This is ld.info, produced by makeinfo version 6.3 from ld.texinfo.

This file documents the GNU linker LD (GNU Binutils) version 2.28.

Copyright (C) 1991-2017 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 1.3 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".

INFO-DIR-SECTION Software development
START-INFO-DIR-ENTRY
* Ld: (ld). The GNU linker.
END-INFO-DIR-ENTRY

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

LD
**

This file documents the GNU linker ld (GNU Binutils) version 2.28.

This document is distributed under the terms of the GNU Free Documentation License version 1.3. A copy of the license is included in the section entitled "GNU Free Documentation License".

* Menu:

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

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. *Note BFD::, 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).

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

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.

* Menu:

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

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. Specifying a script in this way merely augments the main linker script, with the extra commands placed after the main script; use the '-T' option to replace the default linker script entirely, but note the effect of the 'INSERT' command. *Note Scripts::.

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 preceded 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,--start-group foo.o bar.o -Wl,--end-group
```

This is important, because otherwise the compiler driver program may silently drop the linker options, resulting in a bad link. Confusion may also arise when passing options that require values through a driver, as the use of a space between option and argument acts as a separator, and causes the driver to pass only the option to the linker and the argument to the compiler. In this case, it is simplest to use the joined forms of both single- and multiple-letter options, such as:

```
gcc foo.o bar.o -Wl,-eENTRY -Wl,-Map=a.map
```

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

'@FILE'
Read command-line options from FILE. The options read are inserted in place of the original @FILE option. If FILE does not exist, or

cannot be read, then the option will be treated literally, and not removed.

Options in FILE are separated by whitespace. A whitespace character may be included in an option by surrounding the entire option in either single or double quotes. Any character (including a backslash) may be included by prefixing the character to be included with a backslash. The FILE may itself contain additional @FILE options; any such options will be processed recursively.

'-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.

'--audit AUDITLIB'
Adds AUDITLIB to the 'DT_AUDIT' entry of the dynamic section. AUDITLIB is not checked for existence, nor will it use the DT_SONAME specified in the library. If specified multiple times 'DT_AUDIT' will contain a colon separated list of audit interfaces to use. If the linker finds an object with an audit entry while searching for shared libraries, it will add a corresponding 'DT_DEPAUDIT' entry in the output file. This option is only meaningful on ELF platforms supporting the rtld-audit interface.

'-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. *Note 'ld' and the Intel 960 family: i960, 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'.) *Note BFD::.

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'. *Note Environment::. You can also define the input format from a script, using the command 'TARGET'; see *note Format Commands::.

'-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 *note MRI Compatible Script Files: MRI. 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. *Note Miscellaneous Commands::.

'--depaudit AUDITLIB'
'-P AUDITLIB'
Adds AUDITLIB to the 'DT_DEPAUDIT' entry of the dynamic section. AUDITLIB is not checked for existence, nor will it use the DT_SONAME specified in the library. If specified multiple times 'DT_DEPAUDIT' will contain a colon separated list of audit interfaces to use. This option is only meaningful on ELF platforms supporting the rtld-audit interface. The -P option is provided for Solaris compatibility.

'-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). *Note Entry Point::, 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.

'--exclude-modules-for-implib MODULE,MODULE,...'
Specifies a list of object files or archive members, from which symbols should not be automatically exported, but which should be copied wholesale into the import library being generated during the link. The module names may be delimited by commas or colons, and

must match exactly the filenames used by 'ld' to open the files; for archive members, this is simply the member name, but for object files the name listed must include and match precisely any path used to specify the input file on the linker's command-line. This option is available only for the i386 PE targeted port of the linker. Symbols explicitly listed in a .def file are still exported, regardless of this option.

'-E'
'--export-dynamic'
'--no-export-dynamic'
When creating a dynamically linked executable, using the '-E' option or the '--export-dynamic' option causes the linker to 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 either of these options (or use the '--no-export-dynamic' option to restore the default behavior), 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 dynamic list to control what symbols should be added to the dynamic symbol table if the output format supports it. See the description of '--dynamic-list'.

Note that this option is specific to ELF targeted ports. PE targets support a similar function to export all symbols from a DLL or EXE; see the description of '--export-all-symbols' below.

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

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

'-f NAME'
'--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 ELF that support 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 NAMESPEC'
'--library=NAMESPEC'
Add the archive or object file specified by NAMESPEC to the list of files to link. This option may be used any number of times. If NAMESPEC is of the form ':FILENAME', 'ld' will search the library path for a file called FILENAME, otherwise it will search the library path for a file called 'libNAMESPEC.a'.

On systems which support shared libraries, 'ld' may also search for files other than 'libNAMESPEC.a'. Specifically, on ELF and SunOS systems, 'ld' will search a directory for a library called 'libNAMESPEC.so' before searching for one called 'libNAMESPEC.a'. (By convention, a '.so' extension indicates a shared library.) Note that this behavior does not apply to ':FILENAME', which always specifies a file called FILENAME.

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. '-L' options do not affect how 'ld' searches for a linker script unless '-T' option is specified.

If SEARCHDIR begins with '=', then the '=' will be replaced by the "sysroot prefix", controlled by the '--sysroot' option, or 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. *Note Environment::.

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 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.
* The values assigned to symbols.

Note - symbols whose values are computed by an expression which involves a reference to a previous value of the same symbol may not have correct result displayed in the link map. This is because the linker discards intermediate results and only retains the final value of an expression. Under such circumstances the linker will display the final value enclosed by square brackets. Thus for example a linker script containing:

```
foo = 1
foo = foo * 4
foo = foo + 8
```

will produce the following output in the link map if the '-M' option is used:

```
0x00000001                foo = 0x1
[0x0000000c]                foo = (foo * 0x4)
[0x0000000c]                foo = (foo + 0x8)
```

See *note Expressions:: for more information about expressions in linker scripts.

'-n'
'--nmagic'
Turn off page alignment of sections, and disable linking against shared libraries. If the output format supports Unix style magic numbers, mark the output as 'NMAGIC'.

'-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. At the
     moment this option only affects ELF shared library generation.
     Future releases of the linker may make more use of this option.
     Also currently there is no difference in the linker's behaviour for
     different non-zero values of this option. Again this may change
     with future releases.

'--push-state'
     The '--push-state' allows to preserve the current state of the
     flags which govern the input file handling so that they can all be
     restored with one corresponding '--pop-state' option.

     The option which are covered are: '-Bdynamic', '-Bstatic', '-dn',
     '-dy', '-call_shared', '-non_shared', '-static', '-N', '-n',
     '--whole-archive', '--no-whole-archive', '-r', '-Ur',
     '--copy-dt-needed-entries', '--no-copy-dt-needed-entries',
     '--as-needed', '--no-as-needed', and '-a'.

     One target for this option are specifications for 'pkg-config'.
     When used with the '--libs' option all possibly needed libraries
     are listed and then possibly linked with all the time. It is
     better to return something as follows:

          -Wl,--push-state,--as-needed -libone -libtwo -Wl,--pop-state

     Undoes the effect of -push-state, restores the previous values of
     the flags governing input file handling.

'-q'
'--emit-relocs'
     Leave relocation sections and contents in fully linked executables.
     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.

'--force-dynamic'
     Force the output file to have dynamic sections. This option is
     specific to VxWorks targets.

'-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. *Note Scripts::. 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.

'-dT SCRIPTFILE'
'--default-script=SCRIPTFILE'
Use SCRIPTFILE as the default linker script. *Note Scripts::.

This option is similar to the '--script' option except that processing of the script is delayed until after the rest of the command line has been processed. This allows options placed after the '--default-script' option on the command line to affect the behaviour of the linker script, which can be important when the linker command line cannot be directly controlled by the user. (eg

because the command line is being constructed by another tool, such as 'gcc').

'-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.

If this option is being used to force additional modules to be pulled into the link, and if it is an error for the symbol to remain undefined, then the option '--require-defined' should be used instead.

'--require-defined=SYMBOL'
Require that SYMBOL is defined in the output file. This option is the same as option '--undefined' except that if SYMBOL is not defined in the output file then the linker will issue an error and exit. The same effect can be achieved in a linker script by using 'EXTERN', 'ASSERT' and 'DEFINED' together. This option can be used multiple times to require additional symbols.

'-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.

'--orphan-handling=MODE'
Control how orphan sections are handled. An orphan section is one not specifically mentioned in a linker script. *Note Orphan Sections::.

MODE can have any of the following values:

'place'
Orphan sections are placed into a suitable output section following the strategy described in *note Orphan Sections::. The option '--unique' also effects how sections are placed.

'discard'
All orphan sections are discarded, by placing them in the '/DISCARD/' section (*note Output Section Discarding::).

'warn'
The linker will place the orphan section as for 'place' and also issue a warning.

'error'
The linker will exit with an error if any orphan section is found.

The default if '--orphan-handling' is not given is 'place'.

'--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. (These symbols start with
     system-specific local label prefixes, typically '.L' for ELF
     systems or 'L' for traditional a.out systems.)

'-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.

     'common'
          Generate common symbols with the STT_COMMON type druing a
          relocatable link.

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

     'execstack'
          Marks the object as requiring executable stack.

     'global'
          This option is only meaningful when building a shared object.

It makes the symbols defined by this shared object available for symbol resolution of subsequently loaded libraries.

'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.

'lazy'
When generating an executable or shared library, mark it to tell the dynamic linker to defer function call resolution to the point when the function is called (lazy binding), rather than at load time. Lazy binding is the default.

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

'muldefs'
Allows multiple definitions.

'nocombreloc'
Disables multiple reloc sections combining.

'nocommon'
Generate common symbols with the STT_OBJECT type druing a relocatable link.

'nocopyreloc'
Disable linker generated .dynbss variables used in place of variables defined in shared libraries. May result in dynamic text relocations.

'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'.

'noexecstack'
Marks the object as not requiring executable stack.

'text'
Treat DT_TEXTREL in shared object as error.

'notext'
Don't treat DT_TEXTREL in shared object as error.

'textoff'
Don't treat DT_TEXTREL in shared object as error.

'norelro'
Don't create an ELF 'PT_GNU_RELRO' segment header in the object.

'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.

'relro'
Create an ELF 'PT_GNU_RELRO' segment header in the object.

'max-page-size=VALUE'
Set the emulation maximum page size to VALUE.

'common-page-size=VALUE'
Set the emulation common page size to VALUE.

'stack-size=VALUE'
Specify a stack size for in an ELF 'PT_GNU_STACK' segment. Specifying zero will override any default non-zero sized 'PT_GNU_STACK' segment creation.

'bndplt'
Always generate BND prefix in PLT entries. Supported for Linux/x86_64.

'noextern-protected-data'
Don't treat protected data symbol as external when building shared library. This option overrides linker backend default. It can be used to workaround incorrect relocations against protected data symbols generated by compiler. Updates on protected data symbols by another module aren't visible to the resulting shared library. Supported for i386 and x86-64.

'nodynamic-undefined-weak'
Don't treat undefined weak symbols as dynamic when building executable. This option overrides linker backend default. It can be used to avoid dynamic relocations against undefined weak symbols in executable. Supported for i386 and x86-64.

'noreloc-overflow'
Disable relocation overflow check. This can be used to disable relocation overflow check if there will be no dynamic relocation overflow at run-time. Supported for x86_64.

'call-nop=prefix-addr'
'call-nop=prefix-nop'

```
     'call-nop=suffix-nop'
     'call-nop=prefix-BYTE'
     'call-nop=suffix-BYTE'
          Specify the 1-byte 'NOP' padding when transforming indirect
          call to a locally defined function, foo, via its GOT slot.
          'call-nop=prefix-addr' generates '0x67 call foo'.
          'call-nop=prefix-nop' generates '0x90 call foo'.
          'call-nop=suffix-nop' generates 'call foo 0x90'.
          'call-nop=prefix-BYTE' generates 'BYTE call foo'.
          'call-nop=suffix-BYTE' generates 'call foo BYTE'.  Supported
          for i386 and x86_64.

     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 or not.  '--as-needed' causes a
     DT_NEEDED tag to only be emitted for a library that _at that point
     in the link_ satisfies a non-weak undefined symbol reference from a
     regular object file or, if the library is not found in the
     DT_NEEDED lists of other needed libraries, a non-weak undefined
     symbol reference from another needed dynamic library.  Object files
     or libraries appearing on the command line _after_ the library in
     question do not affect whether the library is seen as needed.  This
     is similar to the rules for extraction of object files from
     archives.  '--no-as-needed' restores the default behaviour.
```

'--add-needed'
'--no-add-needed'
These two options have been deprecated because of the similarity of their names to the '--as-needed' and '--no-as-needed' options. They have been replaced by '--copy-dt-needed-entries' and '--no-copy-dt-needed-entries'.

'-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'. This option can be used with '-shared'. Doing so means that a shared library is being created but that all of the library's external references must be resolved by pulling in entries from static libraries.

'-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 can also be used with the '--export-dynamic' option, when creating a position independent executable, to bind references to global symbols to the definition within the executable. This option is only meaningful on ELF platforms which support shared libraries and position independent executables.

'-Bsymbolic-functions'
When creating a shared library, bind references to global function symbols to the definition within the shared library, if any. This option can also be used with the '--export-dynamic' option, when creating a position independent executable, to bind references to global function symbols to the definition within the executable. This option is only meaningful on ELF platforms which support

shared libraries and position independent executables.

'--dynamic-list=DYNAMIC-LIST-FILE'
Specify the name of a dynamic list file to the linker. This is typically used when creating shared libraries to specify a list of global symbols whose references shouldn't be bound to the definition within the shared library, or creating dynamically linked executables to specify a list of symbols which should be added to the symbol table in the executable. This option is only meaningful on ELF platforms which support shared libraries.

The format of the dynamic list is the same as the version node without scope and node name. See *note VERSION:: for more information.

'--dynamic-list-data'
Include all global data symbols to the dynamic list.

'--dynamic-list-cpp-new'
Provide the builtin dynamic list for C++ operator new and delete. It is mainly useful for building shared libstdc++.

'--dynamic-list-cpp-typeinfo'
Provide the builtin dynamic list for C++ runtime type identification.

'--check-sections'
'--no-check-sections'
Asks the linker _not_ to check section addresses after they have been assigned to see if there are 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'. Section overlap is not usually checked for relocatable links. You can force checking in that case by using the '--check-sections' option.

'--copy-dt-needed-entries'
'--no-copy-dt-needed-entries'
This option affects the treatment of dynamic libraries referred to by DT_NEEDED tags _inside_ ELF dynamic libraries mentioned on the command line. Normally the linker won't add a DT_NEEDED tag to the output binary for each library mentioned in a DT_NEEDED tag in an input dynamic library. With '--copy-dt-needed-entries' specified on the command line however any dynamic libraries that follow it will have their DT_NEEDED entries added. The default behaviour can be restored with '--no-copy-dt-needed-entries'.

This option also has an effect on the resolution of symbols in dynamic libraries. With '--copy-dt-needed-entries' dynamic libraries mentioned on the command line will be recursively searched, following their DT_NEEDED tags to other libraries, in order to resolve symbols required by the output binary. With the default setting however the searching of dynamic libraries that follow it will stop with the dynamic library itself. No DT_NEEDED links will be traversed to resolve symbols.

'--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. If the symbol is defined as a common value then any files where this happens appear next. Finally any files that reference the symbol are listed.

'--no-define-common'
This option inhibits the assignment of addresses to common symbols. The script command 'INHIBIT_COMMON_ALLOCATION' has the same effect. *Note Miscellaneous Commands::.

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 (*note Assignments::). _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.

'-IFILE'
'--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.

'--no-dynamic-linker'
     When producing an executable file, omit the request for a dynamic
     linker to be used at load-time. This is only meaningful for ELF
     executables that contain dynamic relocations, and usually requires
     entry point code that is capable of processing these relocations.

'--fatal-warnings'
'--no-fatal-warnings'
     Treat all warnings as errors. The default behaviour can be
     restored with the option '--no-fatal-warnings'.

'--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.

'--gc-sections'
'--no-gc-sections'
     Enable garbage collection of unused input sections. It is ignored
     on targets that do not support this option. The default behaviour
     (of not performing this garbage collection) can be restored by
     specifying '--no-gc-sections' on the command line. Note that
     garbage collection for COFF and PE format targets is supported, but
     the implementation is currently considered to be experimental.

     '--gc-sections' decides which input sections are used by examining
     symbols and relocations. The section containing the entry symbol
     and all sections containing symbols undefined on the command-line
     will be kept, as will sections containing symbols referenced by
     dynamic objects. Note that when building shared libraries, the
     linker must assume that any visible symbol is referenced. Once
     this initial set of sections has been determined, the linker
     recursively marks as used any section referenced by their
     relocations. See '--entry' and '--undefined'.

     This option can be set when doing a partial link (enabled with
     option '-r'). In this case the root of symbols kept must be
     explicitly specified either by an '--entry' or '--undefined' option
     or by a 'ENTRY' command in the linker script.

'--print-gc-sections'
'--no-print-gc-sections'
     List all sections removed by garbage collection. The listing is
     printed on stderr. This option is only effective if garbage
     collection has been enabled via the '--gc-sections') option. The
     default behaviour (of not listing the sections that are removed)
     can be restored by specifying '--no-print-gc-sections' on the
     command line.

'--gc-keep-exported'
     When '--gc-sections' is enabled, this option prevents garbage
     collection of unused input sections that contain global symbols
     having default or protected visibility. This option is intended to

be used for executables where unreferenced sections would otherwise be garbage collected regardless of the external visibility of contained symbols. Note that this option has no effect when linking shared objects since it is already the default behaviour. This option is only supported for ELF format targets.

'--print-output-format'
Print the name of the default output format (perhaps influenced by other command-line options). This is the string that would appear in an 'OUTPUT_FORMAT' linker script command (*note File Commands::).

'--print-memory-usage'
Print used size, total size and used size of memory regions created with the *note MEMORY:: command. This is useful on embedded targets to have a quick view of amount of free memory. The format of the output has one headline and one line per region. It is both human readable and easily parsable by tools. Here is an example of an output:

```
     Memory region         Used Size  Region Size  %age Used
                  ROM:        256 KB         1 MB     25.00%
                  RAM:          32 B         2 GB      0.00%
```

'--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 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 default behaviour is to report errors for any undefined symbols referenced in shared libraries if the linker is being used to create an executable, but to allow them if the linker is being used to create a shared library.

The reasons for allowing undefined symbol references in shared libraries specified at link time are that:

* A shared library specified at link time may not be the same as the one that is available at load time, so the symbol might actually be resolvable at load time.
* There are some operating systems, eg BeOS and HPPA, where undefined symbols in shared libraries are normal.

  The BeOS kernel for example patches shared libraries at load time to select whichever function is most appropriate for the current architecture. This is used, for example, to dynamically select an appropriate memset function.

'--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-warn-search-mismatch'
Normally 'ld' will give a warning if it finds an incompatible library during a library search. This option silences the warning.

'--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. *Note BFD::.

'--out-implib FILE'
Create an import library in FILE corresponding to the executable the linker is generating (eg. a DLL or ELF program). This import library (which should be called '*.dll.a' or '*.a' for DLLs) may be used to link clients against the generated executable; this behaviour makes it possible to skip a separate import library creation step (eg. 'dlltool' for DLLs). This option is only available for the i386 PE and ELF targetted ports of the linker.

'-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'
'--no-relax'
An option with machine dependent effects. This option is only supported on a few targets. *Note 'ld' and the H8/300: H8/300. *Note 'ld' and the Intel 960 family: i960. *Note 'ld' and Xtensa Processors: Xtensa. *Note 'ld' and the 68HC11 and 68HC12: M68HC11/68HC12. *Note 'ld' and the Altera Nios II: Nios II. *Note 'ld' and PowerPC 32-bit ELF Support: PowerPC ELF32.

On some platforms the '--relax' option performs target specific, global optimizations that become possible when the linker resolves addressing in the program, such as relaxing address modes, synthesizing new instructions, selecting shorter version of current instructions, and combining constant values.

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.

On platforms where '--relax' is accepted the option '--no-relax' can be used to disable the feature.

'--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 path 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 file systems.

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

'-rpath-link=DIR'
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.

The tokens $ORIGIN and $LIB can appear in these search directories. They will be replaced by the full path to the directory containing the program or shared object in the case of $ORIGIN and either 'lib' - for 32-bit binaries - or 'lib64' - for 64-bit binaries - in the case of $LIB.

The alternative form of these tokens - ${ORIGIN} and ${LIB} can also be used. The token $PLATFORM is not supported.

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. Searching '-rpath' in this way is only supported by native linkers and cross linkers which have been configured with the '--with-sysroot' option.
3. On an ELF system, for native linkers, if the '-rpath' and '-rpath-link' options were not used, search the contents of the environment variable 'LD_RUN_PATH'.
4. On SunOS, if the '-rpath' option was not used, search any directories specified using '-L' options.
5. For a native linker, search 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'
'--sort-common=ascending'
'--sort-common=descending'
This option tells 'ld' to sort the common symbols by alignment in ascending or descending order when it places them in the appropriate output sections. The symbol alignments considered are

sixteen-byte or larger, eight-byte, four-byte, two-byte, and one-byte. This is to prevent gaps between symbols due to alignment constraints. If no sorting order is specified, then descending order is assumed.

'--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.

'--sysroot=DIRECTORY'
Use DIRECTORY as the location of the sysroot, overriding the configure-time default. This option is only supported by linkers that were configured using '--with-sysroot'.

'--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.

'-Ttext-segment=ORG'
     When creating an ELF executable, it will set the address of the
     first byte of the text segment.

'-Trodata-segment=ORG'
     When creating an ELF executable or shared object for a target where
     the read-only data is in its own segment separate from the
     executable text, it will set the address of the first byte of the
     read-only data segment.

'-Tldata-segment=ORG'
     When creating an ELF executable or shared object for x86-64 medium
     memory model, it will set the address of the first byte of the
     ldata segment.

'--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[=NUMBER]'
     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.  If the
     optional NUMBER argument > 1, plugin symbol status will also be
     displayed.
```

'--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 fully supported on ELF platforms which support shared libraries; see *note VERSION::. It is partially supported on PE platforms, which can use version scripts to filter symbol visibility in auto-export mode: any symbols marked 'local' in the version script will not be exported. *Note WIN32::.

'--warn-common'
Warn when a common symbol is combined with another common symbol or with a symbol definition. Unix linkers allow this somewhat sloppy practice, but linkers on some other operating systems do not. This option allows you to find potential problems from combining global symbols. Unfortunately, some C libraries use this practice, so you may get 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 (*note SECTIONS::).

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

'--warn-alternate-em'
     Warn if an object has alternate ELF machine code.

'--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'.

'--eh-frame-hdr'
'--no-eh-frame-hdr'
     Request ('--eh-frame-hdr') or suppress ('--no-eh-frame-hdr') the
     creation of '.eh_frame_hdr' section and ELF 'PT_GNU_EH_FRAME'
     segment header.

'--no-ld-generated-unwind-info'
     Request creation of '.eh_frame' unwind info for linker generated
     code sections like PLT. This option is on by default if linker
     generated unwind info is supported.
```

'--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 new dynamic tags will be created as
     needed and older dynamic tags will be omitted. 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.

'--hash-size=NUMBER'
     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.

'--hash-style=STYLE'
     Set the type of linker's hash table(s). STYLE can be either 'sysv'
     for classic ELF '.hash' section, 'gnu' for new style GNU
     '.gnu.hash' section or 'both' for both the classic ELF '.hash' and
     new style GNU '.gnu.hash' hash tables. The default is 'sysv'.

'--compress-debug-sections=none'
'--compress-debug-sections=zlib'
'--compress-debug-sections=zlib-gnu'
'--compress-debug-sections=zlib-gabi'
     On ELF platforms, these options control how DWARF debug sections
     are compressed using zlib.

     '--compress-debug-sections=none' doesn't compress DWARF debug
     sections. '--compress-debug-sections=zlib-gnu' compresses DWARF
     debug sections and renames them to begin with '.zdebug' instead of
     '.debug'. '--compress-debug-sections=zlib-gabi' also compresses
     DWARF debug sections, but rather than renaming them it sets the
     SHF_COMPRESSED flag in the sections' headers.

     The '--compress-debug-sections=zlib' option is an alias for
     '--compress-debug-sections=zlib-gabi'.

     Note that this option overrides any compression in input debug
     sections, so if a binary is linked with
     '--compress-debug-sections=none' for example, then any compressed
     debug sections in input files will be uncompressed before they are
     copied into the output binary.

     The default compression behaviour varies depending upon the target
     involved and the configure options used to build the toolchain.
     The default can be determined by examining the output from the
     linker's '--help' option.

'--reduce-memory-overheads'
     This option reduces memory requirements at ld runtime, at the
     expense of linking speed. This was introduced 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 effect 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.

'--build-id'
'--build-id=STYLE'
Request the creation of a '.note.gnu.build-id' ELF note section or a '.buildid' COFF section. The contents of the note are unique bits identifying this linked file. STYLE can be 'uuid' to use 128 random bits, 'sha1' to use a 160-bit SHA1 hash on the normative parts of the output contents, 'md5' to use a 128-bit MD5 hash on the normative parts of the output contents, or '0xHEXSTRING' to use a chosen bit string specified as an even number of hexadecimal digits ('-' and ':' characters between digit pairs are ignored). If STYLE is omitted, 'sha1' is used.

The 'md5' and 'sha1' styles produces an identifier that is always the same in an identical output file, but will be unique among all nonidentical output files. It is not intended to be compared as a checksum for the file's contents. A linked file may be changed later by other tools, but the build ID bit string identifying the original linked file does not change.

Passing 'none' for STYLE disables the setting from any '--build-id' options earlier on the command line.

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-long-section-names'
'--disable-long-section-names'
The PE variants of the COFF object format add an extension that permits the use of section names longer than eight characters, the normal limit for COFF. By default, these names are only allowed in object files, as fully-linked executable images do not carry the COFF string table required to support the longer names. As a GNU extension, it is possible to allow their use in executable images as well, or to (probably pointlessly!) disallow it in object files, by using these two options. Executable images generated with these long section names are slightly non-standard, carrying as they do a string table, and may generate confusing output when examined with non-GNU PE-aware tools, such as file viewers and dumpers. However, GDB relies on the use of PE long section names to find Dwarf-2 debug information sections in an executable image at runtime, and so if neither option is specified on the command-line, 'ld' will enable long section names, overriding the default and technically correct behaviour, when it finds the presence of debug information while linking an executable image and not stripping symbols. [This option is valid for all PE targeted ports 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]

'--leading-underscore'
'--no-leading-underscore'
For most targets default symbol-prefix is an underscore and is defined in target's description. By this option it is possible to disable/enable the default underscore symbol-prefix.

'--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]

'--exclude-all-symbols'
Specifies no symbols should be automatically exported. [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 number of bytes 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 "Characteristics" field of the COFF header is set to indicate that this executable supports virtual addresses greater than 2 gigabytes. This should be used in conjunction 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]

'--disable-large-address-aware'

Reverts the effect of a previous '--large-address-aware' option. This is useful if '--large-address-aware' is always set by the compiler driver (e.g. Cygwin gcc) and the executable does not support virtual addresses greater than 2 gigabytes. [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]

'--enable-auto-image-base'
'--enable-auto-image-base=VALUE'
Automatically choose the image base for DLLs, optionally starting with base VALUE, 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.

Note - use of the 'auto-import' extension will also cause read only data which would normally be placed into the .rdata section to be placed into the .data section instead. This is in order to work around a problem with consts that is described here: http://www.cygwin.com/ml/cygwin/2004-09/msg01101.html

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 practice 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 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 number of bytes 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]

     The following options set flags in the 'DllCharacteristics' field
     of the PE file header: [These options are specific to PE targeted
     ports of the linker]

'--high-entropy-va'
     Image is compatible with 64-bit address space layout randomization
```

(ASLR).

'--dynamicbase'
The image base address may be relocated using address space layout randomization (ASLR). This feature was introduced with MS Windows Vista for i386 PE targets.

'--forceinteg'
Code integrity checks are enforced.

'--nxcompat'
The image is compatible with the Data Execution Prevention. This feature was introduced with MS Windows XP SP2 for i386 PE targets.

'--no-isolation'
Although the image understands isolation, do not isolate the image.

'--no-seh'
The image does not use SEH. No SE handler may be called from this image.

'--no-bind'
Do not bind this image.

'--wdmdriver'
The driver uses the MS Windows Driver Model.

'--tsaware'
The image is Terminal Server aware.

'--insert-timestamp'
'--no-insert-timestamp'
Insert a real timestamp into the image. This is the default behaviour as it matches legacy code and it means that the image will work with other, proprietary tools. The problem with this default is that it will result in slightly different images being produced each time the same sources are linked. The option '--no-insert-timestamp' can be used to insert a zero value for the timestamp, this ensuring that binaries produced from identical sources will compare identically.

2.1.2 Options specific to C6X uClinux targets

The C6X uClinux target uses a binary format called DSBT to support shared libraries. Each shared library in the system needs to have a unique index; all executables use an index of 0.

'--dsbt-size SIZE'
This option sets the number of entries in the DSBT of the current executable or shared library to SIZE. The default is to create a table with 64 entries.

'--dsbt-index INDEX'
This option sets the DSBT index of the current executable or shared library to INDEX. The default is 0, which is appropriate for generating executables. If a shared library is generated with a DSBT index of 0, the 'R_C6000_DSBT_INDEX' relocs are copied into the output file.

The '--no-merge-exidx-entries' switch disables the merging of adjacent exidx entries in frame unwind info.

2.1.3 Options specific to Motorola 68HC11 and 68HC12 targets

The 68HC11 and 68HC12 linkers support specific options to control the memory bank switching mapping and trampoline code generation.

'--no-trampoline'
: This option disables the generation of trampoline. By default a trampoline is generated for each far function which is called using a 'jsr' instruction (this happens when a pointer to a far function is taken).

'--bank-window NAME'
: This option indicates to the linker the name of the memory region in the 'MEMORY' specification that describes the memory bank window. The definition of such region is then used by the linker to compute paging and addresses within the memory window.

2.1.4 Options specific to Motorola 68K target

The following options are supported to control handling of GOT generation when linking for 68K targets.

'--got=TYPE'
: This option tells the linker which GOT generation scheme to use. TYPE should be one of 'single', 'negative', 'multigot' or 'target'. For more information refer to the Info entry for 'ld'.

2.1.5 Options specific to MIPS targets

The following options are supported to control microMIPS instruction generation and branch relocation checks for ISA mode transitions when linking for MIPS targets.

'--insn32'
'--no-insn32'
: These options control the choice of microMIPS instructions used in code generated by the linker, such as that in the PLT or lazy binding stubs, or in relaxation. If '--insn32' is used, then the linker only uses 32-bit instruction encodings. By default or if '--no-insn32' is used, all instruction encodings are used, including 16-bit ones where possible.

'--ignore-branch-isa'
'--no-ignore-branch-isa'
: These options control branch relocation checks for invalid ISA mode transitions. If '--ignore-branch-isa' is used, then the linker accepts any branch relocations and any ISA mode transition required is lost in relocation calculation, except for some cases of 'BAL' instructions which meet relaxation conditions and are converted to equivalent 'JALX' instructions as the associated relocation is calculated. By default or if '--no-ignore-branch-isa' is used a check is made causing the loss of an ISA mode transition to produce

an error.

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

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 (*note BFD::). If there is no 'GNUTARGET' in the environment, 'ld' uses the natural format of the target. If 'GNUTARGET' is set to 'default' then BFD attempts to discover the input format by examining binary input files; this method often succeeds, but there are potential ambiguities, since there is no method of ensuring that the magic number used to specify object-file formats is unique. However, the configuration procedure for BFD on each system places the conventional format for that system first in the search-list, so ambiguities are resolved in favor of convention.

'LDEMULATION' determines the default emulation if you don't use the '-m' option. The emulation can affect various aspects of linker behaviour, particularly the default linker script. You can list the available emulations with the '--verbose' or '-V' options. If the '-m' option is not used, and the 'LDEMULATION' environment variable is not defined, the default emulation depends upon how the linker was configured.

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.

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

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. *Note Implicit Linker Scripts::.

* Menu:

File: ld.info, Node: Basic Script Concepts, Next: Script Format, Up: Scripts

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 means 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.

File: ld.info, Node: Script Format, Next: Simple Example, Prev: Basic Script Concepts, Up: Scripts

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.

File: ld.info, Node: Simple Example, Next: Simple Commands, Prev: Script Format, Up: Scripts

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.

us
File: ld.info, Node: Simple Commands, Next: Assignments, Prev: Simple Example, Up: Scripts

3.4 Simple Linker Script Commands

In this section we describe the simple linker script commands.

* Menu:

* Entry Point::   Setting the entry point
* File Commands::   Commands dealing with files
* Format Commands::   Commands dealing with object file formats

* REGION_ALIAS::   Assign alias names to memory regions
* Miscellaneous Commands:: Other linker script commands

File: ld.info, Node: Entry Point, Next: File Commands, Up: Simple 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 a target specific symbol, if it is defined; For many targets this is 'start', but PE and BeOS based systems for example check a list of possible entry symbols, matching the first one found.
* the address of the first byte of the '.text' section, if present;
* The address '0'.

File: ld.info, Node: File Commands, Next: Format Commands, Prev: Entry Point, Up: Simple Commands

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.

  You can place 'INCLUDE' directives at the top level, in 'MEMORY' or 'SECTIONS' commands, or in output section descriptions.

'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. The "sysroot prefix" can also be forced by specifying '=' as the first character in the filename path. See also the description of '-L' in *note Command Line Options: Options.

If you use 'INPUT (-lFILE)', 'ld' will transform the name to 'libFILE.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 *note Command Line Options: Options.

'AS_NEEDED(FILE, FILE, ...)'
'AS_NEEDED(FILE FILE ...)'
This construct can appear only inside of the 'INPUT' or 'GROUP' commands, among other filenames. The files listed will be handled as if they appear directly in the 'INPUT' or 'GROUP' commands, with the exception of ELF shared libraries, that will be added only when they are actually needed. This construct essentially enables '--as-needed' option for all the files listed inside of it and restores previous '--as-needed' resp. '--no-as-needed' setting afterwards.

'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 (*note Command Line Options: Options.). 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 (*note Command Line Options: Options.). 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.

File: ld.info, Node: Format Commands, Next: REGION_ALIAS, Prev: File Commands, Up: Simple Commands

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 (*note BFD::). Using 'OUTPUT_FORMAT(BFDNAME)' is exactly like using '--oformat BFDNAME' on the command line (*note Command Line Options: Options.). 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 (*note Command Line Options: Options.). 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. *Note BFD::.

File: ld.info, Node: REGION_ALIAS, Next: Miscellaneous Commands, Prev: Format Commands, Up: Simple Commands

3.4.4 Assign alias names to memory regions

Alias names can be added to existing memory regions created with the *note MEMORY:: command. Each name corresponds to at most one memory region.

```
     REGION_ALIAS(ALIAS, REGION)
```

   The 'REGION_ALIAS' function creates an alias name ALIAS for the memory region REGION. This allows a flexible mapping of output sections to memory regions. An example follows.

   Suppose we have an application for embedded systems which come with various memory storage devices. All have a general purpose, volatile memory 'RAM' that allows code execution or data storage. Some may have a read-only, non-volatile memory 'ROM' that allows code execution and read-only data access. The last variant is a read-only, non-volatile memory 'ROM2' with read-only data access and no code execution capability. We have four output sections:

   * '.text' program code;
   * '.rodata' read-only data;
   * '.data' read-write initialized data;
   * '.bss' read-write zero initialized data.

   The goal is to provide a linker command file that contains a system independent part defining the output sections and a system dependent part mapping the output sections to the memory regions available on the system. Our embedded systems come with three different memory setups 'A', 'B' and 'C':

Section	Variant A	Variant B	Variant C
.text	RAM	ROM	ROM
.rodata	RAM	ROM	ROM2
.data	RAM	RAM/ROM	RAM/ROM2
.bss	RAM	RAM	RAM

   The notation 'RAM/ROM' or 'RAM/ROM2' means that this section is loaded into region 'ROM' or 'ROM2' respectively. Please note that the load address of the '.data' section starts in all three variants at the end of the '.rodata' section.

   The base linker script that deals with the output sections follows. It includes the system dependent 'linkcmds.memory' file that describes the memory layout:

```
     INCLUDE linkcmds.memory

     SECTIONS
       {
         .text :
           {
             *(.text)
           } > REGION_TEXT
         .rodata :
           {
             *(.rodata)
             rodata_end = .;
           } > REGION_RODATA
         .data : AT (rodata_end)
           {
             data_start = .;
             *(.data)
           } > REGION_DATA
         data_size = SIZEOF(.data);
         data_load_start = LOADADDR(.data);
         .bss :
```

```
          {
            *(.bss)
          } > REGION_BSS
      }
```

   Now we need three different 'linkcmds.memory' files to define memory regions and alias names. The content of 'linkcmds.memory' for the three variants 'A', 'B' and 'C':

'A'

```
     Here everything goes into the 'RAM'.
          MEMORY
            {
              RAM : ORIGIN = 0, LENGTH = 4M
            }

          REGION_ALIAS("REGION_TEXT", RAM);
          REGION_ALIAS("REGION_RODATA", RAM);
          REGION_ALIAS("REGION_DATA", RAM);
          REGION_ALIAS("REGION_BSS", RAM);
```

'B'

```
     Program code and read-only data go into the 'ROM'.  Read-write data
     goes into the 'RAM'.  An image of the initialized data is loaded
     into the 'ROM' and will be copied during system start into the
     'RAM'.
          MEMORY
            {
              ROM : ORIGIN = 0, LENGTH = 3M
              RAM : ORIGIN = 0x10000000, LENGTH = 1M
            }

          REGION_ALIAS("REGION_TEXT", ROM);
          REGION_ALIAS("REGION_RODATA", ROM);
          REGION_ALIAS("REGION_DATA", RAM);
          REGION_ALIAS("REGION_BSS", RAM);
```

'C'

```
     Program code goes into the 'ROM'.  Read-only data goes into the
     'ROM2'.  Read-write data goes into the 'RAM'.  An image of the
     initialized data is loaded into the 'ROM2' and will be copied
     during system start into the 'RAM'.
          MEMORY
            {
              ROM : ORIGIN = 0, LENGTH = 2M
              ROM2 : ORIGIN = 0x10000000, LENGTH = 1M
              RAM : ORIGIN = 0x20000000, LENGTH = 1M
            }

          REGION_ALIAS("REGION_TEXT", ROM);
          REGION_ALIAS("REGION_RODATA", ROM2);
          REGION_ALIAS("REGION_DATA", RAM);
          REGION_ALIAS("REGION_BSS", RAM);
```

   It is possible to write a common system initialization routine to copy the '.data' section from 'ROM' or 'ROM2' into the 'RAM' if necessary:

```
     #include <string.h>

     extern char data_start [];
     extern char data_size [];
     extern char data_load_start [];
```

```
     void copy_data(void)
     {
       if (data_start != data_load_start)
         {
           memcpy(data_start, data_load_start, (size_t) data_size);
         }
     }
```

File: ld.info, Node: Miscellaneous Commands, Prev: REGION_ALIAS, Up: Simple Commands

3.4.5 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.

  Note that assertions are checked before the final stages of linking take place. This means that expressions involving symbols PROVIDEd inside section definitions will fail if the user has not set values for those symbols. The only exception to this rule is PROVIDEd symbols that just reference dot. Thus an assertion like this:

```
        .stack :
        {
          PROVIDE (__stack = .);
          PROVIDE (__stack_size = 0x100);
          ASSERT ((__stack > (_end + __stack_size)), "Error: No room left for the
        stack");
        }
```

  will fail if '__stack_size' is not defined elsewhere. Symbols PROVIDEd outside of section definitions are evaluated earlier, so they can be used inside ASSERTions. Thus:

```
        PROVIDE (__stack_size = 0x100);
        .stack :
        {
          PROVIDE (__stack = .);
          ASSERT ((__stack > (_end + __stack_size)), "Error: No room left for the
        stack");
        }
```

  will work.

'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 SYMBOLs 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.

'INSERT [AFTER | BEFORE] OUTPUT_SECTION'
This command is typically used in a script specified by '-T' to augment the default 'SECTIONS' with, for example, overlays. It inserts all prior linker script statements after (or before) OUTPUT_SECTION, and also causes '-T' to not override the default linker script. The exact insertion point is as for orphan sections. *Note Location Counter::. The insertion happens after the linker has mapped input sections to output sections. Prior to the insertion, since '-T' scripts are parsed before the default linker script, statements in the '-T' script occur before the default linker script statements in the internal linker representation of the script. In particular, input section assignments will be made to '-T' output sections before those in the default script. Here is an example of how a '-T' script using 'INSERT' might look:

```
SECTIONS
{
  OVERLAY :
  {
    .ov1 { ov1*(.text) }
    .ov2 { ov2*(.text) }
  }
}
INSERT AFTER .text;
```

'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.

'NOCROSSREFS_TO(TOSECTION FROMSECTION ...)'
This command may be used to tell 'ld' to issue an error about any references to one section from a list of other sections.

The 'NOCROSSREFS' command is useful when ensuring that two or more output sections are entirely independent but there are situations where a one-way dependency is needed. For example, in a multi-core application there may be shared code that can be called from each core but for safety must never call back.

The 'NOCROSSREFS_TO' command takes a list of output section names. The first section can not be referenced from any of the other sections. If 'ld' detects any references to the first section from any of the other sections, it reports an error and returns a non-zero exit status. Note that the 'NOCROSSREFS_TO' 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 (*note BFD::). You can see the architecture of an object file by using the 'objdump' program with the '-f' option.

'LD_FEATURE(STRING)'
This command may be used to modify 'ld' behavior. If STRING is '"SANE_EXPR"' then absolute symbols and numbers in a script are simply treated as numbers everywhere. *Note Expression Section::.

File: ld.info, Node: Assignments, Next: SECTIONS, Prev: Simple Commands, Up: Scripts

3.5 Assigning Values to Symbols

You may assign a value to a symbol in a linker script. This will define the symbol and place it into the symbol table with a global scope.

* Menu:

File: ld.info, Node: Simple Assignments, Next: HIDDEN, Up: Assignments

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. *Note Location Counter::.

The semicolon after EXPRESSION is required.

Expressions are defined below; see *note Expressions::.

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 *note Expression Section::.

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.

File: ld.info, Node: HIDDEN, Next: PROVIDE, Prev: Simple Assignments, Up: Assignments

3.5.2 HIDDEN

For ELF targeted ports, define a symbol that will be hidden and won't be exported. The syntax is 'HIDDEN(SYMBOL = EXPRESSION)'.

Here is the example from *note Simple Assignments::, rewritten to use 'HIDDEN':

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

In this case none of the three symbols will be visible outside this

module.

File: ld.info, Node: PROVIDE, Next: PROVIDE_HIDDEN, Prev: HIDDEN, Up: Assignments

3.5.3 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.

File: ld.info, Node: PROVIDE_HIDDEN, Next: Source Code Reference, Prev: PROVIDE, Up: Assignments

3.5.4 PROVIDE_HIDDEN

Similar to 'PROVIDE'. For ELF targeted ports, the symbol will be hidden and won't be exported.

File: ld.info, Node: Source Code Reference, Prev: PROVIDE_HIDDEN, Up: Assignments

3.5.5 Source Code Reference

Accessing a linker script defined variable from source code is not intuitive. In particular a linker script symbol is not equivalent to a variable declaration in a high level language, it is instead a symbol that does not have a value.

Before going further, it is important to note that compilers often transform names in the source code into different names when they are stored in the symbol table. For example, Fortran compilers commonly prepend or append an underscore, and C++ performs extensive 'name

mangling'. Therefore there might be a discrepancy between the name of a variable as it is used in source code and the name of the same variable as it is defined in a linker script. For example in C a linker script variable might be referred to as:

```
extern int foo;
```

But in the linker script it might be defined as:

```
_foo = 1000;
```

In the remaining examples however it is assumed that no name transformation has taken place.

When a symbol is declared in a high level language such as C, two things happen. The first is that the compiler reserves enough space in the program's memory to hold the _value_ of the symbol. The second is that the compiler creates an entry in the program's symbol table which holds the symbol's _address_. ie the symbol table contains the address of the block of memory holding the symbol's value. So for example the following C declaration, at file scope:

```
int foo = 1000;
```

creates an entry called 'foo' in the symbol table. This entry holds the address of an 'int' sized block of memory where the number 1000 is initially stored.

When a program references a symbol the compiler generates code that first accesses the symbol table to find the address of the symbol's memory block and then code to read the value from that memory block. So:

```
foo = 1;
```

looks up the symbol 'foo' in the symbol table, gets the address associated with this symbol and then writes the value 1 into that address. Whereas:

```
int * a = & foo;
```

looks up the symbol 'foo' in the symbol table, gets its address and then copies this address into the block of memory associated with the variable 'a'.

Linker scripts symbol declarations, by contrast, create an entry in the symbol table but do not assign any memory to them. Thus they are an address without a value. So for example the linker script definition:

```
foo = 1000;
```

creates an entry in the symbol table called 'foo' which holds the address of memory location 1000, but nothing special is stored at address 1000. This means that you cannot access the _value_ of a linker script defined symbol - it has no value - all you can do is access the _address_ of a linker script defined symbol.

Hence when you are using a linker script defined symbol in source code you should always take the address of the symbol, and never attempt

to use its value. For example suppose you want to copy the contents of a section of memory called .ROM into a section called .FLASH and the linker script contains these declarations:

```
start_of_ROM   = .ROM;
end_of_ROM     = .ROM + sizeof (.ROM);
start_of_FLASH = .FLASH;
```

Then the C source code to perform the copy would be:

```
extern char start_of_ROM, end_of_ROM, start_of_FLASH;

memcpy (& start_of_FLASH, & start_of_ROM, & end_of_ROM - & start_of_ROM);
```

Note the use of the '&' operators. These are correct. Alternatively the symbols can be treated as the names of vectors or arrays and then the code will again work as expected:

```
extern char start_of_ROM[], end_of_ROM[], start_of_FLASH[];

memcpy (start_of_FLASH, start_of_ROM, end_of_ROM - start_of_ROM);
```

Note how using this method does not require the use of '&' operators.

us

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 (*note Entry command: Entry Point.)
* a symbol assignment (*note Assignments::)
* 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.

* Menu:

* Output Section Description:: Output section description
* Output Section Name:: Output section name
* Output Section Address:: Output section address
* Input Section:: Input section description
* Output Section Data:: Output section data
* Output Section Keywords:: Output section keywords
* Output Section Discarding:: Output section discarding
* Output Section Attributes:: Output section attributes
* Overlay Description:: Overlay description

File: ld.info, Node: Output Section Description, Next: Output Section Name, Up: SECTIONS

3.6.1 Output Section Description

The full description of an output section looks like this:

```
     SECTION [ADDRESS] [(TYPE)] :
       [AT(LMA)]
       [ALIGN(SECTION_ALIGN) | ALIGN_WITH_INPUT]
       [SUBALIGN(SUBSECTION_ALIGN)]
       [CONSTRAINT]
       {
         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 comma at the end may be required if a FILLEXP is used and the next SECTIONS-COMMAND looks like a continuation of the expression. The line breaks and other white space are optional.

Each OUTPUT-SECTION-COMMAND may be one of the following:

* a symbol assignment (*note Assignments::)
* an input section description (*note Input Section::)
* data values to include directly (*note Output Section Data::)
* a special output section keyword (*note Output Section Keywords::)

File: ld.info, Node: Output Section Name, Next: Output Section Address, Prev: Output Section Description, Up: SECTIONS

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; *note Output Section Discarding::.

File: ld.info, Node: Output Section Address, Next: Input Section, Prev: Output Section Name, Up: SECTIONS

3.6.3 Output Section Address

The ADDRESS is an expression for the VMA (the virtual memory address) of the output section. This address is optional, but if it is provided then the output address will be set exactly as specified.

If the output address is not specified then one will be chosen for the section, based on the heuristic below. This address will be adjusted to fit the alignment requirement of the output section. The alignment requirement is the strictest alignment of any input section contained within the output section.

The output section address heuristic is as follows:

* If an output memory REGION is set for the section then it is added to this region and its address will be the next free address in that region.

* If the MEMORY command has been used to create a list of memory regions then the first region which has attributes compatible with the section is selected to contain it. The section's output address will be the next free address in that region; *note MEMORY::.

* If no memory regions were specified, or none match the section then the output address will be based on the current value of the location counter.

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 any of the '.text' input sections.

The ADDRESS may be an arbitrary expression; *note Expressions::. 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, provided that the section is non-empty. (Empty sections are ignored).

File: ld.info, Node: Input Section, Next: Output Section Data, Prev: Output Section Address, Up: SECTIONS

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.

* Menu:

File: ld.info, Node: Input Section Basics, Next: Input Section Wildcards, Up: Input Section

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 (*note Input Section Wildcards::).

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. The EXCLUDE_FILE can also be placed inside the section list, for example:

```
     *(EXCLUDE_FILE (*crtend.o *otherfile.o) .ctors)
```
The result of this is identically to the previous example. Supporting two syntaxes for EXCLUDE_FILE is useful if the section list contains more than one section, as described below.

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.

When using EXCLUDE_FILE with more than one section, if the exclusion is within the section list then the exclusion only applies to the immediately following section, for example:
```
     *(EXCLUDE_FILE (*somefile.o) .text .rdata)
```
will cause all '.text' sections from all files except 'somefile.o' to be included, while all '.rdata' sections from all files, including 'somefile.o', will be included. To exclude the '.rdata' sections from 'somefile.o' the example could be modified to:
```
     *(EXCLUDE_FILE (*somefile.o) .text EXCLUDE_FILE (*somefile.o) .rdata)
```
Alternatively, placing the EXCLUDE_FILE outside of the section list, before the input file selection, will cause the exclusion to apply for all sections. Thus the previous example can be rewritten as:
```
     EXCLUDE_FILE (*somefile.o) *(.text .rdata)
```

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)
```

To refine the sections that are included based on the section flags of an input section, INPUT_SECTION_FLAGS may be used.

Here is a simple example for using Section header flags for ELF sections:

```
     SECTIONS {
       .text : { INPUT_SECTION_FLAGS (SHF_MERGE & SHF_STRINGS) *(.text) }
       .text2 :  { INPUT_SECTION_FLAGS (!SHF_WRITE) *(.text) }
     }
```

In this example, the output section '.text' will be comprised of any input section matching the name *(.text) whose section header flags 'SHF_MERGE' and 'SHF_STRINGS' are set. The output section '.text2' will be comprised of any input section matching the name *(.text) whose section header flag 'SHF_WRITE' is clear.

You can also specify files within archives by writing a pattern matching the archive, a colon, then the pattern matching the file, with no whitespace around the colon.

```
'archive:file'
     matches file within archive
'archive:'
     matches the whole archive
':file'
```

matches file but not one in an archive

Either one or both of 'archive' and 'file' can contain shell wildcards. On DOS based file systems, the linker will assume that a single letter followed by a colon is a drive specifier, so 'c:myfile.o' is a simple file specification, not 'myfile.o' within an archive called 'c'. 'archive:file' filespecs may also be used within an 'EXCLUDE_FILE' list, but may not appear in other linker script contexts. For instance, you cannot extract a file from an archive by using 'archive:file' in an 'INPUT' command.

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 is not an 'archive:file' specifier and 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.

us
File: ld.info, Node: Input Section Wildcards, Next: Input Section Common, Prev: Input Section Basics, Up: Input Section

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 descending order by alignment before placing them in the output file. Larger alignments are placed before smaller alignments in order to reduce the amount of padding necessary.

'SORT_BY_INIT_PRIORITY' is very similar to 'SORT_BY_NAME'. The difference is 'SORT_BY_INIT_PRIORITY' will sort sections into ascending order by numerical value of the GCC init_priority attribute encoded in the section name 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 two 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 two 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.

'SORT_NONE' disables section sorting by ignoring the command line section sorting option.

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) }
}
```

us

File: ld.info, Node: Input Section Common, Next: Input Section Keep, Prev: Input Section Wildcards, Up: Input Section

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)'.

us

File: ld.info, Node: Input Section Keep, Next: Input Section Example, Prev: Input Section Common, Up: Input Section

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))'.

File: ld.info, Node: Input Section Example, Prev: Input Section Keep, Up: Input Section

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)
    }
}
```


File: ld.info, Node: Output Section Data, Next: Output Section Keywords, Prev: Input Section, Up: SECTIONS

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 (*note Expressions::). 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. *Note Output Section Fill::, for details on the fill expression.

File: ld.info, Node: Output Section Keywords, Next: Output Section Discarding, Prev: Output Section Data, Up: SECTIONS

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 '__CTOR_END__' marks the end. Similarly, '__DTOR_LIST__' and '__DTOR_END__' mark the start and end of the global destructors. 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.

File: ld.info, Node: Output Section Discarding, Next: Output Section Attributes, Prev: Output Section Keywords, Up: SECTIONS

3.6.7 Output Section Discarding

The linker will not normally create output sections with no 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, and if the input sections are not all empty. Other link script directives that allocate space in an output section will also create the output section. So too will assignments to dot even if the assignment does not create space, except for '. = 0', '. = . + 0', '. = sym', '. = . + sym' and '. = ALIGN (. != 0, expr, 1)' when 'sym' is an absolute symbol of value 0 defined in the script. This allows you to force output of an empty section with '. = .'.

The linker will ignore address assignments (*note Output Section Address::) on discarded output sections, except when the linker script defines symbols in the output section. In that case the linker will obey the address assignments, possibly advancing dot even though the section is discarded.

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.

File: ld.info, Node: Output Section Attributes, Next: Overlay Description, Prev: Output Section Discarding, Up: SECTIONS

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)]
  [ALIGN(SECTION_ALIGN)]
  [SUBALIGN(SUBSECTION_ALIGN)]
  [CONSTRAINT]
  {
    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.

* Menu:

```
* Output Section Type::     Output section type
* Output Section LMA::      Output section LMA
* Forced Output Alignment:: Forced Output Alignment
* Forced Input Alignment::  Forced Input Alignment
* Output Section Constraint::   Output section constraint
* Output Section Region::   Output section region
* Output Section Phdr::     Output section phdr
* Output Section Fill::     Output section fill
```


File: ld.info, Node: Output Section Type, Next: Output Section LMA, Up: Output 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.

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


File: ld.info, Node: Output Section LMA, Next: Forced Output Alignment, Prev: Output Section Type, Up: Output Section Attributes

3.6.8.2 Output Section LMA

Every section has a virtual address (VMA) and a load address (LMA); see *note Basic Script Concepts::. The virtual address is specified by the *note Output Section Address:: described earlier. The load address is specified by the 'AT' or 'AT>' keywords. Specifying a load address is optional.

The 'AT' keyword takes an expression as an argument. This specifies the exact load address of the section. The 'AT>' keyword takes the name of a memory region as an argument. *Note MEMORY::. The load address of the section is set to the next free address in the region, aligned to

the section's alignment requirements.

   If neither 'AT' nor 'AT>' is specified for an allocatable section, the linker will use the following heuristic to determine the load address:

   * If the section has a specific VMA address, then this is used as the LMA address as well.

   * If the section is not allocatable then its LMA is set to its VMA.

   * Otherwise if a memory region can be found that is compatible with the current section, and this region contains at least one section, then the LMA is set so the difference between the VMA and LMA is the same as the difference between the VMA and LMA of the last section in the located region.

   * If no memory regions have been declared then a default region that covers the entire address space is used in the previous step.

   * If no suitable region could be found, or there was no previous section then the LMA is set equal to the VMA.

   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;
```

File: ld.info, Node: Forced Output Alignment, Next: Forced Input Alignment, Prev: Output Section LMA, Up: Output Section Attributes

3.6.8.3 Forced Output Alignment
....

You can increase an output section's alignment by using ALIGN. As an alternative you can enforce that the difference between the VMA and LMA remains intact throughout this output section with the ALIGN_WITH_INPUT attribute.

File: ld.info, Node: Forced Input Alignment, Next: Output Section Constraint, Prev: Forced Output Alignment, Up: Output Section Attributes

3.6.8.4 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.

File: ld.info, Node: Output Section Constraint, Next: Output Section Region, Prev: Forced Input Alignment, Up: Output Section Attributes

3.6.8.5 Output Section Constraint
....

You can specify that an output section should only be created if all of its input sections are read-only or all of its input sections are read-write by using the keyword 'ONLY_IF_RO' and 'ONLY_IF_RW' respectively.

File: ld.info, Node: Output Section Region, Next: Output Section Phdr, Prev: Output Section Constraint, Up: Output Section Attributes

3.6.8.6 Output Section Region
....

You can assign a section to a previously defined region of memory by using '>REGION'. *Note MEMORY::.

Here is a simple example:

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

File: ld.info, Node: Output Section Phdr, Next: Output Section Fill, Prev: Output Section Region, Up: Output Section Attributes

3.6.8.7 Output Section Phdr
....

You can assign a section to a previously defined program segment by using ':PHDR'. *Note PHDRS::. If a section is assigned to one or more segments, then all subsequent allocated sections will be assigned to

those segments as well, unless they use an explicitly ':PHDR' modifier.
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 }
```


File: ld.info, Node: Output Section Fill, Prev: Output Section Phdr, Up: Output Section Attributes

3.6.8.8 Output Section Fill

You can set the fill pattern for an entire section by using '=FILLEXP'.
FILLEXP is an expression (*note Expressions::). 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; (*note Output Section Data::).

   Here is a simple example:
```
  SECTIONS { .text : { *(.text) } =0x90909090 }
```


File: ld.info, Node: Overlay Description, Prev: Output Section Attributes, Up: SECTIONS

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' construct (*note SECTIONS::), except that no addresses and no memory regions may be defined for sections within an 'OVERLAY'.

   The comma at the end may be required if a FILL is used and the next SECTIONS-COMMAND looks like a continuation of the expression.

   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 are 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. *Note NOCROSSREFS: Miscellaneous Commands.

   For each section within the 'OVERLAY', the linker automatically provides 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 if referenced: '__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) }
       PROVIDE (__load_start_text0 = LOADADDR (.text0));
       PROVIDE (__load_stop_text0 = LOADADDR (.text0) + SIZEOF (.text0));
       .text1 0x1000 : AT (0x4000 + SIZEOF (.text0)) { o2/*.o(.text) }
       PROVIDE (__load_start_text1 = LOADADDR (.text1));
       PROVIDE (__load_stop_text1 = LOADADDR (.text1) + SIZEOF (.text1));
       . = 0x1000 + MAX (SIZEOF (.text0), SIZEOF (.text1));
```

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

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 many uses of the 'MEMORY' command, however, all memory blocks defined are treated as if they were specified inside a single 'MEMORY' command. The syntax for 'MEMORY' 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 within the 'MEMORY' command. However you can add later alias names to existing memory regions with the *note REGION_ALIAS:: command.

   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 *note SECTIONS::, 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 attributes that follow

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. *Note Output Section Region::. 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. The headers are processed in order and it is usual for them to map to sections in ascending load address order.

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. *Note Output Section Phdr::.

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 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. If applied to a loadable segment ('PT_LOAD'), all prior loadable segments must have one of these keywords.

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.

'PT_TLS' (7)
Indicates a segment containing thread local storage.

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 (*note Output Section LMA::). 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
       ...
     }
```

File: ld.info, Node: VERSION, Next: Expressions, Prev: PHDRS, Up: Scripts

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;
        extern "C++" {
            ns::*;
            "f(int, double)";
         };
    } 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"). However, if you specify the symbol name inside double quotes, then the name is treated as literal, rather than as a glob pattern.

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. Note that it's slightly crazy to use wildcards in a global spec except on the last version node. Global wildcards elsewhere run the risk of accidentally adding symbols to the set exported for an old version. That's wrong since older versions ought to have a fixed set of symbols.

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 omitted, 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'. The default 'lang' is 'C'.

   Demangled names may contains spaces and other special characters. As described above, you can use a glob pattern to match demangled names, or you can use a double-quoted string to match the string exactly. In the latter case, be aware that minor differences (such as differing whitespace) between the version script and the demangler output will cause a mismatch. As the exact string generated by the demangler might change in the future, even if the mangled name does not, you should check that all of your version directives are behaving as you expect when you upgrade.

File: ld.info, Node: Expressions, Next: Implicit Linker Scripts, Prev: VERSION, Up: Scripts

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.

* Menu:

File: ld.info, Node: Constants, Next: Symbolic Constants, Up: 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.
Alternatively the linker accepts suffixes of 'h' or 'H' for hexadecimal,
'o' or 'O' for octal, 'b' or 'B' for binary and 'd' or 'D' for decimal.
Any integer value without a prefix or a suffix is considered to be
decimal.

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

```
     _fourk_1 = 4K;
     _fourk_2 = 4096;
     _fourk_3 = 0x1000;
     _fourk_4 = 10000o;
```

   Note - the 'K' and 'M' suffixes cannot be used in conjunction with
the base suffixes mentioned above.

File: ld.info, Node: Symbolic Constants, Next: Symbols, Prev: Constants, Up: Expressions

3.10.2 Symbolic Constants

It is possible to refer to target specific constants via the use of the
'CONSTANT(NAME)' operator, where NAME is one of:

'MAXPAGESIZE'
     The target's maximum page size.

'COMMONPAGESIZE'
     The target's default page size.

   So for example:

```
       .text ALIGN (CONSTANT (MAXPAGESIZE)) : { *(.text) }
```

   will create a text section aligned to the largest page boundary
supported by the target.

File: ld.info, Node: Symbols, Next: Orphan Sections, Prev: Symbolic Constants, Up: Expressions

3.10.3 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.

File: ld.info, Node: Orphan Sections, Next: Location Counter, Prev: Symbols, Up: Expressions

3.10.4 Orphan Sections

Orphan sections are sections present in the input files which are not explicitly placed into the output file by the linker script. The linker will still copy these sections into the output file, but it has to guess as to where they should be placed. The linker uses a simple heuristic to do this. It attempts to place orphan sections after non-orphan sections of the same attribute, such as code vs data, loadable vs non-loadable, etc. If there is not enough room to do this then it places at the end of the file.

For ELF targets, the attribute of the section includes section type as well as section flag.

The command line options '--orphan-handling' and '--unique' (*note Command Line Options: Options.) can be used to control which output sections an orphan is placed in.

If an orphaned section's name is representable as a C identifier then the linker will automatically *note PROVIDE:: two symbols: __start_SECNAME and __stop_SECNAME, where SECNAME is the name of the section. These indicate the start address and end address of the orphaned section respectively. Note: most section names are not representable as C identifiers because they contain a '.' character.

File: ld.info, Node: Location Counter, Next: Operators, Prev: Orphan Sections, Up: Expressions

3.10.5 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 not be moved backwards inside an output section, and may not be moved backwards outside of an output section if so doing creates areas with overlapping LMAs.

```
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 (*note Output Section Fill::).

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.

Setting symbols to the value of the location counter outside of an output section statement can result in unexpected values if the linker needs to place orphan sections. For example, given the following:

```
    SECTIONS
    {
        start_of_text = . ;
        .text: { *(.text) }
        end_of_text = . ;

        start_of_data = . ;
        .data: { *(.data) }
        end_of_data = . ;
    }
```

If the linker needs to place some input section, e.g. '.rodata', not mentioned in the script, it might choose to place that section between '.text' and '.data'. You might think the linker should place '.rodata' on the blank line in the above script, but blank lines are of no particular significance to the linker. As well, the linker doesn't

associate the above symbol names with their sections. Instead, it assumes that all assignments or other statements belong to the previous output section, except for the special case of an assignment to '.'. I.e., the linker will place the orphan '.rodata' section as if the script was written as follows:

```
SECTIONS
{
    start_of_text = . ;
    .text: { *(.text) }
    end_of_text = . ;

    start_of_data = . ;
    .rodata: { *(.rodata) }
    .data: { *(.data) }
    end_of_data = . ;
}
```

This may or may not be the script author's intention for the value of 'start_of_data'. One way to influence the orphan section placement is to assign the location counter to itself, as the linker assumes that an assignment to '.' is setting the start address of a following output section and thus should be grouped with that section. So you could write:

```
SECTIONS
{
    start_of_text = . ;
    .text: { *(.text) }
    end_of_text = . ;

    . = . ;
    start_of_data = . ;
    .data: { *(.data) }
    end_of_data = . ;
}
```

Now, the orphan '.rodata' section will be placed between 'end_of_text' and 'start_of_data'.

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

3.10.6 Operators

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

precedence (highest)	associativity	Operators	Notes
1	left	! - ~	(1)
2	left	* / %	
3	left	+ -	
4	left	>> <<	
5	left	== != > < <= >=	
6	left	&	
7	left	|	
8	left	&&	

```
     9              left           ||
     10             right          ? :
     11             right          &=  +=  -=  *=  /=       (2)
     (lowest)
   Notes: (1) Prefix operators (2) *Note Assignments::.
```


File: ld.info, Node: Evaluation, Next: Expression Section, Prev: Operators, Up: Expressions

3.10.7 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'.

File: ld.info, Node: Expression Section, Next: Builtin Functions, Prev: Evaluation, Up: Expressions

3.10.8 The Section of an Expression

Addresses and symbols may be section relative, or absolute. A section relative symbol is relocatable. If you request relocatable output using the '-r' option, a further link operation may change the value of a section relative symbol. On the other hand, an absolute symbol will retain the same value throughout any further link operations.

Some terms in linker expressions are addresses. This is true of section relative symbols and for builtin functions that return an

address, such as 'ADDR', 'LOADADDR', 'ORIGIN' and 'SEGMENT_START'. Other terms are simply numbers, or are builtin functions that return a non-address value, such as 'LENGTH'. One complication is that unless you set 'LD_FEATURE ("SANE_EXPR")' (*note Miscellaneous Commands::), numbers and absolute symbols are treated differently depending on their location, for compatibility with older versions of 'ld'. Expressions appearing outside an output section definition treat all numbers as absolute addresses. Expressions appearing inside an output section definition treat absolute symbols as numbers. If 'LD_FEATURE ("SANE_EXPR")' is given, then absolute symbols and numbers are simply treated as numbers everywhere.

In the following simple example,

```
SECTIONS
  {
    . = 0x100;
    __executable_start = 0x100;
    .data :
    {
      . = 0x10;
      __data_start = 0x10;
      *(.data)
    }
    ...
  }
```

both '.' and '__executable_start' are set to the absolute address 0x100 in the first two assignments, then both '.' and '__data_start' are set to 0x10 relative to the '.data' section in the second two assignments.

For expressions involving numbers, relative addresses and absolute addresses, ld follows these rules to evaluate terms:

* Unary operations on an absolute address or number, and binary operations on two absolute addresses or two numbers, or between one absolute address and a number, apply the operator to the value(s).
* Unary operations on a relative address, and binary operations on two relative addresses in the same section or between one relative address and a number, apply the operator to the offset part of the address(es).
* Other binary operations, that is, between two relative addresses not in the same section, or between a relative address and an absolute address, first convert any non-absolute term to an absolute address before applying the operator.

The result section of each sub-expression is as follows:

* An operation involving only numbers results in a number.
* The result of comparisons, '&&' and '||' is also a number.
* The result of other binary arithmetic and logical operations on two relative addresses in the same section or two absolute addresses (after above conversions) is also a number when 'LD_FEATURE ("SANE_EXPR")' or inside an output section definition but an absolute address otherwise.
* The result of other operations on relative addresses or one relative address and a number, is a relative address in the same section as the relative operand(s).

* The result of other operations on absolute addresses (after above conversions) is an absolute address.

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.

Using 'LOADADDR' also forces an expression absolute, since this particular builtin function returns an absolute address.

us
File: ld.info, Node: Builtin Functions, Prev: Expression Section, Up: Expressions

3.10.9 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. *Note Expression Section::.

'ADDR(SECTION)'
Return the address (VMA) of the named SECTION. Your script must previously have defined the location of that section. In the following example, 'start_of_output_1', 'symbol_1' and 'symbol_2' are assigned equivalent values, except that 'symbol_1' will be relative to the '.output1' section while the other two will be absolute:

```
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(ABSOLUTE(.), 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 (*note Output Section Address::).  The second
     use of 'ALIGN' is used to defines the value of a symbol.

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

'ALIGNOF(SECTION)'
     Return the alignment 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, the alignment of the '.output' section is stored
     as the first value in that section.
          SECTIONS{ ...
            .output {
              LONG (ALIGNOF (.output))
              ...
              }
          ... }

'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)
           + ((. + COMMONPAGESIZE - 1) & (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. 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'. Evaluates to the
     second argument plus any padding needed at the end of the
     'PT_GNU_RELRO' segment due to section alignment.

            . = 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. (*note Output
     Section LMA::).

'LOG2CEIL(EXP)'
     Return the binary logarithm of EXP rounded towards infinity.
     'LOG2CEIL(0)' returns 0.

'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 already been given for this segment (with a command-line '-T'
     option) then 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 can 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 (*note PHDRS::).
```

File: ld.info, Node: Implicit Linker Scripts, Prev: Expressions, Up: Scripts

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.

File: ld.info, Node: Machine Dependent, Next: BFD, Prev: Scripts, Up: Top

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.

* Menu:

File: ld.info, Node: H8/300, Next: i960, Up: Machine Dependent

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).

     'ld' finds all 'mov' instructions which use the register indirect with 32-bit displacement addressing mode, but use a small

displacement inside 16-bit displacement range, and changes them to use the 16-bit displacement form. (That is: the linker turns 'mov.b '@'D:32,ERx' into 'mov.b '@'D:16,ERx' whenever the displacement D is in the 16 bit signed integer range. Only implemented in ELF-format ld).

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' instructions 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).

File: ld.info, Node: i960, Next: M68HC11/68HC12, Prev: H8/300, Up: Machine Dependent

4.2 'ld' and the Intel 960 Family

You can use the '-AARCHITECTURE' 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).

File: ld.info, Node: M68HC11/68HC12, Next: ARM, Prev: i960, Up: Machine Dependent

4.3 'ld' and the Motorola 68HC11 and 68HC12 families

4.3.1 Linker Relaxation

For the Motorola 68HC11, '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.

'ld' also looks at all 16-bit extended addressing modes and transforms them in a direct addressing mode when the address is in page 0 (between 0 and 0x0ff).

relaxing gcc instruction group
When 'gcc' is called with '-mrelax', it can emit group of instructions that the linker can optimize to use a 68HC11 direct addressing mode. These instructions consists of 'bclr' or 'bset' instructions.

4.3.2 Trampoline Generation

For 68HC11 and 68HC12, 'ld' can generate trampoline code to call a far
function using a normal 'jsr' instruction. The linker will also change
the relocation to some far function to use the trampoline address
instead of the function address. This is typically the case when a
pointer to a function is taken. The pointer will in fact point to the
function trampoline.

File: ld.info, Node: ARM, Next: HPPA ELF32, Prev: M68HC11/68HC12, Up: Machine Dependent

4.4 'ld' and the ARM family

For the ARM, 'ld' will generate code stubs to allow functions calls
between 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 '--use-nul-prefixed-import-tables' switch is specifying, that the import tables idata4 and idata5 have to be generated with a zero element prefix for import libraries. This is the old style to generate import tables. By default this option is turned off.

The '--be8' switch instructs 'ld' to generate BE8 format executables. This option is only valid when linking big-endian objects - ie ones which have been assembled with the '-EB' option. 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)

The 'R_ARM_V4BX' relocation (defined by the ARM AAELF specification) enables objects compiled for the ARMv4 architecture to be interworking-safe when linked with other objects compiled for ARMv4t, but also allows pure ARMv4 binaries to be built from the same ARMv4 objects.

In the latter case, the switch '--fix-v4bx' must be passed to the linker, which causes v4t 'BX rM' instructions to be rewritten as 'MOV PC,rM', since v4 processors do not have a 'BX' instruction.

In the former case, the switch should not be used, and 'R_ARM_V4BX' relocations are ignored.

Replace 'BX rM' instructions identified by 'R_ARM_V4BX' relocations with a branch to the following veneer:

```
TST rM, #1
MOVEQ PC, rM
BX Rn
```

This allows generation of libraries/applications that work on ARMv4 cores and are still interworking safe. Note that the above veneer clobbers the condition flags, so may cause incorrect program behavior in rare cases.

The '--use-blx' switch enables the linker to use ARM/Thumb BLX instructions (available on ARMv5t and above) in various situations. Currently it is used to perform calls via the PLT from Thumb code using BLX rather than using BX and a mode-switching stub before each PLT

entry. This should lead to such calls executing slightly faster.

This option is enabled implicitly for SymbianOS, so there is no need to specify it if you are using that target.

The '--vfp11-denorm-fix' switch enables a link-time workaround for a bug in certain VFP11 coprocessor hardware, which sometimes allows instructions with denorm operands (which must be handled by support code) to have those operands overwritten by subsequent instructions before the support code can read the intended values.

The bug may be avoided in scalar mode if you allow at least one intervening instruction between a VFP11 instruction which uses a register and another instruction which writes to the same register, or at least two intervening instructions if vector mode is in use. The bug only affects full-compliance floating-point mode: you do not need this workaround if you are using "runfast" mode. Please contact ARM for further details.

If you know you are using buggy VFP11 hardware, you can enable this workaround by specifying the linker option '--vfp-denorm-fix=scalar' if you are using the VFP11 scalar mode only, or '--vfp-denorm-fix=vector' if you are using vector mode (the latter also works for scalar code). The default is '--vfp-denorm-fix=none'.

If the workaround is enabled, instructions are scanned for potentially-troublesome sequences, and a veneer is created for each such sequence which may trigger the erratum. The veneer consists of the first instruction of the sequence and a branch back to the subsequent instruction. The original instruction is then replaced with a branch to the veneer. The extra cycles required to call and return from the veneer are sufficient to avoid the erratum in both the scalar and vector cases.

The '--fix-arm1176' switch enables a link-time workaround for an erratum in certain ARM1176 processors. The workaround is enabled by default if you are targeting ARM v6 (excluding ARM v6T2) or earlier. It can be disabled unconditionally by specifying '--no-fix-arm1176'.

Further information is available in the "ARM1176JZ-S and ARM1176JZF-S Programmer Advice Notice" available on the ARM documentation website at: http://infocenter.arm.com/.

The '--fix-stm32l4xx-629360' switch enables a link-time workaround for a bug in the bus matrix / memory controller for some of the STM32 Cortex-M4 based products (STM32L4xx). When accessing off-chip memory via the affected bus for bus reads of 9 words or more, the bus can generate corrupt data and/or abort. These are only core-initiated accesses (not DMA), and might affect any access: integer loads such as LDM, POP and floating-point loads such as VLDM, VPOP. Stores are not affected.

The bug can be avoided by splitting memory accesses into the necessary chunks to keep bus reads below 8 words.

The workaround is not enabled by default, this is equivalent to use '--fix-stm32l4xx-629360=none'. If you know you are using buggy STM32L4xx hardware, you can enable the workaround by specifying the linker option '--fix-stm32l4xx-629360', or the equivalent

'--fix-stm32l4xx-629360=default'.

If the workaround is enabled, instructions are scanned for potentially-troublesome sequences, and a veneer is created for each such sequence which may trigger the erratum. The veneer consists in a replacement sequence emulating the behaviour of the original one and a branch back to the subsequent instruction. The original instruction is then replaced with a branch to the veneer.

The workaround does not always preserve the memory access order for the LDMDB instruction, when the instruction loads the PC.

The workaround is not able to handle problematic instructions when they are in the middle of an IT block, since a branch is not allowed there. In that case, the linker reports a warning and no replacement occurs.

The workaround is not able to replace problematic instructions with a PC-relative branch instruction if the '.text' section is too large. In that case, when the branch that replaces the original code cannot be encoded, the linker reports a warning and no replacement occurs.

The '--no-enum-size-warning' switch prevents the linker from warning when linking object files that specify incompatible EABI enumeration size attributes. For example, with this switch enabled, linking of an object file using 32-bit enumeration values with another using enumeration values fitted into the smallest possible space will not be diagnosed.

The '--no-wchar-size-warning' switch prevents the linker from warning when linking object files that specify incompatible EABI 'wchar_t' size attributes. For example, with this switch enabled, linking of an object file using 32-bit 'wchar_t' values with another using 16-bit 'wchar_t' values will not be diagnosed.

The '--pic-veneer' switch makes the linker use PIC sequences for ARM/Thumb interworking veneers, even if the rest of the binary is not PIC. This avoids problems on uClinux targets where '--emit-relocs' is used to generate relocatable binaries.

The linker will automatically generate and insert small sequences of code into a linked ARM ELF executable whenever an attempt is made to perform a function call to a symbol that is too far away. The placement of these sequences of instructions - called stubs - is controlled by the command line option '--stub-group-size=N'. The placement is important because a poor choice can create a need for duplicate stubs, increasing the code size. The linker will try to group stubs together in order to reduce interruptions to the flow of code, but it needs guidance as to how big these groups should be and where they should be placed.

The value of 'N', the parameter to the '--stub-group-size=' option controls where the stub groups are placed. If it is negative then all stubs are placed after the first branch that needs them. If it is positive then the stubs can be placed either before or after the branches that need them. If the value of 'N' is 1 (either +1 or -1) then the linker will choose exactly where to place groups of stubs, using its built in heuristics. A value of 'N' greater than 1 (or smaller than -1) tells the linker that a single group of stubs can service at most 'N' bytes from the input sections.

The default, if '--stub-group-size=' is not specified, is 'N = +1'.

Farcalls stubs insertion is fully supported for the ARM-EABI target only, because it relies on object files properties not present otherwise.

The '--fix-cortex-a8' switch enables a link-time workaround for an erratum in certain Cortex-A8 processors. The workaround is enabled by default if you are targeting the ARM v7-A architecture profile. It can be enabled otherwise by specifying '--fix-cortex-a8', or disabled unconditionally by specifying '--no-fix-cortex-a8'.

The erratum only affects Thumb-2 code. Please contact ARM for further details.

The '--fix-cortex-a53-835769' switch enables a link-time workaround for erratum 835769 present on certain early revisions of Cortex-A53 processors. The workaround is disabled by default. It can be enabled by specifying '--fix-cortex-a53-835769', or disabled unconditionally by specifying '--no-fix-cortex-a53-835769'.

Please contact ARM for further details.

The '--no-merge-exidx-entries' switch disables the merging of adjacent exidx entries in debuginfo.

The '--long-plt' option enables the use of 16 byte PLT entries which support up to 4Gb of code. The default is to use 12 byte PLT entries which only support 512Mb of code.

The '--no-apply-dynamic-relocs' option makes AArch64 linker do not apply link-time values for dynamic relocations.

All SG veneers are placed in the special output section '.gnu.sgstubs'. Its start address must be set, either with the command line option '--section-start' or in a linker script, to indicate where to place these veneers in memory.

The '--cmse-implib' option requests that the import libraries specified by the '--out-implib' and '--in-implib' options are secure gateway import libraries, suitable for linking a non-secure executable against secure code as per ARMv8-M Security Extensions.

The '--in-implib=file' specifies an input import library whose symbols must keep the same address in the executable being produced. A warning is given if no '--out-implib' is given but new symbols have been introduced in the executable that should be listed in its import library. Otherwise, if '--out-implib' is specified, the symbols are added to the output import library. A warning is also given if some symbols present in the input import library have disappeared from the executable. This option is only effective for Secure Gateway import libraries, ie. when '--cmse-implib' is specified.

File: ld.info, Node: HPPA ELF32, Next: M68K, Prev: ARM, Up: Machine Dependent

4.5 '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.

File: ld.info, Node: M68K, Next: MIPS, Prev: HPPA ELF32, Up: Machine Dependent

4.6 'ld' and the Motorola 68K family

The '--got=TYPE' option lets you choose the GOT generation scheme. The choices are 'single', 'negative', 'multigot' and 'target'. When 'target' is selected the linker chooses the default GOT generation scheme for the current target. 'single' tells the linker to generate a single GOT with entries only at non-negative offsets. 'negative' instructs the linker to generate a single GOT with entries at both negative and positive offsets. Not all environments support such GOTs. 'multigot' allows the linker to generate several GOTs in the output file. All GOT references from a single input object file access the same GOT, but references from different input object files might access different GOTs. Not all environments support such GOTs.

File: ld.info, Node: MIPS, Next: MMIX, Prev: M68K, Up: Machine Dependent

4.7 'ld' and the MIPS family

The '--insn32' and '--no-insn32' options control the choice of microMIPS instructions used in code generated by the linker, such as that in the PLT or lazy binding stubs, or in relaxation. If '--insn32' is used, then the linker only uses 32-bit instruction encodings. By default or if '--no-insn32' is used, all instruction encodings are used, including 16-bit ones where possible.

The '--ignore-branch-isa' and '--no-ignore-branch-isa' options control branch relocation checks for invalid ISA mode transitions. If

'--ignore-branch-isa' is used, then the linker accepts any branch relocations and any ISA mode transition required is lost in relocation calculation, except for some cases of 'BAL' instructions which meet relaxation conditions and are converted to equivalent 'JALX' instructions as the associated relocation is calculated. By default or if '--no-ignore-branch-isa' is used a check is made causing the loss of an ISA mode transition to produce an error.

File: ld.info, Node: MMIX, Next: MSP430, Prev: MIPS, Up: Machine Dependent

4.8 '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.

Global symbols with the prefix '__.MMIX.start.', for example '__.MMIX.start..text' and '__.MMIX.start..data' are special. 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.

File: ld.info, Node: MSP430, Next: NDS32, Prev: MMIX, Up: Machine Dependent

4.9 '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.

us
File: ld.info, Node: NDS32, Next: Nios II, Prev: MSP430, Up: Machine Dependent

4.10 'ld' and NDS32

For NDS32, there are some options to select relaxation behavior. The linker relaxes objects according to these options.

''--m[no-]fp-as-gp''
Disable/enable fp-as-gp relaxation.

''--mexport-symbols=FILE''
Exporting symbols and their address into FILE as linker script.

''--m[no-]ex9''
Disable/enable link-time EX9 relaxation.

''--mexport-ex9=FILE''
Export the EX9 table after linking.

''--mimport-ex9=FILE''
Import the Ex9 table for EX9 relaxation.

''--mupdate-ex9''
Update the existing EX9 table.

''--mex9-limit=NUM''
Maximum number of entries in the ex9 table.

''--mex9-loop-aware''
Avoid generating the EX9 instruction inside the loop.

''--m[no-]ifc''
Disable/enable the link-time IFC optimization.

''--mifc-loop-aware''
Avoid generating the IFC instruction inside the loop.

us
File: ld.info, Node: Nios II, Next: PowerPC ELF32, Prev: NDS32, Up: Machine Dependent

4.11 'ld' and the Altera Nios II

Call and immediate jump instructions on Nios II processors are limited to transferring control to addresses in the same 256MB memory segment,

which may result in 'ld' giving 'relocation truncated to fit' errors with very large programs. The command-line option '--relax' enables the generation of trampolines that can access the entire 32-bit address space for calls outside the normal 'call' and 'jmpi' address range. These trampolines are inserted at section boundaries, so may not themselves be reachable if an input section and its associated call trampolines are larger than 256MB.

The '--relax' option is enabled by default unless '-r' is also specified. You can disable trampoline generation by using the '--no-relax' linker option. You can also disable this optimization locally by using the 'set .noat' directive in assembly-language source files, as the linker-inserted trampolines use the 'at' register as a temporary.

Note that the linker '--relax' option is independent of assembler relaxation options, and that using the GNU assembler's '-relax-all' option interferes with the linker's more selective call instruction relaxation.

File: ld.info, Node: PowerPC ELF32, Next: PowerPC64 ELF64, Prev: Nios II, Up: Machine Dependent

4.12 'ld' and PowerPC 32-bit ELF Support

Branches on PowerPC processors are limited to a signed 26-bit displacement, which may result in 'ld' giving 'relocation truncated to fit' errors with very large programs. '--relax' enables the generation of trampolines that can access the entire 32-bit address space. These trampolines are inserted at section boundaries, so may not themselves be reachable if an input section exceeds 33M in size. You may combine '-r' and '--relax' to add trampolines in a partial link. In that case both branches to undefined symbols and inter-section branches are also considered potentially out of range, and trampolines inserted.

'--bss-plt'
: Current PowerPC GCC accepts a '-msecure-plt' option that generates code capable of using a newer PLT and GOT layout that has the security advantage of no executable section ever needing to be writable and no writable section ever being executable. PowerPC 'ld' will generate this layout, including stubs to access the PLT, if all input files (including startup and static libraries) were compiled with '-msecure-plt'. '--bss-plt' forces the old BSS PLT (and GOT layout) which can give slightly better performance.

'--secure-plt'
: 'ld' will use the new PLT and GOT layout if it is linking new '-fpic' or '-fPIC' code, but does not do so automatically when linking non-PIC code. This option requests the new PLT and GOT layout. A warning will be given if some object file requires the old style BSS PLT.

'--sdata-got'
: The new secure PLT and GOT are placed differently relative to other sections compared to older BSS PLT and GOT placement. The location of '.plt' must change because the new secure PLT is an initialized section while the old PLT is uninitialized. The reason for the

'.got' change is more subtle: The new placement allows '.got' to be read-only in applications linked with '-z relro -z now'. However, this placement means that '.sdata' cannot always be used in shared libraries, because the PowerPC ABI accesses '.sdata' in shared libraries from the GOT pointer. '--sdata-got' forces the old GOT placement. PowerPC GCC doesn't use '.sdata' in shared libraries, so this option is really only useful for other compilers that may do so.

'--emit-stub-syms'
This option causes 'ld' to label linker stubs with a local symbol that encodes the stub type and destination.

'--no-tls-optimize'
PowerPC 'ld' normally performs some optimization of code sequences used to access Thread-Local Storage. Use this option to disable the optimization.

us
File: ld.info, Node: PowerPC64 ELF64, Next: SPU ELF, Prev: PowerPC ELF32, Up: Machine Dependent

4.13 'ld' and PowerPC64 64-bit ELF Support

'--stub-group-size'
Long branch stubs, PLT call stubs and TOC adjusting 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.

'--emit-stub-syms'
This option causes 'ld' to label linker stubs with a local symbol that encodes the stub type and destination.

'--dotsyms'
'--no-dotsyms'
These two options control how 'ld' interprets version patterns in a version script. Older PowerPC64 compilers emitted both a function descriptor symbol with the same name as the function, and a code entry symbol with the name prefixed by a dot ('.'). To properly version a function 'foo', the version script thus needs to control

both 'foo' and '.foo'. The option '--dotsyms', on by default, automatically adds the required dot-prefixed patterns. Use '--no-dotsyms' to disable this feature.

'--save-restore-funcs'
'--no-save-restore-funcs'
These two options control whether PowerPC64 'ld' automatically provides out-of-line register save and restore functions used by '-Os' code. The default is to provide any such referenced function for a normal final link, and to not do so for a relocatable link.

'--no-tls-optimize'
PowerPC64 'ld' normally performs some optimization of code sequences used to access Thread-Local Storage. Use this option to disable the optimization.

'--tls-get-addr-optimize'
'--no-tls-get-addr-optimize'
These options control whether PowerPC64 'ld' uses a special stub to call __tls_get_addr. PowerPC64 glibc 2.22 and later support an optimization that allows the second and subsequent calls to '__tls_get_addr' for a given symbol to be resolved by the special stub without calling in to glibc. By default the linker enables this option when glibc advertises the availability of __tls_get_addr_opt. Forcing this option on when using an older glibc won't do much besides slow down your applications, but may be useful if linking an application against an older glibc with the expectation that it will normally be used on systems having a newer glibc.

'--no-opd-optimize'
PowerPC64 'ld' normally removes '.opd' section entries corresponding to deleted link-once functions, or functions removed by the action of '--gc-sections' or linker script '/DISCARD/'. Use this option to disable '.opd' optimization.

'--non-overlapping-opd'
Some PowerPC64 compilers have an option to generate compressed '.opd' entries spaced 16 bytes apart, overlapping the third word, the static chain pointer (unused in C) with the first word of the next entry. This option expands such entries to the full 24 bytes.

'--no-toc-optimize'
PowerPC64 'ld' normally removes unused '.toc' section entries. Such entries are detected by examining relocations that reference the TOC in code sections. A reloc in a deleted code section marks a TOC word as unneeded, while a reloc in a kept code section marks a TOC word as needed. Since the TOC may reference itself, TOC relocs are also examined. TOC words marked as both needed and unneeded will of course be kept. TOC words without any referencing reloc are assumed to be part of a multi-word entry, and are kept or discarded as per the nearest marked preceding word. This works reliably for compiler generated code, but may be incorrect if assembly code is used to insert TOC entries. Use this option to disable the optimization.

'--no-multi-toc'
If given any toc option besides '-mcmodel=medium' or '-mcmodel=large', PowerPC64 GCC generates code for a TOC model

where TOC entries are accessed with a 16-bit offset from r2. This limits the total TOC size to 64K. PowerPC64 'ld' extends this limit by grouping code sections such that each group uses less than 64K for its TOC entries, then inserts r2 adjusting stubs between inter-group calls. 'ld' does not split apart input sections, so cannot help if a single input file has a '.toc' section that exceeds 64K, most likely from linking multiple files with 'ld -r'. Use this option to turn off this feature.

'--no-toc-sort'
By default, 'ld' sorts TOC sections so that those whose file happens to have a section called '.init' or '.fini' are placed first, followed by TOC sections referenced by code generated with PowerPC64 gcc's '-mcmodel=small', and lastly TOC sections referenced only by code generated with PowerPC64 gcc's '-mcmodel=medium' or '-mcmodel=large' options. Doing this results in better TOC grouping for multi-TOC. Use this option to turn off this feature.

'--plt-align'
'--no-plt-align'
Use these options to control whether individual PLT call stubs are padded so that they don't cross a 32-byte boundary, or to the specified power of two boundary when using '--plt-align='. Note that this isn't alignment in the usual sense. By default PLT call stubs are packed tightly.

'--plt-static-chain'
'--no-plt-static-chain'
Use these options to control whether PLT call stubs load the static chain pointer (r11). 'ld' defaults to not loading the static chain since there is never any need to do so on a PLT call.

'--plt-thread-safe'
'--no-thread-safe'
With power7's weakly ordered memory model, it is possible when using lazy binding for ld.so to update a plt entry in one thread and have another thread see the individual plt entry words update in the wrong order, despite ld.so carefully writing in the correct order and using memory write barriers. To avoid this we need some sort of read barrier in the call stub, or use LD_BIND_NOW=1. By default, 'ld' looks for calls to commonly used functions that create threads, and if seen, adds the necessary barriers. Use these options to change the default behaviour.

File: ld.info, Node: SPU ELF, Next: TI COFF, Prev: PowerPC64 ELF64, Up: Machine Dependent

4.14 'ld' and SPU ELF Support

'--plugin'
This option marks an executable as a PIC plugin module.

'--no-overlays'
Normally, 'ld' recognizes calls to functions within overlay regions, and redirects such calls to an overlay manager via a stub. 'ld' also provides a built-in overlay manager. This option turns

off all this special overlay handling.

'--emit-stub-syms'
This option causes 'ld' to label overlay stubs with a local symbol that encodes the stub type and destination.

'--extra-overlay-stubs'
This option causes 'ld' to add overlay call stubs on all function calls out of overlay regions. Normally stubs are not added on calls to non-overlay regions.

'--local-store=lo:hi'
'ld' usually checks that a final executable for SPU fits in the address range 0 to 256k. This option may be used to change the range. Disable the check entirely with '--local-store=0:0'.

'--stack-analysis'
SPU local store space is limited. Over-allocation of stack space unnecessarily limits space available for code and data, while under-allocation results in runtime failures. If given this option, 'ld' will provide an estimate of maximum stack usage. 'ld' does this by examining symbols in code sections to determine the extents of functions, and looking at function prologues for stack adjusting instructions. A call-graph is created by looking for relocations on branch instructions. The graph is then searched for the maximum stack usage path. Note that this analysis does not find calls made via function pointers, and does not handle recursion and other cycles in the call graph. Stack usage may be under-estimated if your code makes such calls. Also, stack usage for dynamic allocation, e.g. alloca, will not be detected. If a link map is requested, detailed information about each function's stack usage and calls will be given.

'--emit-stack-syms'
This option, if given along with '--stack-analysis' will result in 'ld' emitting stack sizing symbols for each function. These take the form '__stack_<function_name>' for global functions, and '__stack_<number>_<function_name>' for static functions. '<number>' is the section id in hex. The value of such symbols is the stack requirement for the corresponding function. The symbol size will be zero, type 'STT_NOTYPE', binding 'STB_LOCAL', and section 'SHN_ABS'.

us
File: ld.info, Node: TI COFF, Next: WIN32, Prev: SPU ELF, Up: Machine Dependent

4.15 '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.

us
File: ld.info, Node: WIN32, Next: Xtensa, Prev: TI COFF, Up: Machine Dependent

4.16 'ld' and WIN32 (cygwin/mingw)

==========

This section describes some of the win32 specific 'ld' issues. See *note Command Line Options: Options. for detailed description 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
* -exclude-modules-for-implib
* -version-script

When auto-export is in operation, 'ld' will export all the non-local (global and common) symbols it finds in a DLL, with the exception of a few symbols known to belong to the system's runtime and libraries. As it will often not be desirable to export all of a DLL's symbols, which may include private functions that are not part of any public interface, the command-line options listed above may be used to filter symbols out from the list for exporting. The '--output-def' option can be used in order to see the final list of exported symbols with all exclusions taken into effect.

If '--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=0x20000000

     EXPORTS
     foo
     bar
     _bar = bar
     another_foo = abc.dll.afoo
     var1 DATA
     doo = foo == foo2
     eoo DATA == var1
```

This example defines a DLL with a non-default base address and seven symbols in the export table. The third exported symbol '_bar' is an alias for the second. The fourth symbol, 'another_foo' is resolved by "forwarding" to another module and treating it as an alias for 'afoo' exported from the DLL 'abc.dll'. The final symbol 'var1' is declared to be a data object. The 'doo' symbol in export library is an alias of 'foo', which gets the string name in export table 'foo2'. The 'eoo' symbol is an data export symbol, which gets in export table the name 'var1'.

The optional 'LIBRARY <name>' command indicates the _internal_ name of the output DLL. If '<name>' does not include a suffix, the default library suffix, '.DLL' is appended.

When the .DEF file is used to build an application, rather than a library, the 'NAME <name>' command should be used instead of 'LIBRARY'. If '<name>' does not include a suffix, the default executable suffix, '.EXE' is appended.

With either 'LIBRARY <name>' or 'NAME <name>' the optional specification 'BASE = <number>' may be used to specify a non-default base address for the image.

If neither 'LIBRARY <name>' nor 'NAME <name>' is specified, or they specify an empty string, the internal name is the same as the filename specified on the command line.

The complete specification of an export symbol is:

```
     EXPORTS
       ( (  ( <name1> [ = <name2> ] )
          | ( <name1> = <module-name> . <external-name>))
       [ @ <integer> ] [NONAME] [DATA] [CONSTANT] [PRIVATE] [== <name3>] )
       *
```

Declares '<name1>' as an exported symbol from the DLL, or declares '<name1>' as an exported alias for '<name2>'; or declares '<name1>' as a "forward" alias for the symbol '<external-name>' in the DLL '<module-name>'. Optionally, the symbol may be exported by the specified ordinal '<integer>' alias. The optional '<name3>' is the to be used string in import/export table for the symbol.

The optional keywords that follow the declaration indicate:

'NONAME': Do not put the symbol name in the DLL's export table. It will still be exported by its ordinal alias (either the value specified by the .def specification or, otherwise, the value assigned by the linker). The symbol name, however, does remain visible in the import library (if any), unless 'PRIVATE' is also specified.

'DATA': The symbol is a variable or object, rather than a function. The import lib will export only an indirect reference to 'foo' as the symbol '_imp__foo' (ie, 'foo' must be resolved as '*_imp__foo').

'CONSTANT': Like 'DATA', but put the undecorated 'foo' as well as '_imp__foo' into the import library. Both refer to the read-only import address table's pointer to the variable, not to the variable itself. This can be dangerous. If the user code fails to add the 'dllimport' attribute and also fails to explicitly add the extra indirection that the use of the attribute enforces, the application will behave unexpectedly.

'PRIVATE': Put the symbol in the DLL's export table, but do not put it into the static import library used to resolve imports at link time. The symbol can still be imported using the 'LoadLibrary/GetProcAddress' API at runtime or by by using the GNU ld extension of linking directly to the DLL without an import library.

See ld/deffilep.y in the binutils sources for the full specification of other DEF file statements

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 information.

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 achieve 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, especially 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
xxx.lib
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 three 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.

3. Symbol aliases can only be resolved using an import lib. This is critical when linking against OS-supplied dll's (eg, the win32 API) in which symbols are usually exported as undecorated aliases of their stdcall-decorated assembly names.

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 many 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.

aligned common symbols

As a GNU extension to the PE file format, it is possible to specify the desired alignment for a common symbol. This information is conveyed from the assembler or compiler to the linker by means of GNU-specific commands carried in the object file's '.drectve' section, which are recognized by 'ld' and respected when laying out the common symbols. Native tools will be able to process object files employing this GNU extension, but will fail to respect the alignment instructions, and may issue noisy warnings about unknown linker directives.

File: ld.info, Node: Xtensa, Prev: WIN32, Up: Machine Dependent

4.17 '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'/'CALLXN' when the target functions are within range of direct 'CALLN' instructions.

For each of these cases where an indirect call sequence can be optimized to a direct call, the linker will change the 'CALLXN' instruction to a 'CALLN' 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:

'--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.

us
File: ld.info, Node: BFD, Next: Reporting Bugs, Prev: Machine Dependent, Up: Top

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' (*note objdump: (binutils.info)objdump.) 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 BFD information loss in the BFD internal documentation.

* Menu:

us
File: ld.info, Node: BFD outline, Up: BFD

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.

* Menu:

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.

File: ld.info, Node: Canonical format, Prev: BFD information loss, Up: BFD outline

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).

File: ld.info, Node: Reporting Bugs, Next: MRI, Prev: BFD, Up: Top

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.

* Menu:

File: ld.info, Node: Bug Criteria, Next: Bug Reporting, Up: Reporting Bugs

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.

File: ld.info, Node: Bug Reporting, Prev: Bug Criteria, Up: Reporting Bugs

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 <http://www.sourceware.org/bugzilla/>.

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 sync, 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.

us
File: ld.info, Node: MRI, Next: GNU Free Documentation License, Prev: Reporting Bugs, Up: Top

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 *note Scripts::. 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.

'ALIGN 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.

File: ld.info, Node: GNU Free Documentation License, Next: LD Index, Prev: MRI, Up: Top

Appendix B GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright (C) 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc. <http://fsf.org/>

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 functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are

listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

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 (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If 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 computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. 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 fewer than five), unless they release you from this requirement.

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", Preserve 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. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, 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 to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

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, and that you preserve all their Warranty Disclaimers.

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, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which 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 half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket 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, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements",

"Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

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. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the

site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

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 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled ``GNU Free Documentation License''.

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

File: ld.info, Node: LD Index, Prev: GNU Free Documentation License, Up: Top

LD Index
****

[index]
* Menu:

Tag Table:

Node: Implicit Linker Scripts225767
Node: Machine Dependent226542
Node: H8/300227649
Node: i960229712
Node: M68HC11/68HC12231408
Node: ARM232853
Node: HPPA ELF32245100
Node: M68K246723
Node: MIPS247632
Node: MMIX248748
Node: MSP430249913
Node: NDS32250953
Node: Nios II251919
Node: PowerPC ELF32253235
Node: PowerPC64 ELF64256066
Node: SPU ELF263391
Node: TI COFF266025
Node: WIN32266551
Node: Xtensa286680
Node: BFD289646
Node: BFD outline291134
Node: BFD information loss292422
Node: Canonical format294948
Node: Reporting Bugs299310
Node: Bug Criteria300004
Node: Bug Reporting300703
Node: MRI307741
Node: GNU Free Documentation License312383
Node: LD Index337520

End Tag Table