interp : { *(.interp) } :text :interp
  .text : { *(.text) } :text
  .rodata : { *(.rodata) } /* defaults to :text */
  ...
  . = . + 0x1000; /* move to a new page in memory */
  .data : { *(.data) } :data
  .dynamic : { *(.dynamic) } :data :dynamic
  ...
}
```

3.9 VERSION Command

The linker supports symbol versions when using ELF. Symbol versions are only useful when using shared libraries. The dynamic linker can use symbol versions to select a specific version of a function when it runs a program that may have been linked against an earlier version of the shared library.

You can include a version script directly in the main linker script, or you can supply the version script as an implicit linker script. You can also use the '`--version-script`' linker option.

The syntax of the `VERSION` command is simply

```
VERSION { version-script-commands }
```

The format of the version script commands is identical to that used by Sun's linker in Solaris 2.5. The version script defines a tree of version nodes. You specify the node names and interdependencies in the version script. You can specify which symbols are bound to which version nodes, and you can reduce a specified set of symbols to local scope so that they are not globally visible outside of the shared library.

The easiest way to demonstrate the version script language is with a few examples.

```
VERS_1.1 {
 global:
 foo1;
 local:
 old*;
 original*;
 new*;
};
```

```
VERS_1.2 {
 foo2;
} VERS_1.1;

VERS_2.0 {
 bar1; bar2;
} VERS_1.2;
```

This example version script defines three version nodes. The first version node defined is '`VERS_1.1`'; it has no other dependencies. The script binds the symbol '`foo1`' to '`VERS_1.1`'. It reduces a number of symbols to local scope so that they are not visible outside of the shared library; this is done using wildcard patterns, so that any symbol whose name begins with '`old`', '`original`', or '`new`' is matched. The wildcard patterns available are the same as those used in the shell when matching filenames (also known as "globbing").

Next, the version script defines node '`VERS_1.2`'. This node depends upon '`VERS_1.1`'. The script binds the symbol '`foo2`' to the version node '`VERS_1.2`'.

Finally, the version script defines node '`VERS_2.0`'. This node depends upon '`VERS_1.2`'. The scripts binds the symbols '`bar1`' and '`bar2`' are bound to the version node '`VERS_2.0`'.

When the linker finds a symbol defined in a library which is not specifically bound to a version node, it will effectively bind it to an unspecified base version of the library. You can bind all otherwise unspecified symbols to a given version node by using '`global: *;`' somewhere in the version script.

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

Node name can be omited, provided it is the only version node in the version script. Such version script doesn't assign any versions to symbols, only selects which symbols will be globally visible out and which won't.

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

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

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

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

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

in the C source file. This renames the function '`original_foo`' to be an alias for '`foo`' bound to the version node '`VERS_1.1`'. The '`local:`' directive can be used to prevent the symbol '`original_foo`' from being exported. A '`.symver`' directive takes precedence over a version script.

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

To do this, you must use multiple '`.symver`' directives in the source file. Here is an example:

```
__asm__(".symver original_foo,foo@");
__asm__(".symver old_foo,foo@VERS_1.1");
__asm__(".symver old_foo1,foo@VERS_1.2");
__asm__(".symver new_foo,foo@@VERS_2.0");
```

In this example, '`foo@`' represents the symbol '`foo`' bound to the unspecified base version of the symbol. The source file that contains this example would define 4 C functions: '`original_foo`', '`old_foo`', '`old_foo1`', and '`new_foo`'.

When you have multiple definitions of a given symbol, there needs to be some way to specify a default version to which external references to this symbol will be bound. You can do this with the '`foo@@VERS_2.0`' type of '`.symver`' directive. You can only declare one version of a symbol as the default in this manner; otherwise you would effectively have multiple definitions of the same symbol.

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

You can also specify the language in the version script:

```
VERSION extern "lang" { version-script-commands }
```

The supported '`lang`'s are '`C`', '`C++`', and '`Java`'. The linker will iterate over the list of symbols at the link time and demangle them according to '`lang`' before matching them to the patterns specified in '`version-script-commands`'.

3.10 Expressions in Linker Scripts

The syntax for expressions in the linker script language is identical to that of C expressions. All expressions are evaluated as integers. All expressions are evaluated in the same size, which is 32 bits if both the host and target are 32 bits, and is otherwise 64 bits.

You can use and set symbol values in expressions.

The linker defines several special purpose builtin functions for use in expressions.

3.10.1 Constants

All constants are integers.

As in C, the linker considers an integer beginning with '`0`' to be octal, and an integer beginning with '`0x`' or '`0X`' to be hexadecimal. The linker considers other integers to be decimal.

In addition, you can use the suffixes `K` and `M` to scale a constant by 1024 or 1024^2 respectively. For example, the following all refer to the same quantity:

```
_fourk_1 = 4K;
_fourk_2 = 4096;
_fourk_3 = 0x1000;
```

3.10.2 Symbol Names

Unless quoted, symbol names start with a letter, underscore, or period and may include letters, digits, underscores, periods, and hyphens. Unquoted symbol names must not conflict with any keywords. You can specify a symbol which contains odd characters or has the same name as a keyword by surrounding the symbol name in double quotes:

```
"SECTION" = 9;
"with a space" = "also with a space" + 10;
```

Since symbols can contain many non-alphabetic characters, it is safest to delimit symbols with spaces. For example, '`A-B`' is one symbol, whereas '`A - B`' is an expression involving subtraction.

3.10.3 The Location Counter

The special linker variable *dot* '`.`' always contains the current output location counter. Since the `.` always refers to a location in an output section, it may only appear in an expression within a `SECTIONS` command. The `.` symbol may appear anywhere that an ordinary symbol is allowed in an expression.

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

```
SECTIONS
{
  output :
    {
      file1(.text)
      . = . + 1000;
      file2(.text)
      . += 1000;
      file3(.text)
    } = 0x12345678;
}
```

In the previous example, the '`.text`' section from '`file1`' is located at the beginning of the output section '`output`'. It is followed by a 1000 byte gap. Then the '`.text`' section from '`file2`' appears, also with a 1000 byte gap following before the '`.text`' section from '`file3`'. The notation '`= 0x12345678`' specifies what data to write in the gaps (see Section 3.6.8.6 [Output Section Fill], page 47).

Note: `.` actually refers to the byte offset from the start of the current containing object. Normally this is the `SECTIONS` statement, whose start address is 0, hence `.` can be used as an absolute address. If `.` is used inside a section description however, it refers to the byte offset from the start of that section, not an absolute address. Thus in a script like this:

```
SECTIONS
{
    . = 0x100
    .text: {
      *(.text)
      . = 0x200
    }
    . = 0x500
    .data: {
```

```
        *(.data)
        . += 0x600
      }
}
```

The '`.text`' section will be assigned a starting address of 0x100 and a size of exactly 0x200 bytes, even if there is not enough data in the '`.text`' input sections to fill this area. (If there is too much data, an error will be produced because this would be an attempt to move `.` backwards). The '`.data`' section will start at 0x500 and it will have an extra 0x600 bytes worth of space after the end of the values from the '`.data`' input sections and before the end of the '`.data`' output section itself.

3.10.4 Operators

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

Precedence	Associativity	Operators
highest		
1	left	`- ~ !` †
2	left	`* / %`
3	left	`+ -`
4	left	`>> <<`
5	left	`== != > < <= >=`
6	left	`&`
7	left	`|`
8	left	`&&`
9	left	`||`
10	right	`? :`
11	right	`&= += -= *= /=` ‡
lowest		

† Prefix operators.
‡ See Section 3.5 [Assignments], page 36.

3.10.5 Evaluation

The linker evaluates expressions lazily. It only computes the value of an expression when absolutely necessary.

The linker needs some information, such as the value of the start address of the first section, and the origins and lengths of memory regions, in order to do any linking at all. These values are computed as soon as possible when the linker reads in the linker script.

However, other values (such as symbol values) are not known or needed until after storage allocation. Such values are evaluated later, when other information (such as the sizes of output sections) is available for use in the symbol assignment expression.

The sizes of sections cannot be known until after allocation, so assignments dependent upon these are not performed until after allocation.

Some expressions, such as those depending upon the location counter '`.`', must be evaluated during section allocation.

If the result of an expression is required, but the value is not available, then an error results. For example, a script like the following

```
SECTIONS
  {
    .text 9+this_isnt_constant :
      { *(.text) }
  }
```

will cause the error message '`non constant expression for initial address`'.

3.10.6 The Section of an Expression

When the linker evaluates an expression, the result is either absolute or relative to some section. A relative expression is expressed as a fixed offset from the base of a section.

The position of the expression within the linker script determines whether it is absolute or relative. An expression which appears within an output section definition is relative to the base of the output section. An expression which appears elsewhere will be absolute.

A symbol set to a relative expression will be relocatable if you request relocatable output using the '`-r`' option. That means that a further link operation may change the value of the symbol. The symbol's section will be the section of the relative expression.

A symbol set to an absolute expression will retain the same value through any further link operation. The symbol will be absolute, and will not have any particular associated section.

You can use the builtin function `ABSOLUTE` to force an expression to be absolute when it would otherwise be relative. For example, to create an absolute symbol set to the address of the end of the output section '`.data`':

```
SECTIONS
  {
    .data : { *(.data) _edata = ABSOLUTE(.); }
  }
```

If '`ABSOLUTE`' were not used, '`_edata`' would be relative to the '`.data`' section.

3.10.7 Builtin Functions

The linker script language includes a number of builtin functions for use in linker script expressions.

`ABSOLUTE(`*exp*`)`
: Return the absolute (non-relocatable, as opposed to non-negative) value of the expression *exp*. Primarily useful to assign an absolute value to a symbol within a section definition, where symbol values are normally section relative. See Section 3.10.6 [Expression Section], page 57.

`ADDR(`*section*`)`
: Return the absolute address (the VMA) of the named *section*. Your script must previously have defined the location of that section. In the following example, `symbol_1` and `symbol_2` are assigned identical values:

```
SECTIONS { ...
  .output1 :
    {
    start_of_output_1 = ABSOLUTE(.);
    ...
    }
  .output :
    {
    symbol_1 = ADDR(.output1);
    symbol_2 = start_of_output_1;
    }
... }
```

`ALIGN(`*align*`)`
`ALIGN(`*exp*`,`*align*`)`

Return the location counter (`.`) or arbitrary expression aligned to the next *align* boundary. The single operand `ALIGN` doesn't change the value of the location counter—it just does arithmetic on it. The two operand `ALIGN` allows an arbitrary expression to be aligned upwards (`ALIGN(`*align*`)` is equivalent to `ALIGN(.,` *align*`)`).

Here is an example which aligns the output `.data` section to the next `0x2000` byte boundary after the preceding section and sets a variable within the section to the next `0x8000` boundary after the input sections:

```
SECTIONS { ...
  .data ALIGN(0x2000): {
    *(.data)
    variable = ALIGN(0x8000);
  }
... }
```

The first use of `ALIGN` in this example specifies the location of a section because it is used as the optional *address* attribute of a section definition (see Section 3.6.3 [Output Section Address], page 39). The second use of `ALIGN` is used to defines the value of a symbol.

The builtin function `NEXT` is closely related to `ALIGN`.

`BLOCK(`*exp*`)`

This is a synonym for `ALIGN`, for compatibility with older linker scripts. It is most often seen when setting the address of an output section.

`DATA_SEGMENT_ALIGN(`*maxpagesize*`,` *commonpagesize*`)`

This is equivalent to either

```
(ALIGN(maxpagesize) + (. & (maxpagesize - 1)))
```

or

```
(ALIGN(maxpagesize) + (. & (maxpagesize - commonpagesize)))
```

depending on whether the latter uses fewer *commonpagesize* sized pages for the data segment (area between the result of this expression and `DATA_SEGMENT_END`) than the former or not. If the latter form is used, it means *commonpagesize* bytes of runtime memory will be saved at the expense of up to *commonpagesize* wasted bytes in the on-disk file.

This expression can only be used directly in `SECTIONS` commands, not in any output section descriptions and only once in the linker script. *commonpagesize* should be less or equal to *maxpagesize* and should be the system page size the

object wants to be optimized for (while still working on system page sizes up to *maxpagesize*).

Example:

```
. = DATA_SEGMENT_ALIGN(0x10000, 0x2000);
```

`DATA_SEGMENT_END(`*exp*`)`

This defines the end of data segment for `DATA_SEGMENT_ALIGN` evaluation purposes.

```
. = DATA_SEGMENT_END(.);
```

`DATA_SEGMENT_RELRO_END(`*offset*`, `*exp*`)`

This defines the end of the `PT_GNU_RELRO` segment when '`-z relro`' option is used. Second argument is returned. When '`-z relro`' option is not present, `DATA_SEGMENT_RELRO_END` does nothing, otherwise `DATA_SEGMENT_ALIGN` is padded so that *exp* + *offset* is aligned to the most commonly used page boundary for particular target. If present in the linker script, it must always come in between `DATA_SEGMENT_ALIGN` and `DATA_SEGMENT_END`.

```
. = DATA_SEGMENT_RELRO_END(24, .);
```

`DEFINED(`*symbol*`)`

Return 1 if *symbol* is in the linker global symbol table and is defined before the statement using DEFINED in the script, otherwise return 0. You can use this function to provide default values for symbols. For example, the following script fragment shows how to set a global symbol '`begin`' to the first location in the '`.text`' section—but if a symbol called '`begin`' already existed, its value is preserved:

```
SECTIONS { ...
  .text : {
    begin = DEFINED(begin) ? begin : . ;
    ...
  }
  ...
}
```

`LENGTH(`*memory*`)`

Return the length of the memory region named *memory*.

`LOADADDR(`*section*`)`

Return the absolute LMA of the named *section*. This is normally the same as `ADDR`, but it may be different if the `AT` attribute is used in the output section definition (see Section 3.6.8.2 [Output Section LMA], page 45).

`MAX(`*exp1*`, `*exp2*`)`

Returns the maximum of *exp1* and *exp2*.

`MIN(`*exp1*`, `*exp2*`)`

Returns the minimum of *exp1* and *exp2*.

`NEXT(`*exp*`)`

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

`ORIGIN(`*memory*`)`
: Return the origin of the memory region named *memory*.

`SEGMENT_START(`*segment*`, `*default*`)`
: Return the base address of the named *segment*. If an explicit value has been given for this segment (with a command-line '`-T`' option) that value will be returned; otherwise the value will be *default*. At present, the '`-T`' command-line option can only be used to set the base address for the "text", "data", and "bss" sections, but you use `SEGMENT_START` with any segment name.

`SIZEOF(`*section*`)`
: Return the size in bytes of the named *section*, if that section has been allocated. If the section has not been allocated when this is evaluated, the linker will report an error. In the following example, `symbol_1` and `symbol_2` are assigned identical values:

```
SECTIONS{ ...
  .output {
    .start = . ;
    ...
    .end = . ;
    }
  symbol_1 = .end - .start ;
  symbol_2 = SIZEOF(.output);
... }
```

`SIZEOF_HEADERS`
`sizeof_headers`
: Return the size in bytes of the output file's headers. This is information which appears at the start of the output file. You can use this number when setting the start address of the first section, if you choose, to facilitate paging.

  When producing an ELF output file, if the linker script uses the `SIZEOF_HEADERS` builtin function, the linker must compute the number of program headers before it has determined all the section addresses and sizes. If the linker later discovers that it needs additional program headers, it will report an error '`not enough room for program headers`'. To avoid this error, you must avoid using the `SIZEOF_HEADERS` function, or you must rework your linker script to avoid forcing the linker to use additional program headers, or you must define the program headers yourself using the `PHDRS` command (see Section 3.8 [PHDRS], page 50).

3.11 Implicit Linker Scripts

If you specify a linker input file which the linker can not recognize as an object file or an archive file, it will try to read the file as a linker script. If the file can not be parsed as a linker script, the linker will report an error.

An implicit linker script will not replace the default linker script.

Typically an implicit linker script would contain only symbol assignments, or the `INPUT`, `GROUP`, or `VERSION` commands.

Any input files read because of an implicit linker script will be read at the position in the command line where the implicit linker script was read. This can affect archive searching.

4 Machine Dependent Features

`ld` has additional features on some platforms; the following sections describe them. Machines where `ld` has no additional functionality are not listed.

4.1 `ld` and the H8/300

For the H8/300, `ld` can perform these global optimizations when you specify the '`--relax`' command-line option.

relaxing address modes
: `ld` finds all `jsr` and `jmp` instructions whose targets are within eight bits, and turns them into eight-bit program-counter relative `bsr` and `bra` instructions, respectively.

synthesizing instructions
: `ld` finds all `mov.b` instructions which use the sixteen-bit absolute address form, but refer to the top page of memory, and changes them to use the eight-bit address form. (That is: the linker turns '`mov.b @aa:16`' into '`mov.b @aa:8`' whenever the address *aa* is in the top page of memory).

bit manipulation instructions
: `ld` finds all bit manipulation instructions like `band, bclr, biand, bild, bior, bist, bixor, bld, bnot, bor, bset, bst, btst, bxor` which use 32 bit and 16 bit absolute address form, but refer to the top page of memory, and changes them to use the 8 bit address form. (That is: the linker turns '`bset #xx:3,@aa:32`' into '`bset #xx:3,@aa:8`' whenever the address *aa* is in the top page of memory).

system control instructions
: `ld` finds all `ldc.w, stc.w` instrcutions which use the 32 bit absolute address form, but refer to the top page of memory, and changes them to use 16 bit address form. (That is: the linker turns '`ldc.w @aa:32,ccr`' into '`ldc.w @aa:16,ccr`' whenever the address *aa* is in the top page of memory).

4.2 `ld` and the Intel 960 Family

You can use the '`-A`*architecture*' command line option to specify one of the two-letter names identifying members of the 960 family; the option specifies the desired output target, and warns of any incompatible instructions in the input files. It also modifies the linker's search strategy for archive libraries, to support the use of libraries specific to each particular architecture, by including in the search loop names suffixed with the string identifying the architecture.

For example, if your `ld` command line included '`-ACA`' as well as '`-ltry`', the linker would look (in its built-in search paths, and in any paths you specify with '`-L`') for a library with the names

```
try
libtry.a
tryca
libtryca.a
```

The first two possibilities would be considered in any event; the last two are due to the use of '`-ACA`'.

You can meaningfully use '`-A`' more than once on a command line, since the 960 architecture family allows combination of target architectures; each use will add another pair of name variants to search for when '`-l`' specifies a library.

`ld` supports the '`--relax`' option for the i960 family. If you specify '`--relax`', `ld` finds all `balx` and `calx` instructions whose targets are within 24 bits, and turns them into 24-bit program-counter relative `bal` and `cal` instructions, respectively. `ld` also turns `cal` instructions into `bal` instructions when it determines that the target subroutine is a leaf routine (that is, the target subroutine does not itself call any subroutines).

4.3 `ld` and the ARM family

For the ARM, `ld` will generate code stubs to allow functions calls betweem ARM and Thumb code. These stubs only work with code that has been compiled and assembled with the '`-mthumb-interwork`' command line option. If it is necessary to link with old ARM object files or libraries, which have not been compiled with the -mthumb-interwork option then the '`--support-old-code`' command line switch should be given to the linker. This will make it generate larger stub functions which will work with non-interworking aware ARM code. Note, however, the linker does not support generating stubs for function calls to non-interworking aware Thumb code.

The '`--thumb-entry`' switch is a duplicate of the generic '`--entry`' switch, in that it sets the program's starting address. But it also sets the bottom bit of the address, so that it can be branched to using a BX instruction, and the program will start executing in Thumb mode straight away.

The '`--be8`' switch instructs `ld` to generate BE8 format executables. This option is only valid when linking big-endian objects. The resulting image will contain big-endian data and little-endian code.

The '`R_ARM_TARGET1`' relocation is typically used for entries in the '`.init_array`' section. It is interpreted as either '`R_ARM_REL32`' or '`R_ARM_ABS32`', depending on the target. The '`--target1-rel`' and '`--target1-abs`' switches override the default.

The '`--target2=type`' switch overrides the default definition of the '`R_ARM_TARGET2`' relocation. Valid values for '`type`', their meanings, and target defaults are as follows:

'`rel`' '`R_ARM_REL32`' (arm*-*-elf, arm*-*-eabi)

'`abs`' '`R_ARM_ABS32`' (arm*-*-symbianelf)

'`got-rel`' '`R_ARM_GOT_PREL`' (arm*-*-linux, arm*-*-*bsd)

4.4 `ld` and HPPA 32-bit ELF Support

When generating a shared library, `ld` will by default generate import stubs suitable for use with a single sub-space application. The '`--multi-subspace`' switch causes `ld` to generate export stubs, and different (larger) import stubs suitable for use with multiple sub-spaces.

Long branch stubs and import/export stubs are placed by `ld` in stub sections located between groups of input sections. '`--stub-group-size`' specifies the maximum size of a group of input sections handled by one stub section. Since branch offsets are signed, a stub

section may serve two groups of input sections, one group before the stub section, and one group after it. However, when using conditional branches that require stubs, it may be better (for branch prediction) that stub sections only serve one group of input sections. A negative value for 'N' chooses this scheme, ensuring that branches to stubs always use a negative offset. Two special values of 'N' are recognized, '1' and '-1'. These both instruct ld to automatically size input section groups for the branch types detected, with the same behaviour regarding stub placement as other positive or negative values of 'N' respectively.

Note that '--stub-group-size' does not split input sections. A single input section larger than the group size specified will of course create a larger group (of one section). If input sections are too large, it may not be possible for a branch to reach its stub.

4.5 ld and MMIX

For MMIX, there is a choice of generating ELF object files or mmo object files when linking. The simulator mmix understands the mmo format. The binutils objcopy utility can translate between the two formats.

There is one special section, the '.MMIX.reg_contents' section. Contents in this section is assumed to correspond to that of global registers, and symbols referring to it are translated to special symbols, equal to registers. In a final link, the start address of the '.MMIX.reg_contents' section corresponds to the first allocated global register multiplied by 8. Register $255 is not included in this section; it is always set to the program entry, which is at the symbol Main for mmo files.

Symbols with the prefix __.MMIX.start., for example __.MMIX.start..text and __.MMIX.start..data are special; there must be only one each, even if they are local. The default linker script uses these to set the default start address of a section.

Initial and trailing multiples of zero-valued 32-bit words in a section, are left out from an mmo file.

4.6 ld and MSP430

For the MSP430 it is possible to select the MPU architecture. The flag '-m [mpu type]' will select an appropriate linker script for selected MPU type. (To get a list of known MPUs just pass '-m help' option to the linker).

The linker will recognize some extra sections which are MSP430 specific:

'.vectors'
: Defines a portion of ROM where interrupt vectors located.

'.bootloader'
: Defines the bootloader portion of the ROM (if applicable). Any code in this section will be uploaded to the MPU.

'.infomem'
: Defines an information memory section (if applicable). Any code in this section will be uploaded to the MPU.

'.infomemnobits'
: This is the same as the '.infomem' section except that any code in this section will not be uploaded to the MPU.

‘.noinit’
: Denotes a portion of RAM located above ‘.bss’ section.

  The last two sections are used by gcc.

4.7 ld's Support for Various TI COFF Versions

The ‘--format’ switch allows selection of one of the various TI COFF versions. The latest of this writing is 2; versions 0 and 1 are also supported. The TI COFF versions also vary in header byte-order format; ld will read any version or byte order, but the output header format depends on the default specified by the specific target.

4.8 ld and WIN32 (cygwin/mingw)

This section describes some of the win32 specific ld issues. See Section 2.1 [Command Line Options], page 3 for detailed decription of the command line options mentioned here.

import libraries
: The standard Windows linker creates and uses so-called import libraries, which contains information for linking to dll's. They are regular static archives and are handled as any other static archive. The cygwin and mingw ports of ld have specific support for creating such libraries provided with the ‘--out-implib’ command line option.

exporting DLL symbols
: The cygwin/mingw ld has several ways to export symbols for dll's.

  using auto-export functionality
  : By default ld exports symbols with the auto-export functionality, which is controlled by the following command line options:

    - –export-all-symbols [This is the default]
    - –exclude-symbols
    - –exclude-libs

    If, however, ‘--export-all-symbols’ is not given explicitly on the command line, then the default auto-export behavior will be *disabled* if either of the following are true:

    - A DEF file is used.
    - Any symbol in any object file was marked with the __declspec(dllexport) attribute.

  using a DEF file
  : Another way of exporting symbols is using a DEF file. A DEF file is an ASCII file containing definitions of symbols which should be exported when a dll is created. Usually it is named ‘<dll name>.def’ and is added as any other object file to the linker's command line. The file's name must end in ‘.def’ or ‘.DEF’.

    ```
    gcc -o <output> <objectfiles> <dll name>.def
    ```

    Using a DEF file turns off the normal auto-export behavior, unless the ‘--export-all-symbols’ option is also used.

Here is an example of a DEF file for a shared library called 'xyz.dll':

```
LIBRARY "xyz.dll" BASE=0x10000000

EXPORTS
foo
bar
_bar = bar
```

This example defines a base address and three symbols. The third symbol is an alias for the second. For the complete format specification see ld/deffilep.y in the binutils sources.

While linking a shared dll, `ld` is able to create a DEF file with the '`--output-def <file>`' command line option.

Using decorations

Another way of marking symbols for export is to modify the source code itself, so that when building the DLL each symbol to be exported is declared as:

```
__declspec(dllexport) int a_variable
__declspec(dllexport) void a_function(int with_args)
```

All such symbols will be exported from the DLL. If, however, any of the object files in the DLL contain symbols decorated in this way, then the normal auto-export behavior is disabled, unless the '`--export-all-symbols`' option is also used.

Note that object files that wish to access these symbols must *not* decorate them with dllexport. Instead, they should use dllimport, instead:

```
__declspec(dllimport) int a_variable
__declspec(dllimport) void a_function(int with_args)
```

This complicates the structure of library header files, because when included by the library itself the header must declare the variables and functions as dllexport, but when included by client code the header must declare them as dllimport. There are a number of idioms that are typically used to do this; often client code can omit the __declspec() declaration completely. See '`--enable-auto-import`' and '`automatic data imports`' for more imformation.

automatic data imports

The standard Windows dll format supports data imports from dlls only by adding special decorations (dllimport/dllexport), which let the compiler produce specific assembler instructions to deal with this issue. This increases the effort necessary to port existing Un*x code to these platforms, especially for large c++ libraries and applications. The auto-import feature, which was initially provided by Paul Sokolovsky, allows one to omit the decorations to archieve a behavior that conforms to that on POSIX/Un*x platforms. This feature is

enabled with the '`--enable-auto-import`' command-line option, although it is enabled by default on cygwin/mingw. The '`--enable-auto-import`' option itself now serves mainly to suppress any warnings that are ordinarily emitted when linked objects trigger the feature's use.

auto-import of variables does not always work flawlessly without additional assistance. Sometimes, you will see this message

"variable '<var>' can't be auto-imported. Please read the documentation for ld's `--enable-auto-import` for details."

The '`--enable-auto-import`' documentation explains why this error occurs, and several methods that can be used to overcome this difficulty. One of these methods is the *runtime pseudo-relocs* feature, described below.

For complex variables imported from DLLs (such as structs or classes), object files typically contain a base address for the variable and an offset (*addend*) within the variable–to specify a particular field or public member, for instance. Unfortunately, the runtime loader used in win32 environments is incapable of fixing these references at runtime without the additional information supplied by dllimport/dllexport decorations. The standard auto-import feature described above is unable to resolve these references.

The '`--enable-runtime-pseudo-relocs`' switch allows these references to be resolved without error, while leaving the task of adjusting the references themselves (with their non-zero addends) to specialized code provided by the runtime environment. Recent versions of the cygwin and mingw environments and compilers provide this runtime support; older versions do not. However, the support is only necessary on the developer's platform; the compiled result will run without error on an older system.

'`--enable-runtime-pseudo-relocs`' is not the default; it must be explicitly enabled as needed.

direct linking to a dll

The cygwin/mingw ports of `ld` support the direct linking, including data symbols, to a dll without the usage of any import libraries. This is much faster and uses much less memory than does the traditional import library method, expecially when linking large libraries or applications. When `ld` creates an import lib, each function or variable exported from the dll is stored in its own bfd, even though a single bfd could contain many exports. The overhead involved in storing, loading, and processing so many bfd's is quite large, and explains the tremendous time, memory, and storage needed to link against particularly large or complex libraries when using import libs.

Linking directly to a dll uses no extra command-line switches other than '`-L`' and '`-l`', because `ld` already searches for a number of names to match each library. All that is needed from the developer's perspective is an understanding of this search, in order to force ld to select the dll instead of an import library.

For instance, when ld is called with the argument '`-lxxx`' it will attempt to find, in the first directory of its search path,

```
libxxx.dll.a
xxx.dll.a
```

```
libxxx.a
cygxxx.dll (*)
libxxx.dll
xxx.dll
```

before moving on to the next directory in the search path.

(*) Actually, this is not '`cygxxx.dll`' but in fact is '`<prefix>xxx.dll`', where '`<prefix>`' is set by the `ld` option '`--dll-search-prefix=<prefix>`'. In the case of cygwin, the standard gcc spec file includes '`--dll-search-prefix=cyg`', so in effect we actually search for '`cygxxx.dll`'.

Other win32-based unix environments, such as mingw or pw32, may use other '`<prefix>`'es, although at present only cygwin makes use of this feature. It was originally intended to help avoid name conflicts among dll's built for the various win32/un*x environments, so that (for example) two versions of a zlib dll could coexist on the same machine.

The generic cygwin/mingw path layout uses a '`bin`' directory for applications and dll's and a '`lib`' directory for the import libraries (using cygwin nomenclature):

```
bin/
cygxxx.dll
lib/
libxxx.dll.a   (in case of dll's)
libxxx.a       (in case of static archive)
```

Linking directly to a dll without using the import library can be done two ways:

1. Use the dll directly by adding the '`bin`' path to the link line

```
gcc -Wl,-verbose  -o a.exe -L../bin/ -lxxx
```

However, as the dll's often have version numbers appended to their names ('`cygncurses-5.dll`') this will often fail, unless one specifies '`-L../bin -lncurses-5`' to include the version. Import libs are generally not versioned, and do not have this difficulty.

2. Create a symbolic link from the dll to a file in the '`lib`' directory according to the above mentioned search pattern. This should be used to avoid unwanted changes in the tools needed for making the app/dll.

```
ln -s bin/cygxxx.dll lib/[cyg|lib|]xxx.dll[.a]
```

Then you can link without any make environment changes.

```
gcc -Wl,-verbose  -o a.exe -L../lib/ -lxxx
```

This technique also avoids the version number problems, because the following is perfectly legal

```
bin/
cygxxx-5.dll
lib/
libxxx.dll.a -> ../bin/cygxxx-5.dll
```

Linking directly to a dll without using an import lib will work even when auto-import features are exercised, and even when '`--enable-runtime-pseudo-relocs`' is used.

Given the improvements in speed and memory usage, one might justifiably wonder why import libraries are used at all. There are two reasons:

1. Until recently, the link-directly-to-dll functionality did *not* work with auto-imported data.

2. Sometimes it is necessary to include pure static objects within the import library (which otherwise contains only bfd's for indirection symbols that point to the exports of a dll). Again, the import lib for the cygwin kernel makes use of this ability, and it is not possible to do this without an import lib.

So, import libs are not going away. But the ability to replace true import libs with a simple symbolic link to (or a copy of) a dll, in most cases, is a useful addition to the suite of tools binutils makes available to the win32 developer. Given the massive improvements in memory requirements during linking, storage requirements, and linking speed, we expect that many developers will soon begin to use this feature whenever possible.

symbol aliasing

adding additional names

Sometimes, it is useful to export symbols with additional names. A symbol '`foo`' will be exported as '`foo`', but it can also be exported as '`_foo`' by using special directives in the DEF file when creating the dll. This will affect also the optional created import library. Consider the following DEF file:

```
LIBRARY "xyz.dll" BASE=0x61000000

EXPORTS
foo
_foo = foo
```

The line '`_foo = foo`' maps the symbol '`foo`' to '`_foo`'.

Another method for creating a symbol alias is to create it in the source code using the "weak" attribute:

```
void foo () { /* Do something.  */; }
void _foo () __attribute__ ((weak, alias ("foo")));
```

See the gcc manual for more information about attributes and weak symbols.

renaming symbols

Sometimes it is useful to rename exports. For instance, the cygwin kernel does this regularly. A symbol '`_foo`' can be exported as '`foo`' but not as '`_foo`' by using special directives in the DEF file. (This will also affect the import library, if it is created). In the following example:

```
LIBRARY "xyz.dll" BASE=0x61000000

EXPORTS
_foo = foo
```

The line '`_foo = foo`' maps the exported symbol '`foo`' to '`_foo`'.

Note: using a DEF file disables the default auto-export behavior, unless the '`--export-all-symbols`' command line option is used. If, however, you are trying to rename symbols, then you should list *all* desired exports in the DEF file, including the symbols that are not being renamed, and do *not* use the '`--export-all-symbols`' option. If you list only the renamed symbols in the DEF file, and use '`--export-all-symbols`' to handle the other symbols, then the both the new names *and* the original names for the renamed symbols will be exported. In effect, you'd be aliasing those symbols, not renaming them, which is probably not what you wanted.

weak externals

The Windows object format, PE, specifies a form of weak symbols called weak externals. When a weak symbol is linked and the symbol is not defined, the weak symbol becomes an alias for some other symbol. There are three variants of weak externals:

- Definition is searched for in objects and libraries, historically called lazy externals.
- Definition is searched for only in other objects, not in libraries. This form is not presently implemented.
- No search; the symbol is an alias. This form is not presently implemented.

As a GNU extension, weak symbols that do not specify an alternate symbol are supported. If the symbol is undefined when linking, the symbol uses a default value.

4.9 `ld` and Xtensa Processors

The default `ld` behavior for Xtensa processors is to interpret `SECTIONS` commands so that lists of explicitly named sections in a specification with a wildcard file will be interleaved when necessary to keep literal pools within the range of PC-relative load offsets. For example, with the command:

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

`ld` may interleave some of the `.literal` and `.text` sections from different object files to ensure that the literal pools are within the range of PC-relative load offsets. A valid interleaving might place the `.literal` sections from an initial group of files followed by the `.text` sections of that group of files. Then, the `.literal` sections from the rest of the files and the `.text` sections from the rest of the files would follow.

Relaxation is enabled by default for the Xtensa version of `ld` and provides two important link-time optimizations. The first optimization is to combine identical literal values to reduce code size. A redundant literal will be removed and all the `L32R` instructions that use it will be changed to reference an identical literal, as long as the location of the replacement literal is within the offset range of all the `L32R` instructions. The second optimization is to remove unnecessary overhead from assembler-generated "longcall" sequences of `L32R`/`CALLX`*n* when the target functions are within range of direct `CALL`*n* instructions.

For each of these cases where an indirect call sequence can be optimized to a direct call, the linker will change the `CALLX`*n* instruction to a `CALL`*n* instruction, remove the `L32R` instruction, and remove the literal referenced by the `L32R` instruction if it is not used for anything else. Removing the `L32R` instruction always reduces code size but can potentially hurt performance by changing the alignment of subsequent branch targets. By default, the linker will always preserve alignments, either by switching some instructions between 24-bit encodings and the equivalent density instructions or by inserting a no-op in place of the `L32R` instruction that was removed. If code size is more important than performance, the '`--size-opt`' option can be used to prevent the linker from widening density instructions or inserting no-ops, except in a few cases where no-ops are required for correctness.

The following Xtensa-specific command-line options can be used to control the linker:

'`--no-relax`'

Since the Xtensa version of `ld` enables the '`--relax`' option by default, the '`--no-relax`' option is provided to disable relaxation.

'`--size-opt`'

When optimizing indirect calls to direct calls, optimize for code size more than performance. With this option, the linker will not insert no-ops or widen density instructions to preserve branch target alignment. There may still be some cases where no-ops are required to preserve the correctness of the code.

5 BFD

The linker accesses object and archive files using the BFD libraries. These libraries allow the linker to use the same routines to operate on object files whatever the object file format. A different object file format can be supported simply by creating a new BFD back end and adding it to the library. To conserve runtime memory, however, the linker and associated tools are usually configured to support only a subset of the object file formats available. You can use `objdump -i` (see section "objdump" in *The GNU Binary Utilities*) to list all the formats available for your configuration.

As with most implementations, BFD is a compromise between several conflicting requirements. The major factor influencing BFD design was efficiency: any time used converting between formats is time which would not have been spent had BFD not been involved. This is partly offset by abstraction payback; since BFD simplifies applications and back ends, more time and care may be spent optimizing algorithms for a greater speed.

One minor artifact of the BFD solution which you should bear in mind is the potential for information loss. There are two places where useful information can be lost using the BFD mechanism: during conversion and during output. See Section 5.1.1 [BFD information loss], page 71.

5.1 How It Works: An Outline of BFD

When an object file is opened, BFD subroutines automatically determine the format of the input object file. They then build a descriptor in memory with pointers to routines that will be used to access elements of the object file's data structures.

As different information from the object files is required, BFD reads from different sections of the file and processes them. For example, a very common operation for the linker is processing symbol tables. Each BFD back end provides a routine for converting between the object file's representation of symbols and an internal canonical format. When the linker asks for the symbol table of an object file, it calls through a memory pointer to the routine from the relevant BFD back end which reads and converts the table into a canonical form. The linker then operates upon the canonical form. When the link is finished and the linker writes the output file's symbol table, another BFD back end routine is called to take the newly created symbol table and convert it into the chosen output format.

5.1.1 Information Loss

Information can be lost during output. The output formats supported by BFD do not provide identical facilities, and information which can be described in one form has nowhere to go in another format. One example of this is alignment information in `b.out`. There is nowhere in an `a.out` format file to store alignment information on the contained data, so when a file is linked from `b.out` and an `a.out` image is produced, alignment information will not propagate to the output file. (The linker will still use the alignment information internally, so the link is performed correctly).

Another example is COFF section names. COFF files may contain an unlimited number of sections, each one with a textual section name. If the target of the link is a format which does not have many sections (e.g., `a.out`) or has sections without names (e.g., the Oasys format), the link cannot be done simply. You can circumvent this problem by describing the desired input-to-output section mapping with the linker command language.

Information can be lost during canonicalization. The BFD internal canonical form of the external formats is not exhaustive; there are structures in input formats for which there is no direct representation internally. This means that the BFD back ends cannot maintain all possible data richness through the transformation between external to internal and back to external formats.

This limitation is only a problem when an application reads one format and writes another. Each BFD back end is responsible for maintaining as much data as possible, and the internal BFD canonical form has structures which are opaque to the BFD core, and exported only to the back ends. When a file is read in one format, the canonical form is generated for BFD and the application. At the same time, the back end saves away any information which may otherwise be lost. If the data is then written back in the same format, the back end routine will be able to use the canonical form provided by the BFD core as well as the information it prepared earlier. Since there is a great deal of commonality between back ends, there is no information lost when linking or copying big endian COFF to little endian COFF, or `a.out` to `b.out`. When a mixture of formats is linked, the information is only lost from the files whose format differs from the destination.

5.1.2 The BFD canonical object-file format

The greatest potential for loss of information occurs when there is the least overlap between the information provided by the source format, that stored by the canonical format, and that needed by the destination format. A brief description of the canonical form may help you understand which kinds of data you can count on preserving across conversions.

files Information stored on a per-file basis includes target machine architecture, particular implementation format type, a demand pageable bit, and a write protected bit. Information like Unix magic numbers is not stored here—only the magic numbers' meaning, so a `ZMAGIC` file would have both the demand pageable bit and the write protected text bit set. The byte order of the target is stored on a per-file basis, so that big- and little-endian object files may be used with one another.

sections Each section in the input file contains the name of the section, the section's original address in the object file, size and alignment information, various flags, and pointers into other BFD data structures.

symbols Each symbol contains a pointer to the information for the object file which originally defined it, its name, its value, and various flag bits. When a BFD back end reads in a symbol table, it relocates all symbols to make them relative to the base of the section where they were defined. Doing this ensures that each symbol points to its containing section. Each symbol also has a varying amount of hidden private data for the BFD back end. Since the symbol points to the original file, the private data format for that symbol is accessible. `ld` can operate on a collection of symbols of wildly different formats without problems.

Normal global and simple local symbols are maintained on output, so an output file (no matter its format) will retain symbols pointing to functions and to global, static, and common variables. Some symbol information is not worth retaining; in `a.out`, type information is stored in the symbol table as long

symbol names. This information would be useless to most COFF debuggers; the linker has command line switches to allow users to throw it away.

There is one word of type information within the symbol, so if the format supports symbol type information within symbols (for example, COFF, IEEE, Oasys) and the type is simple enough to fit within one word (nearly everything but aggregates), the information will be preserved.

relocation level

Each canonical BFD relocation record contains a pointer to the symbol to relocate to, the offset of the data to relocate, the section the data is in, and a pointer to a relocation type descriptor. Relocation is performed by passing messages through the relocation type descriptor and the symbol pointer. Therefore, relocations can be performed on output data using a relocation method that is only available in one of the input formats. For instance, Oasys provides a byte relocation format. A relocation record requesting this relocation type would point indirectly to a routine to perform this, so the relocation may be performed on a byte being written to a 68k COFF file, even though 68k COFF has no such relocation type.

line numbers

Object formats can contain, for debugging purposes, some form of mapping between symbols, source line numbers, and addresses in the output file. These addresses have to be relocated along with the symbol information. Each symbol with an associated list of line number records points to the first record of the list. The head of a line number list consists of a pointer to the symbol, which allows finding out the address of the function whose line number is being described. The rest of the list is made up of pairs: offsets into the section and line numbers. Any format which can simply derive this information can pass it successfully between formats (COFF, IEEE and Oasys).

6 Reporting Bugs

Your bug reports play an essential role in making `ld` reliable.

Reporting a bug may help you by bringing a solution to your problem, or it may not. But in any case the principal function of a bug report is to help the entire community by making the next version of `ld` work better. Bug reports are your contribution to the maintenance of `ld`.

In order for a bug report to serve its purpose, you must include the information that enables us to fix the bug.

6.1 Have You Found a Bug?

If you are not sure whether you have found a bug, here are some guidelines:

- If the linker gets a fatal signal, for any input whatever, that is a `ld` bug. Reliable linkers never crash.
- If `ld` produces an error message for valid input, that is a bug.
- If `ld` does not produce an error message for invalid input, that may be a bug. In the general case, the linker can not verify that object files are correct.
- If you are an experienced user of linkers, your suggestions for improvement of `ld` are welcome in any case.

6.2 How to Report Bugs

A number of companies and individuals offer support for GNU products. If you obtained `ld` from a support organization, we recommend you contact that organization first.

You can find contact information for many support companies and individuals in the file '`etc/SERVICE`' in the GNU Emacs distribution.

Otherwise, send bug reports for `ld` to '`bug-binutils@gnu.org`'.

The fundamental principle of reporting bugs usefully is this: **report all the facts**. If you are not sure whether to state a fact or leave it out, state it!

Often people omit facts because they think they know what causes the problem and assume that some details do not matter. Thus, you might assume that the name of a symbol you use in an example does not matter. Well, probably it does not, but one cannot be sure. Perhaps the bug is a stray memory reference which happens to fetch from the location where that name is stored in memory; perhaps, if the name were different, the contents of that location would fool the linker into doing the right thing despite the bug. Play it safe and give a specific, complete example. That is the easiest thing for you to do, and the most helpful.

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

Sometimes people give a few sketchy facts and ask, "Does this ring a bell?" This cannot help us fix a bug, so it is basically useless. We respond by asking for enough details to enable us to investigate. You might as well expedite matters by sending them to begin with.

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

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

  Without this, we will not know whether there is any point in looking for the bug in the current version of `ld`.
- Any patches you may have applied to the `ld` source, including any patches made to the `BFD` library.
- The type of machine you are using, and the operating system name and version number.
- What compiler (and its version) was used to compile `ld`—e.g. "`gcc-2.7`".
- The command arguments you gave the linker to link your example and observe the bug. To guarantee you will not omit something important, list them all. A copy of the Makefile (or the output from make) is sufficient.

  If we were to try to guess the arguments, we would probably guess wrong and then we might not encounter the bug.
- A complete input file, or set of input files, that will reproduce the bug. It is generally most helpful to send the actual object files provided that they are reasonably small. Say no more than 10K. For bigger files you can either make them available by FTP or HTTP or else state that you are willing to send the object file(s) to whomever requests them. (Note - your email will be going to a mailing list, so we do not want to clog it up with large attachments). But small attachments are best.

  If the source files were assembled using `gas` or compiled using `gcc`, then it may be OK to send the source files rather than the object files. In this case, be sure to say exactly what version of `gas` or `gcc` was used to produce the object files. Also say how `gas` or `gcc` were configured.
- A description of what behavior you observe that you believe is incorrect. For example, "It gets a fatal signal."

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

  Even if the problem you experience is a fatal signal, you should still say so explicitly. Suppose something strange is going on, such as, your copy of `ld` is out of synch, or you have encountered a bug in the C library on your system. (This has happened!) Your copy might crash and ours would not. If you told us to expect a crash, then when ours fails to crash, we would know that the bug was not happening for us. If you had not told us to expect a crash, then we would not be able to draw any conclusion from our observations.
- If you wish to suggest changes to the `ld` source, send us context diffs, as generated by `diff` with the '`-u`', '`-c`', or '`-p`' option. Always send diffs from the old file to the new file. If you even discuss something in the `ld` source, refer to it by context, not by line number.

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

Here are some things that are not necessary:

- A description of the envelope of the bug.

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

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

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

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

- A patch for the bug.

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

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

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

- A guess about what the bug is or what it depends on.

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

Appendix A MRI Compatible Script Files

To aid users making the transition to GNU `ld` from the MRI linker, `ld` can use MRI compatible linker scripts as an alternative to the more general-purpose linker scripting language described in Chapter 3 [Scripts], page 31. MRI compatible linker scripts have a much simpler command set than the scripting language otherwise used with `ld`. GNU `ld` supports the most commonly used MRI linker commands; these commands are described here.

In general, MRI scripts aren't of much use with the `a.out` object file format, since it only has three sections and MRI scripts lack some features to make use of them.

You can specify a file containing an MRI-compatible script using the '`-c`' command-line option.

Each command in an MRI-compatible script occupies its own line; each command line starts with the keyword that identifies the command (though blank lines are also allowed for punctuation). If a line of an MRI-compatible script begins with an unrecognized keyword, `ld` issues a warning message, but continues processing the script.

Lines beginning with '`*`' are comments.

You can write these commands using all upper-case letters, or all lower case; for example, '`chip`' is the same as '`CHIP`'. The following list shows only the upper-case form of each command.

`ABSOLUTE` *secname*
`ABSOLUTE` *secname*`,` *secname*`, ...` *secname*
: Normally, `ld` includes in the output file all sections from all the input files. However, in an MRI-compatible script, you can use the `ABSOLUTE` command to restrict the sections that will be present in your output program. If the `ABSOLUTE` command is used at all in a script, then only the sections named explicitly in `ABSOLUTE` commands will appear in the linker output. You can still use other input sections (whatever you select on the command line, or using `LOAD`) to resolve addresses in the output file.

`ALIAS` *out-secname*`,` *in-secname*
: Use this command to place the data from input section *in-secname* in a section called *out-secname* in the linker output file.

  in-secname may be an integer.

`ALIAS` *secname* `=` *expression*
: Align the section called *secname* to *expression*. The *expression* should be a power of two.

`BASE` *expression*
: Use the value of *expression* as the lowest address (other than absolute addresses) in the output file.

`CHIP` *expression*
`CHIP` *expression*`,` *expression*
: This command does nothing; it is accepted only for compatibility.

`END`
: This command does nothing whatever; it's only accepted for compatibility.

`FORMAT` *output-format*

Similar to the `OUTPUT_FORMAT` command in the more general linker language, but restricted to one of these output formats:

1. S-records, if *output-format* is '`S`'
2. IEEE, if *output-format* is '`IEEE`'
3. COFF (the '`coff-m68k`' variant in BFD), if *output-format* is '`COFF`'

`LIST` *anything*...

Print (to the standard output file) a link map, as produced by the `ld` command-line option '`-M`'.

The keyword `LIST` may be followed by anything on the same line, with no change in its effect.

`LOAD` *filename*
`LOAD` *filename*, *filename*, ... *filename*

Include one or more object file *filename* in the link; this has the same effect as specifying *filename* directly on the `ld` command line.

`NAME` *output-name*

output-name is the name for the program produced by `ld`; the MRI-compatible command `NAME` is equivalent to the command-line option '`-o`' or the general script language command `OUTPUT`.

`ORDER` *secname*, *secname*, ... *secname*
`ORDER` *secname secname secname*

Normally, `ld` orders the sections in its output file in the order in which they first appear in the input files. In an MRI-compatible script, you can override this ordering with the `ORDER` command. The sections you list with `ORDER` will appear first in your output file, in the order specified.

`PUBLIC` *name*=*expression*
`PUBLIC` *name*,*expression*
`PUBLIC` *name expression*

Supply a value (*expression*) for external symbol *name* used in the linker input files.

`SECT` *secname*, *expression*
`SECT` *secname*=*expression*
`SECT` *secname expression*

You can use any of these three forms of the `SECT` command to specify the start address (*expression*) for section *secname*. If you have more than one `SECT` statement for the same *secname*, only the *first* sets the start address.

Appendix B GNU Free Documentation License

Version 1.1, March 2000

Copyright (C) 2000, Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you."

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque."

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long

as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the

Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section entitled "Endorsements." Such a section may not be included in the Modified Version.
N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents,

unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications." You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement

between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.1
or any later version published by the Free Software Foundation;
with the Invariant Sections being list their titles, with the
Front-Cover Texts being list, and with the Back-Cover Texts being list.
A copy of the license is included in the section entitled "GNU
Free Documentation License."
```

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being *list*"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Index

"

" 55

-

-(..... 11
--accept-unknown-input-arch 12
--add-needed 12
--add-stdcall-alias 23
--allow-multiple-definition 15
--allow-shlib-undefined 15
--architecture=*arch* 4
--as-needed 12
--auxiliary 6
--base-file 23
--be8 62
--check-sections 13
--cref 13
--default-imported-symver 15
--default-symver 15
--defsym *symbol*=*exp* 13
--demangle[=*style*] 14
--disable-auto-image-base 26
--disable-auto-import 28
--disable-new-dtags 22
--disable-runtime-pseudo-reloc 28
--disable-stdcall-fixup 23
--discard-all 10
--discard-locals 10
--dll 23
--dll-search-prefix 26
--dynamic-linker *file* 14
--emit-relocs 8
--enable-auto-image-base 26
--enable-auto-import 26
--enable-extra-pe-debug 28
--enable-new-dtags 22
--enable-runtime-pseudo-reloc 28
--enable-stdcall-fixup 23
--entry=*entry* 5
--error-unresolved-symbols 22
--exclude-libs 5
--exclude-symbols 24
--export-all-symbols 24
--export-dynamic 5
--fatal-warnings 14
--file-alignment 24
--filter 6
--force-exe-suffix 14
--format=*format* 4
--format=*version* 64
--gc-sections 14
--gpsize 6
--hash-size=*number* 23
--heap 24
--help 14
--image-base 25
--just-symbols=*file* 9
--kill-at 25
--large-address-aware 25
--library-path=*dir* 7
--library=*archive* 7
--major-image-version 25
--major-os-version 25
--major-subsystem-version 25
--minor-image-version 25
--minor-os-version 25
--minor-subsystem-version 25
--mri-script=*MRI-cmdfile* 4
--multi-subspace 62
--nmagic 8
--no-accept-unknown-input-arch 12
--no-add-needed 12
--no-allow-shlib-undefined 15
--no-as-needed 12
--no-check-sections 13
--no-define-common 13
--no-demangle 14
--no-gc-sections 14
--no-keep-memory 15
--no-omagic 8
--no-relax 70
--no-undefined 15
--no-undefined-version 15
--no-warn-mismatch 15
--no-whole-archive 16
--noinhibit-exec 16
--oformat 16
--omagic 8
--out-implib 26
--output-def 25
--output=*output* 8
--pic-executable 16
--print-map 8
--reduce-memory-overheads 23
--relax 16
'--relax' on i960 62
'--relax' on Xtensa 69
--relocatable 9
--script=*script* 9
--section-alignment 28
--section-start *sectionname*=*org* 19
--sort-common 18
--sort-section alignment 18
--sort-section name 18
--split-by-file 18
--split-by-reloc 18
--stack 29
--stats 19

--strip-all 9
--strip-debug 9
--stub-group-size=*N* 62
--subsystem 29
--support-old-code 62
--target-help 14
--target1-abs 62
--target1-rel 62
--target2=*type* 62
--thumb-entry=*entry* 62
--trace 9
--trace-symbol=*symbol* 10
--traditional-format 19
--undefined=*symbol* 9
--unique[=*SECTION*] 10
--unresolved-symbols 19
--verbose 20
--version 10
--version-script=*version-scriptfile* 20
--warn-common 20
--warn-constructors 21
--warn-multiple-gp 21
--warn-once 21
--warn-section-align 21
--warn-shared-textrel 22
--warn-unresolved-symbols 22
--whole-archive 22
--wrap 22
-A*arch* 4
-a*keyword* 4
-assert *keyword* 12
-b *format* 4
-Bdynamic 12
-Bgroup 12
-Bshareable 18
-Bstatic 13
-Bsymbolic 13
-c *MRI-cmdfile* 4
-call_shared 12
-d 5
-dc 5
-dn 13
-dp 5
-dy 12
-E 5
-e *entry* 5
-EB 5
-EL 6
-f 6
-F 6
-fini 6
-g 6
-G 6
-h*name* 7
-i 7
-I*file* 14
-init 7
-l*archive* 7
-L*dir* 7
-M 8
-m *emulation* 8
-Map 14
-n 8
-N 8
-non_shared 13
-nostdlib 16
-O *level* 8
-o *output* 8
-pie 16
-q 8
-qmagic 16
-Qy 16
-r 9
-R *file* 9
-rpath 17
-rpath-link 17
-s 9
-S 9
-shared 18
-soname=*name* 7
-static 13
-t 9
-T *script* 9
-Tbss *org* 19
-Tdata *org* 19
-Ttext *org* 19
-u *symbol* 9
-Ur 10
-v 10
-V 10
-x 10
-X 10
-Y *path* 10
-y *symbol* 10
-z defs 15
-z *keyword* 10
-z muldefs 15

.

. 55

/

/DISCARD/ 45

:

:*phdr* 46

=

=*fillexp* 47

>

`>region` ... 46

[

[COMMON] ... 42

A

`ABSOLUTE` (MRI) ... 79
absolute and relocatable symbols ... 57
absolute expressions ... 57
`ABSOLUTE(exp)` ... 57
`ADDR(section)` ... 57
address, section ... 39
`ALIAS` (MRI) ... 79
`ALIGN` (MRI) ... 79
align expression ... 58
align location counter ... 58
`ALIGN(align)` ... 58
`ALIGN(exp,align)` ... 58
allocating memory ... 48
architecture ... 36
architectures ... 4
archive files, from cmd line ... 7
archive search path in linker script ... 34
arithmetic ... 54
arithmetic operators ... 56
ARM interworking support ... 62
`ASSERT` ... 35
assertion in linker script ... 35
assignment in scripts ... 36
`AT(lma)` ... 45
`AT>lma_region` ... 45
automatic data imports ... 65

B

back end ... 71
`BASE` (MRI) ... 79
BE8 ... 62
BFD canonical format ... 72
BFD requirements ... 71
big-endian objects ... 5
binary input format ... 4
`BLOCK(exp)` ... 58
bug criteria ... 75
bug reports ... 75
bugs in `ld` ... 75
`BYTE(expression)` ... 42

C

C++ constructors, arranging in link ... 43
`CHIP` (MRI) ... 79
`COLLECT_NO_DEMANGLE` ... 29
combining symbols, warnings on ... 20
command files ... 31
command line ... 3
common allocation ... 5, 13
common allocation in linker script ... 35
common symbol placement ... 41
compatibility, MRI ... 4
constants in linker scripts ... 54
constructors ... 10
`CONSTRUCTORS` ... 43
constructors, arranging in link ... 43
crash of linker ... 75
`CREATE_OBJECT_SYMBOLS` ... 43
creating a DEF file ... 65
cross reference table ... 13
cross references ... 36
current output location ... 55

D

data ... 42
`DATA_SEGMENT_ALIGN(maxpagesize, commonpagesize)` ... 58
`DATA_SEGMENT_END(exp)` ... 59
`DATA_SEGMENT_RELRO_END(offset, exp)` ... 59
dbx ... 19
DEF files, creating ... 25
default emulation ... 29
default input format ... 29
`DEFINED(symbol)` ... 59
deleting local symbols ... 10
demangling, default ... 29
demangling, from command line ... 14
direct linking to a dll ... 66
discarding sections ... 44
discontinuous memory ... 48
DLLs, creating ... 24, 25, 26
DLLs, linking to ... 26
dot ... 55
dot inside sections ... 55
dynamic linker, from command line ... 14
dynamic symbol table ... 5

E

ELF program headers ... 50
emulation ... 8
emulation, default ... 29
`END` (MRI) ... 79
endianness ... 5
entry point ... 33
entry point, from command line ... 5
entry point, thumb ... 62
`ENTRY(symbol)` ... 33

error on valid input 75
example of linker script 32
exporting DLL symbols 64
expression evaluation order 56
expression sections 57
expression, absolute 57
expressions 54
EXTERN 35

F

fatal signal 75
file name wildcard patterns 40
FILEHDR 51
filename symbols 43
fill pattern, entire section 47
FILL(*expression*) 43
finalization function 6
first input file 34
first instruction 33
FORCE_COMMON_ALLOCATION 35
forcing input section alignment 46
FORMAT (MRI) 79
functions in expressions 57

G

garbage collection 14, 42
generating optimized output 8
GNU linker 1
GNUTARGET 29
GROUP(*files*) 34
grouping input files 34
groups of archives 11

H

H8/300 support 61
header size 60
heap size 24
help 14
holes 55
holes, filling 43
HPPA multiple sub-space stubs 62
HPPA stub grouping 62

I

i960 support 61
image base 25
implicit linker scripts 60
import libraries 64
INCLUDE *filename* 33
including a linker script 33
including an entire archive 22
incremental link 7
INHIBIT_COMMON_ALLOCATION 35
initialization function 7
initialized data in ROM 46
input file format in linker script 35
input filename symbols 43
input files in linker scripts 33
input files, displaying 9
input format 4
input object files in linker scripts 33
input section alignment 46
input section basics 39
input section wildcards 40
input sections 39
INPUT(*files*) 33
integer notation 54
integer suffixes 54
internal object-file format 72
invalid input 75

K

K and M integer suffixes 54
KEEP 42

L

l = ... 49
L, deleting symbols beginning 10
lazy evaluation 56
ld bugs, reporting 75
LDEMULATION 29
len = ... 49
LENGTH = ... 49
LENGTH(*memory*) 59
library search path in linker script 34
link map 8
link-time runtime library search path 17
linker crash 75
linker script concepts 31
linker script example 32
linker script file commands 33
linker script format 32
linker script input object files 33
linker script simple commands 33
linker scripts 31
LIST (MRI) 80
little-endian objects 6
LOAD (MRI) 80
load address 45
LOADADDR(*section*) 59
loading, preventing 45
local symbols, deleting 10
location counter 55
LONG(*expression*) 42

M

M and K integer suffixes 54
machine architecture 36
machine dependencies 61
mapping input sections to output sections 39
`MAX` 59
`MEMORY` 48
memory region attributes 49
memory regions 48
memory regions and sections 46
memory usage 15
`MIN` 59
MRI compatibility 79
MSP430 extra sections 63

N

`NAME` (MRI) 80
name, section 38
names 55
naming the output file 8
`NEXT(`*`exp`*`)` 59
NMAGIC 8
`NOCROSSREFS(`*`sections`*`)` 36
`NOLOAD` 45
not enough room for program headers 60

O

`o =` 49
`objdump -i` 71
object file management 71
object files 3
object formats available 71
object size 6
OMAGIC 8
opening object files 71
operators for arithmetic 56
options 3
`ORDER` (MRI) 80
`org =` 49
`ORIGIN =` 49
`ORIGIN(`*`memory`*`)` 60
output file after errors 16
output file format in linker script 35
output file name in linker scripot 34
output section attributes 45
output section data 42
`OUTPUT(`*`filename`*`)` 34
`OUTPUT_ARCH(`*`bfdarch`*`)` 36
`OUTPUT_FORMAT(`*`bfdname`*`)` 35
`OVERLAY` 47
overlays 47

P

partial link 9
`PHDRS` 50, 51
position independent executables 16
precedence in expressions 56
prevent unnecessary loading 45
program headers 50
program headers and sections 46
program headers, not enough room 60
program segments 50
PROVIDE 37
`PUBLIC` (MRI) 80

Q

`QUAD(`*`expression`*`)` 42
quoted symbol names 55

R

read-only text 8
read/write from cmd line 8
regions of memory 48
relative expressions 57
relaxing addressing modes 16
relaxing on H8/300 61
relaxing on i960 62
relaxing on Xtensa 69
relocatable and absolute symbols 57
relocatable output 9
removing sections 44
reporting bugs in `ld` 75
requirements for BFD 71
retain relocations in final executable 8
retaining specified symbols 17
ROM initialized data 46
round up expression 58
round up location counter 58
runtime library name 7
runtime library search path 17
runtime pseudo-relocation 66

S

scaled integers 54
scommon section 42
script files 9
scripts 31
search directory, from cmd line 7
search path in linker script 34
`SEARCH_DIR(`*`path`*`)` 34
`SECT` (MRI) 80
section address 39
section address in expression 57
section alignment, warnings on 21
section data 42
section fill pattern 47

section load address ... 45
section load address in expression ... 59
section name ... 38
section name wildcard patterns ... 40
section size ... 60
section, assigning to memory region ... 46
section, assigning to program header ... 46
SECTIONS ... 37
sections, discarding ... 44
segment origins, cmd line ... 19
SEGMENT_START(*segment*, *default*) ... 60
segments, ELF ... 50
shared libraries ... 18
SHORT(*expression*) ... 42
SIZEOF(*section*) ... 60
SIZEOF_HEADERS ... 60
small common symbols ... 42
SORT ... 41
SORT_BY_ALIGNMENT ... 41
SORT_BY_NAME ... 40
SQUAD(*expression*) ... 42
stack size ... 29
standard Unix system ... 3
start of execution ... 33
STARTUP(*filename*) ... 34
strip all symbols ... 9
strip debugger symbols ... 9
stripping all but some symbols ... 17
SUBALIGN(*subsection_align*) ... 46
suffixes for integers ... 54
symbol defaults ... 59
symbol definition, scripts ... 36
symbol names ... 55
symbol tracing ... 10
symbol versions ... 52
symbol-only input ... 9
symbols, from command line ... 13
symbols, relocatable and absolute ... 57
symbols, retaining selectively ... 17
synthesizing linker ... 16
synthesizing on H8/300 ... 61

T

TARGET(*bfdname*) ... 35
TARGET1 ... 62
TARGET2 ... 62
thumb entry point ... 62
TI COFF versions ... 64
traditional format ... 19

U

unallocated address, next ... 59
undefined symbol ... 9
undefined symbol in linker script ... 35
undefined symbols, warnings on ... 21
uninitialized data placement ... 41
unspecified memory ... 43
usage ... 14
using a DEF file ... 64
using auto-export functionality ... 64
Using decorations ... 65

V

variables, defining ... 36
verbose ... 20
version ... 10
VERSION {script text} ... 52
version script ... 52
version script, symbol versions ... 20
versions of symbols ... 52

W

warnings, on combining symbols ... 20
warnings, on section alignment ... 21
warnings, on undefined symbols ... 21
weak externals ... 69
what is this? ... 1
wildcard file name patterns ... 40

X

Xtensa options ... 70
Xtensa processors ... 69

The body of this manual is set in
cmr10 at 10.95pt,
with headings in **cmb10 at 10.95pt**
and examples in `cmtt10 at 10.95pt`.
cmti10 at 10.95pt and
cmsl10 at 10.95pt
are used for emphasis.

Table of Contents

1 Overview 1

2 Invocation 3

2.1 Command Line Options 3
2.1.1 Options Specific to i386 PE Targets 23
2.2 Environment Variables 29

3 Linker Scripts 31

3.1 Basic Linker Script Concepts 31
3.2 Linker Script Format 32
3.3 Simple Linker Script Example 32
3.4 Simple Linker Script Commands 33
3.4.1 Setting the Entry Point 33
3.4.2 Commands Dealing with Files 33
3.4.3 Commands Dealing with Object File Formats 34
3.4.4 Other Linker Script Commands 35
3.5 Assigning Values to Symbols 36
3.5.1 Simple Assignments 36
3.5.2 PROVIDE 37
3.6 SECTIONS Command 37
3.6.1 Output Section Description 38
3.6.2 Output Section Name 38
3.6.3 Output Section Address 39
3.6.4 Input Section Description 39
3.6.4.1 Input Section Basics 39
3.6.4.2 Input Section Wildcard Patterns 40
3.6.4.3 Input Section for Common Symbols 41
3.6.4.4 Input Section and Garbage Collection 42
3.6.4.5 Input Section Example 42
3.6.5 Output Section Data 42
3.6.6 Output Section Keywords 43
3.6.7 Output Section Discarding 44
3.6.8 Output Section Attributes 45
3.6.8.1 Output Section Type 45
3.6.8.2 Output Section LMA 45
3.6.8.3 Forced Input Alignment 46
3.6.8.4 Output Section Region 46
3.6.8.5 Output Section Phdr 46
3.6.8.6 Output Section Fill 47
3.6.9 Overlay Description 47
3.7 MEMORY Command 48
3.8 PHDRS Command 50

3.9 VERSION Command ... 52
3.10 Expressions in Linker Scripts ... 54
3.10.1 Constants ... 54
3.10.2 Symbol Names ... 55
3.10.3 The Location Counter ... 55
3.10.4 Operators ... 56
3.10.5 Evaluation ... 56
3.10.6 The Section of an Expression ... 57
3.10.7 Builtin Functions ... 57
3.11 Implicit Linker Scripts ... 60

4 Machine Dependent Features ... 61
4.1 `ld` and the H8/300 ... 61
4.2 `ld` and the Intel 960 Family ... 61
4.3 `ld` and the ARM family ... 62
4.4 `ld` and HPPA 32-bit ELF Support ... 62
4.5 `ld` and MMIX ... 63
4.6 `ld` and MSP430 ... 63
4.7 `ld`'s Support for Various TI COFF Versions ... 64
4.8 `ld` and WIN32 (cygwin/mingw) ... 64
4.9 `ld` and Xtensa Processors ... 69

5 BFD ... 71
5.1 How It Works: An Outline of BFD ... 71
5.1.1 Information Loss ... 71
5.1.2 The BFD canonical object-file format ... 72

6 Reporting Bugs ... 75
6.1 Have You Found a Bug? ... 75
6.2 How to Report Bugs ... 75

Appendix A MRI Compatible Script Files ... 79

Appendix B GNU Free Documentation License ... 81
ADDENDUM: How to use this License for your documents ... 86

Index ... 87