Using ld

The GNU linker

`ld` version 2
Version 2.9-mipssde-030910

Steve Chamberlain
Ian Lance Taylor
Cygnus Solutions

Cygnus Solutions
ian@cygnus.com, doc@cygnus.com
Using LD, the GNU linker
Edited by Jeffrey Osier (jeffrey@cygnus.com)

1 Overview

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

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

This version of `ld` uses the general purpose BFD libraries to operate on object files. This allows `ld` to read, combine, and write object files in many different formats—for example, COFF or `a.out`. Different formats may be linked together to produce any available kind of object file. See Chapter 5 [BFD], page 53, for more information.

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

2 Invocation

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

2.1 Command Line Options

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

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

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

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

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

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

If the linker can not recognize the format of an object file, it will assume that it is a linker script. A script specified in this way augments the main linker script used for the link (either the default linker script or the one specified by using '`-T`'). This feature permits the linker to link against a file which appears to be an object or an archive, but actually merely defines some symbol values, or uses `INPUT` or `GROUP` to load other objects. Note that specifying a script in this way should only be used to augment the main linker script; if you want to use some command that logically can only appear once, such as the `SECTIONS` or `MEMORY` command, you must replace the default linker script using the '`-T`' option. See Chapter 3 [Scripts], page 21.

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, '`--oformat`' and '`--oformat`' are equivalent. 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, '`--oformat srec`' and '`--oformat=srec`' 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 (eg '`gcc`') then all the linker command line options should be prefixed by '`-Wl,`' (or whatever is appropriate for the particular compiler driver) like this:

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

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

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

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

`-A`*architecture*

`--architecture=`*architecture*

In the current release of `ld`, this option is useful only for the Intel 960 family of architectures. In that `ld` configuration, the *architecture* argument identifies the particular architecture in the 960 family, enabling some safeguards and modifying the archive-library search path. See Section 4.2 [`ld` and the Intel 960 family], page 51, for details.

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

`-b` *input-format*

`--format=`*input-format*

`ld` may be configured to support more than one kind of object file. If your `ld` is configured this way, you can use the '`-b`' option to specify the binary format for input object files that follow this option on the command line. Even when `ld` is configured to support alternative object formats, you don't usually need to specify this, as `ld` should be configured to expect as a default input format the most usual format on each machine. *input-format* is a text string, the name of a particular format supported by the BFD libraries. (You can list the available binary formats with '`objdump -i`'.) See Chapter 5 [BFD], page 53.

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

The default format is taken from the environment variable `GNUTARGET`. See Section 2.2 [Environment], page 19. You can also define the input format from a script, using the command `TARGET`; see Section 3.4.3 [Format Commands], page 24.

`-c` *MRI-commandfile*

`--mri-script=`*MRI-commandfile*

For compatibility with linkers produced by MRI, `ld` accepts script files written in an alternate, restricted command language, described in Appendix A [MRI Compatible Script Files], page 61. Introduce MRI script files with the option '`-c`'; use the '`-T`' option to run linker scripts written in the general-purpose `ld` scripting language. If *MRI-cmdfile* does not exist, `ld` looks for it in the directories specified by any '`-L`' options.

`-d`

`-dc`

`-dp` These three options are equivalent; multiple forms are supported for compatibility with other linkers. They assign space to common symbols even if a relocatable output file

is specified (with '-r'). The script command FORCE_COMMON_ALLOCATION has the same effect. See Section 3.4.4 [Miscellaneous Commands], page 25.

-e *entry*
--entry=*entry*

Use *entry* as the explicit symbol for beginning execution of your program, rather than the default entry point. If there is no symbol named *entry*, the linker will try to parse *entry* as a number, and use that as the entry address (the number will be interpreted in base 10; you may use a leading '0x' for base 16, or a leading '0' for base 8). See Section 3.4.1 [Entry Point], page 23, for a discussion of defaults and other ways of specifying the entry point.

-E
--export-dynamic

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

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

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

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

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

-f
--auxiliary *name*

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

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

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

-F *name*
--filter *name*

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

If you later link a program against this filter object, then, when you run the program, the dynamic linker will see the DT_FILTER field. The dynamic linker will resolve

symbols according to the symbol table of the filter object as usual, but it will actually link to the definitions found in the shared object *name*. Thus the filter object can be used to select a subset of the symbols provided by the object *name*.

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

`-fini` *name*

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

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

`-G`*value*
`--gpsize=`*value*

Set the maximum size of objects to be optimized using the GP register to *size*. This is only meaningful for object file formats such as MIPS ECOFF which supports putting large and small objects into different sections. This is ignored for other object file formats.

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

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

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

`-init` *name*

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

`-l`*archive*
`--library=`*archive*

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

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

The linker will search an archive only once, at the location where it is specified on the command line. If the archive defines a symbol which was undefined in some object which appeared before the archive on the command line, the linker will include

the appropriate file(s) from the archive. However, an undefined symbol in an object appearing later on the command line will not cause the linker to search the archive again.

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

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

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

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

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

The default set of paths searched (without being specified with '`-L`') depends on which emulation mode `ld` is using, and in some cases also on how it was configured. See Section 2.2 [Environment], page 19.

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

`-m`*emulation*

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

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

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

`-M`
`--print-map`

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

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

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

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

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

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

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

`-r`
`--relocateable`
: 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`'.

  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. You must use this option if you want to use a command which can only appear once in a linker script, such as the `SECTIONS` or `MEMORY` command. See Chapter 3 [Scripts], page 21. If *scriptfile* does not exist in the current directory, `ld` looks for it in the directories specified by any preceding '`-L`' options. Multiple '`-T`' options accumulate.

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

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

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

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

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

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

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

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

`-z` *keyword*
: This option is 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.

`-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.

`-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.

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

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

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

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

`--defsym` *symbol=expression*
: Create a global symbol in the output file, containing the absolute address given by *expression*. You may use this option as many times as necessary to define multiple symbols in the command line. A limited form of arithmetic is supported for the *expression* in this context: you may give a hexadecimal constant or the name of an existing symbol, or use + and - to add or subtract hexadecimal constants or symbols. If you need more elaborate expressions, consider using the linker command language from a script (see Section 3.5 [Assignment: Symbol Definitions], page 26). *Note:* there should be no white space between *symbol*, the equals sign ("="), and *expression*.

`--demangle`
`--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. The linker will demangle by default unless the environment variable '`COLLECT_NO_DEMANGLE`' is set. These options may be used to override the default.

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

`--embedded-relocs`
: This option is only meaningful when linking MIPS embedded PIC code, generated by the -membedded-pic option to the GNU compiler and assembler. It causes the linker to create a table which may be used at runtime to relocate any data which was statically initialized to pointer values. See the code in testsuite/ld-empic for details.

`--errors-to-file` *file*
: This option is useful on deficient systems that cannot otherwise redirect stderr to a file.

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

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

`--no-gc-sections`
`--gc-sections`
: Enable garbage collection of unused input sections. It is ignored on targets that do not support this option. This option is not compatible with '`-r`', nor should it be used with dynamic linking. The default behaviour (of not performing this garbage collection) can be restored by specifying '`--no-gc-sections`' on the command line.

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

`-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`
: Normally when creating a non-symbolic shared library, undefined symbols are allowed and left to be resolved by the runtime loader. This option disallows such undefined symbols.

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

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

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

`--oformat` *output-format*
: `ld` may be configured to support more than one kind of object file. If your `ld` is configured this way, you can use the '`--oformat`' option to specify the binary format for the output object file. Even when `ld` is configured to support alternative object formats, you don't usually need to specify this, as `ld` should be configured to produce as a default output format the most usual format on each machine. *output-format* is a text string, the name of a particular format supported by the BFD libraries. (You can list the available binary formats with '`objdump -i`'.) The script command `OUTPUT_FORMAT` can also specify the output format, but this option overrides it. See Chapter 5 [BFD], page 53.

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

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

`--relax` An option with machine dependent effects. This option is only supported on a few targets. See Section 4.1 [`ld` and the H8/300], page 51. See Section 4.2 [`ld` and the Intel 960 family], page 51.

On some platforms, the '`--relax`' option performs global optimizations that become possible when the linker resolves addressing in the program, such as relaxing address modes and synthesizing new instructions in the output object file.

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

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

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

‘--retain-symbols-file’ does *not* discard undefined symbols, or symbols needed for relocations.

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

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

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

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

`-rpath-link` *DIR*

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

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

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

1. Any directories specified by `-rpath-link` options.
2. Any directories specified by `-rpath` options. The difference between `-rpath` and `-rpath-link` is that directories specified by `-rpath` options are included in the executable and used at runtime, whereas the `-rpath-link` option is only effective at link time.
3. On an ELF system, 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, the contents of the environment variable `LD_LIBRARY_PATH`.
6. The default directories, normally ‘`/lib`’ and ‘`/usr/lib`’.
7. For a native linker on an ELF system, if the file ‘`/etc/ld.so.conf`’ exists, the list of directories found in that file.

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

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

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

`--split-by-file`
: Similar to `--split-by-reloc` but creates a new output section for each input file.

`--split-by-reloc` *count*
: Trys 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 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.

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

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

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

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

`-Tbss` *org*
`-Tdata` *org*
`-Ttext` *org*
: Use *org* as the starting address for—respectively—the `bss`, `data`, or the `text` segment of the output file. *org* must be a single hexadecimal integer; for compatibility with other linkers, you may omit the leading '`0x`' usually associated with hexadecimal values.

`--dll-verbose`

`--verbose` Display the version number for `ld` and list the linker emulations supported. Display which input files can and cannot be opened. Display the linker script if using a default builtin script.

`--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 heirarchy for the library being created. This option is only meaningful on ELF platforms which support shared libraries. See Section 3.9 [VERSION], page 41.

`--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 (see Section 3.6 [SECTIONS], page 27).

`--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.

`--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 (int c)
{
  printf ("malloc called with %ld\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`.

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.

`--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`’.

`--dll` Create a DLL instead of a regular executable. You may also use `-shared` or specify a `LIBRARY` in a given `.def` file.

`--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.

`--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`, and `impure_ptr` will not be automatically exported.

`--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.

`--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.

`--heap` *reserve*
`--heap` *reserve*, *commit*
: Specify the amount of memory to reserve (and optionally commit) to be used as heap for this program. The default is 1Mb reserved, 4K committed.

`--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.

`--kill-at` If given, the stdcall suffixes (@*nn*) will be stripped from symbols before they are exported.

`--major-image-version` *value*
: Sets the major number of the "image version". Defaults to 1.

`--major-os-version` *value*
: Sets the major number of the "os version". Defaults to 4.

`--major-subsystem-version` *value*
: Sets the major number of the "subsystem version". Defaults to 4.

`--minor-image-version` *value*
: Sets the minor number of the "image version". Defaults to 0.

`--minor-os-version` *value*
: Sets the minor number of the "os version". Defaults to 0.

`--minor-subsystem-version` *value*
: Sets the minor number of the "subsystem version". Defaults to 0.

`--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.

`--section-alignment`
: Sets the section alignment. Sections in memory will always begin at addresses which are a multiple of this number. Defaults to 0x1000.

`--stack` *reserve*
`--stack` *reserve*,*commit*
: Specify the amount of memory to reserve (and optionally commit) to be used as stack for this program. The default is 32Mb reserved, 4K committed.

`--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`, and `posix`. You may optionally set the subsystem version also.

2.2 Environment Variables

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

`GNUTARGET` determines the input-file object format if you don't use '`-b`' (or its synonym '`--format`'). Its value should be one of the BFD names for an input format (see Chapter 5 [BFD], page 53). If there is no `GNUTARGET` in the environment, `ld` uses the natural format of the target. If `GNUTARGET` is set to `default` then BFD attempts to discover the input format by examining binary input files; this method often succeeds, but there are potential ambiguities, since there is no method of ensuring that the magic number used to specify object-file formats is unique. However, the configuration procedure for BFD on each system places the conventional format for that system first in the search-list, so ambiguities are resolved in favor of convention.

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

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

3 Linker Scripts

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

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

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

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

You may also use linker scripts implicitly by naming them as input files to the linker, as though they were files to be linked. See Section 3.11 [Implicit Linker Scripts], page 49.

3.1 Basic Linker Script Concepts

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

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

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

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

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

Every object file also has a list of *symbols*, known as the *symbol table*. A symbol may be defined or undefined. Each symbol has a name, and each defined symbol has an address, among other information. If you compile a C or C++ program into an object file, you will get a defined symbol for every defined function and global or static variable. Every undefined function or global variable which is referenced in the input file will become an undefined symbol.

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

3.2 Linker Script Format

Linker scripts are text files.

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

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

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

3.3 Simple Linker Script Example

Many linker scripts are fairly simple.

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

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

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

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

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

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

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

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

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

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

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

3.4 Simple Linker Script Commands

In this section we describe the simple linker script commands.

3.4.1 Setting the entry point

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

`ENTRY(`*symbol*`)`

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

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

3.4.2 Commands dealing with files

Several linker script commands deal with files.

`INCLUDE` *filename*

Include the linker script *filename* at this point. The file will be searched for in the current directory, and in any directory specified with the `-L` option. You can nest calls to `INCLUDE` up to 10 levels deep.

`INPUT(`*file*, *file*, `...)`
`INPUT(`*file file* `...)`

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

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

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

The linker will first try to open the file in the current directory. If it is not found, the linker will search through the archive library search path. See the description of '`-L`' in Section 2.1 [Command Line Options], page 3.

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

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

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

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

`OUTPUT(`*filename*`)`

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

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

`SEARCH_DIR(`*path*`)`

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

`STARTUP(`*filename*`)`

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

3.4.3 Commands dealing with object file formats

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

`OUTPUT_FORMAT(`*bfdname*`)`
`OUTPUT_FORMAT(`*default*, *big*, *little*`)`

The `OUTPUT_FORMAT` command names the BFD format to use for the output file (see Chapter 5 [BFD], page 53). Using `OUTPUT_FORMAT(`*bfdname*`)` is exactly like using '`-oformat` *bfdname*' on the command line (see Section 2.1 [Command Line Options], page 3). If both are used, the command line option takes precedence.

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

If neither '`-EB`' nor '`-EL`' are used, then the output format will be the first argument, *default*. If '`-EB`' is used, the output format will be the second argument, *big*. If '`-EL`' is used, the output format will be the third argument, *little*.

For example, the default linker script for the MIPS ELF target uses this command:

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

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

`TARGET(`*bfdname*`)`

The `TARGET` command names the BFD format to use when reading input files. It affects subsequent `INPUT` and `GROUP` commands. This command is like using '`-b` *bfdname*' on the command line (see Section 2.1 [Command Line Options], page 3). If the `TARGET` command is used but `OUTPUT_FORMAT` is not, then the last `TARGET` command is also used to set the format for the output file. See Chapter 5 [BFD], page 53.

3.4.4 Other linker script commands

There are a few other linker scripts commands.

`ASSERT(`*exp*, *message*`)`

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

`EXTERN(`*symbol symbol* `...)`

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

`FORCE_COMMON_ALLOCATION`

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

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

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

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

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

`OUTPUT_ARCH(`*bfdarch*`)`

Specify a particular output machine architecture. The argument is one of the names used by the BFD library (see Chapter 5 [BFD], page 53). You can see the architecture of an object file by using the `objdump` program with the '`-f`' option.

3.5 Assigning Values to Symbols

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

3.5.1 Simple Assignments

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

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

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

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

The semicolon after *expression* is required.

Expressions are defined below; see Section 3.10 [Expressions], page 43.

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

The section of the symbol will be set from the section of the expression; for more information, see Section 3.10.6 [Expression Section], page 46.

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

```
floating_point = 0;
SECTIONS
{
  .text :
    {
      *(.text)
      _etext = .;
    }
  _bdata = (. + 3) & ~ 4;
  .data : { *(.data) }
}
```

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

3.5.2 PROVIDE

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

Here is an example of using PROVIDE to define ‘etext’:

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

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

3.6 SECTIONS command

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

The format of the SECTIONS command is:

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

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

- an ENTRY command (see Section 3.4.1 [Entry command], page 23)
- a symbol assignment (see Section 3.5 [Assignments], page 26)
- an output section description
- an overlay description

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

Output section descriptions and overlay descriptions are described below.

If you do not use a SECTIONS command in your linker script, the linker will place each input section into an identically named output section in the order that the sections are first encountered in the

input files. If all input sections are present in the first file, for example, the order of sections in the output file will match the order in the first input file. The first section will be at address zero.

3.6.1 Output section description

The full description of an output section looks like this:

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

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

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

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

- a symbol assignment (see Section 3.5 [Assignments], page 26)
- an input section description (see Section 3.6.4 [Input Section], page 29)
- data values to include directly (see Section 3.6.5 [Output Section Data], page 32)
- a special output section keyword (see Section 3.6.6 [Output Section Keywords], page 33)

3.6.2 Output section name

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

The output section name '`/DISCARD/`' is special; Section 3.6.7 [Output Section Discarding], page 34.

3.6.3 Output section address

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

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

For example,

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

and

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

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

The *address* may be an arbitrary expression; Section 3.10 [Expressions], page 43. For example, if you want to align the section on a 0x10 byte boundary, so that the lowest four bits of the section address are zero, you could do something like this:

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

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

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

3.6.4 Input section description

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

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

3.6.4.1 Input section basics

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

The file name and the section name may be wildcard patterns, which we describe further below (see Section 3.6.4.2 [Input Section Wildcards], page 30).

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

```
*(.text)
```

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

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

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

There are two ways to include more than one section:

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

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

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

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

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

```
data.o
```

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

3.6.4.2 Input section wildcard patterns

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

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

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

'`*`' matches any number of characters

'`?`' matches any single character

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

'`\`' quotes the following character

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

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

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

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

Normally, the linker will place files and sections matched by wildcards in the order in which they are seen during the link. You can change this by using the `SORT` keyword, which appears before a wildcard pattern in parentheses (e.g., `SORT(.text*)`). When the `SORT` keyword is used, the linker will sort the files or sections into ascending order by name before placing them in the output file.

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

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

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

3.6.4.3 Input section for common symbols

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

You may use file names with the '`COMMON`' section just as with any other input sections. You can use this to place common symbols from a particular input file in one section while common symbols from other input files are placed in another section.

In most cases, common symbols in input files will be placed in the '`.bss`' section in the output file. For example:

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

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

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

3.6.4.4 Input section and garbage collection

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

3.6.4.5 Input section example

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

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

3.6.5 Output section data

You can include explicit bytes of data in an output section by using `BYTE`, `SHORT`, `LONG`, `QUAD`, or `SQUAD` as an output section command. Each keyword is followed by an expression in parentheses providing the value to store (see Section 3.10 [Expressions], page 43). 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 two least significant bytes 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 '`0x9090`':

```
FILL(0x9090)
```

The `FILL` command is similar to the '*=fillexp*' output section attribute (see Section 3.6.8.5 [Output Section Fill], page 36), 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.

3.6.6 Output section keywords

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

`CREATE_OBJECT_SYMBOLS`

The command tells the linker to create a symbol for each input file. The name of each symbol will be the name of the corresponding input file. The section of each symbol will be the output section in which the `CREATE_OBJECT_SYMBOLS` command appears.

This is conventional for the a.out object file format. It is not normally used for any other object file format.

`CONSTRUCTORS`

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

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

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

```
__CTOR_LIST__ = .;
LONG((__CTOR_END__ - __CTOR_LIST__) / 4 - 2)
*(.ctors)
LONG(0)
__CTOR_END__ = .;
__DTOR_LIST__ = .;
LONG((__DTOR_END__ - __DTOR_LIST__) / 4 - 2)
*(.dtors)
LONG(0)
```

```
__DTOR_END__ = .;
```

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

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

3.6.7 Output section discarding

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

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

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

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

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

3.6.8 Output section attributes

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

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

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

3.6.8.1 Output section type

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

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

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

The linker normally sets the attributes of an output section based on the input sections which map into it. You can override this by using the section type. For example, in the script sample below, the '`ROM`' section is addressed at memory location '`0`' and does not need to be loaded when the program is run. The contents of the '`ROM`' section will appear in the linker output file as usual.

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

3.6.8.2 Output section LMA

Every section has a virtual address (VMA) and a load address (LMA); see Section 3.1 [Basic Script Concepts], page 21. The address expression which may appear in an output section description sets the VMA (see Section 3.6.3 [Output Section Address], page 28).

The linker will normally set the LMA equal to the VMA. You can change that by using the `AT` keyword. The expression *lma* that follows the `AT` keyword specifies the load address of the section. Alternatively, with '`AT>`*lma_region*' expression, you may specify a memory region for the section's load address. See Section 3.7 [MEMORY], page 38.

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

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

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

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

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

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

3.6.8.3 Output section region

You can assign a section to a previously defined region of memory by using '*>region*'. See Section 3.7 [MEMORY], page 38.

Here is a simple example:

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

3.6.8.4 Output section phdr

You can assign a section to a previously defined program segment by using '*:phdr*'. See Section 3.8 [PHDRS], page 39. If a section is assigned to one or more segments, then all subsequent allocated sections will be assigned to those segments as well, unless they use an explicitly *:phdr* modifier. You can use `:NONE` to tell the linker to not put the section in any segment at all.

Here is a simple example:

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

3.6.8.5 Output section fill

You can set the fill pattern for an entire section by using '*=fillexp*'. *fillexp* is an expression (see Section 3.10 [Expressions], page 43). 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 two least significant bytes of the value, repeated as necessary.

You can also change the fill value with a `FILL` command in the output section commands; see Section 3.6.5 [Output Section Data], page 32.

Here is a simple example:

```
SECTIONS { .text : { *(.text) } =0x9090 }
```

3.6.9 Overlay description

An overlay description provides an easy way to describe sections which are to be loaded as part of a single memory image but are to be run at the same memory address. At run time, some sort of overlay manager will copy the overlaid sections in and out of the runtime memory address as

required, perhaps by simply manipulating addressing bits. This approach can be useful, for example, when a certain region of memory is faster than another.

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

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

Everything is optional except `OVERLAY` (a keyword), and each section must have a name (*secname1* and *secname2* above). The section definitions within the `OVERLAY` construct are identical to those within the general `SECTIONS` contruct (see Section 3.6 [SECTIONS], page 27), except that no addresses and no memory regions may be defined for sections within an `OVERLAY`.

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

If the `NOCROSSREFS` keyword is used, and there any references among the sections, the linker will report an error. Since the sections all run at the same address, it normally does not make sense for one section to refer directly to another. See Section 3.4.4 [Miscellaneous Commands], page 25.

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

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

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

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

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

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

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

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

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

3.7 MEMORY command

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

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

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

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

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

The *attr* string is an optional list of attributes that specify whether to use a particular memory region for an input section which is not explicitly mapped in the linker script. As described in Section 3.6 [SECTIONS], page 27, if you do not specify an output section for some input section, the linker will create an output section with the same name as the input section. If you define region attributes, the linker will use them to select the memory region for the output section that it creates.

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

'R' Read-only section

'W' Read/write section

‘`X`’ Executable section

‘`A`’ Allocatable section

‘`I`’ Initialized section

‘`L`’ Same as ‘`I`’

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

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

The *origin* is an expression for the start address of the memory region. The expression must evaluate to a constant before memory allocation is performed, which means that you may not use any section relative 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 evaluate to a constant before memory allocation is performed. The keyword `LENGTH` may be abbreviated to `len` or `l`.

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

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

Once you define a memory region, you can direct the linker to place specific output sections into that memory region by using the ‘`>`*region*’ output section attribute. For example, if you have a memory region named ‘`mem`’, you would use ‘`>mem`’ in the output section definition. See Section 3.6.8.3 [Output Section Region], page 36. 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.

3.8 PHDRS Command

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

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

The linker will create reasonable program headers by default. However, in some cases, you may need to specify the program headers more precisely. You may use the `PHDRS` command for this

purpose. When the linker sees the PHDRS command in the linker script, it will not create any program headers other than the ones specified.

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

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

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

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

Certain program header types describe segments of memory which the system loader will load from the file. In the linker script, you specify the contents of these segments by placing allocatable output sections in the segments. You use the '`:`*phdr*' output section attribute to place a section in a particular segment. See Section 3.6.8.4 [Output Section Phdr], page 36.

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

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

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

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

PT_NULL (0)
: Indicates an unused program header.

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

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

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

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

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

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

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

You can specify that a segment should be loaded at a particular address in memory by using an `AT` expression. This is identical to the `AT` command used as an output section attribute (see Section 3.6.8.2 [Output Section LMA], page 35). The `AT` command for a program header overrides the output section attribute.

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

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

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

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

3.9 VERSION Command

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

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

The syntax of the `VERSION` command is simply

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

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

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

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

VERS_1.2 {
 foo2;
} VERS_1.1;

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

This example version script defines three version nodes. The first version node defined is '`VERS_1.1`'; it has no other dependencies. The script binds the symbol '`foo1`' to '`VERS_1.1`'. It reduces a number of symbols to local scope so that they are not visible outside of the shared library.

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

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

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

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

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.

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.

3.10 Expressions in Linker Scripts

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

You can use and set symbol values in expressions.

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

3.10.1 Constants

All constants are integers.

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

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

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

3.10.2 Symbol Names

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

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

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

3.10.3 The Location Counter

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

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

```
SECTIONS
{
  output :
    {
      file1(.text)
      . = . + 1000;
      file2(.text)
      . += 1000;
      file3(.text)
    } = 0x1234;
}
```

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 '`= 0x1234`' specifies what data to write in the gaps (see Section 3.6.8.5 [Output Section Fill], page 36).

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

```
SECTIONS
{
```

```
        . = 0x100
        .text: {
          *(.text)
          . = 0x200
        }
        . = 0x500
        .data: {
          *(.data)
          . += 0x600
        }
    }
```

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

3.10.4 Operators

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

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

† Prefix operators.
‡ See Section 3.5 [Assignments], page 26.

3.10.5 Evaluation

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

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

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

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

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

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

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

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

3.10.6 The Section of an Expression

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

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

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

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

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

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

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

3.10.7 Builtin Functions

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

`ABSOLUTE(`*exp*`)`

Return the absolute (non-relocatable, as opposed to non-negative) value of the expression *exp*. Primarily useful to assign an absolute value to a symbol within a section definition, where symbol values are normally section relative. See Section 3.10.6 [Expression Section], page 46.

`ADDR(`*section*`)`

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

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

`ALIGN(`*exp*`)`

Return the location counter (`.`) aligned to the next *exp* boundary. *exp* must be an expression whose value is a power of two. This is equivalent to

```
(. + exp - 1) & ~(exp - 1)
```

`ALIGN` doesn't change the value of the location counter—it just does arithmetic on it. Here is an example which aligns the output `.data` section to the next `0x2000` byte boundary after the preceding section and sets a variable within the section to the next `0x8000` boundary after the input sections:

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

The first use of `ALIGN` in this example specifies the location of a section because it is used as the optional *address* attribute of a section definition (see Section 3.6.3 [Output Section Address], page 28). The second use of `ALIGN` is used to defines the value of a symbol.

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

`BLOCK(`*exp*`)`

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

`DEFINED(`*symbol*`)`

Return 1 if *symbol* is in the linker global symbol table and is defined, 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 : . ;
    ...
  }
  ...
}
```

`LOADADDR(`*section*`)`
: Return the absolute LMA of the named *section*. This is normally the same as `ADDR`, but it may be different if the `AT` attribute is used in the output section definition (see Section 3.6.8.2 [Output Section LMA], page 35).

`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.

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

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

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

  When producing an ELF output file, if the linker script uses the `SIZEOF_HEADERS` builtin function, the linker must compute the number of program headers before it has determined all the section addresses and sizes. If the linker later discovers that it needs additional program headers, it will report an error '`not enough room for program headers`'. To avoid this error, you must avoid using the `SIZEOF_HEADERS` function, or you must rework your linker script to avoid forcing the linker to use additional program headers, or you must define the program headers yourself using the `PHDRS` command (see Section 3.8 [PHDRS], page 39).

3.11 Implicit Linker Scripts

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

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

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

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

4 Machine Dependent Features

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

4.1 `ld` and the H8/300

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

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

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

4.2 `ld` and the Intel 960 family

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

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

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

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

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

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

4.3 `ld`'s support for interworking between ARM and Thumb code

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

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

4.4 `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.

5 BFD

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

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

One minor artifact of the BFD solution which you should bear in mind is the potential for information loss. There are two places where useful information can be lost using the BFD mechanism: during conversion and during output. See Section 5.1.1 [BFD information loss], page 53.

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

5.1.1 Information Loss

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

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

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

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

5.1.2 The BFD canonical object-file format

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

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

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

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

Normal global and simple local symbols are maintained on output, so an output file (no matter its format) will retain symbols pointing to functions and to global, static, and common variables. Some symbol information is not worth retaining; in `a.out`, type information is stored in the symbol table as long symbol names. This information would be useless to most COFF debuggers; the linker has command line switches to allow users to throw it away.

There is one word of type information within the symbol, so if the format supports symbol type information within symbols (for example, COFF, IEEE, Oasys) and the

type is simple enough to fit within one word (nearly everything but aggregates), the information will be preserved.

relocation level

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

line numbers

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

6 Reporting Bugs

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

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

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

6.1 Have you found a bug?

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

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

6.2 How to report bugs

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

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

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

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

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

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

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

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, uuencoded if necessary to get them through the mail system. Making them available for anonymous FTP is not as good, but may be the only reasonable choice for large object files.

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

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

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

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

Here are some things that are not necessary:

- A description of the envelope of the bug.

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

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

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

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

- A patch for the bug.

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

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

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

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

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

Appendix A MRI Compatible Script Files

To aid users making the transition to GNU `ld` from the MRI linker, `ld` can use MRI compatible linker scripts as an alternative to the more general-purpose linker scripting language described in Chapter 3 [Scripts], page 21. 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.

Index

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

The body of this manual is set in
ptmr7t at 10.94398pt,
with headings in **ptmb7t at 10.94398pt**
and examples in `pcrr7t at 8.75519pt`.
ptmri7t at 10.94398pt and
ptmro7t at 10.94398pt
are used for emphasis.

Table of Contents