# Using ld

The GNU linker

`ld` version 2
March 1993

**Steve Chamberlain and Roland Pesch**
**Cygnus Support**

Cygnus Support
steve@cygnus.com, pesch@cygnus.com
*Using LD, the GNU linker*
Edited by Jeffrey Osier (jeffrey@cygnus.com), March 1993.

# 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 4 [BFD], page 29 for a list of formats supported on various architectures.

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.

Here is a summary of the options you can use on the `ld` command line:

```
ld [-o output ] objfile...
  [ -Aarchitecture ]  [ -b input-format ]  [ -Bstatic ]
  [ -c MRI-commandfile ]  [ -d | -dc | -dp ]
  [ -defsym symbol=expression ]
  [ -e entry ]  [ -F ]  [ -F format ]
  [ -format input-format ]  [ -g ]  [ -i ]
  [ -lar ]  [ -Lsearchdir ]  [ -M | -m ]
  [ -n | -N ]  [ -noinhibit-exec ]  [ -R filename ]
  [ -relax ]  [ -r | -Ur ]  [ -S ]  [ -s ]  [ -T commandfile ]
  [ -Ttext textorg ]  [ -Tdata dataorg ]  [ -Tbss bssorg ]
  [ -t ]  [ -u sym]  [-v]  [ -X ]  [ -x ]  [ -ysymbol ]
  [ { script } ]
```

This plethora of command-line options may seem intimidating, 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.)

The command-line options to `ld` may be specified in any order, and may be repeated at will. Repeating most options with a different argument will either have no further effect, or override prior occurrences (those further to the left on the command line) of that option.

The exceptions—which may meaningfully be used more than once—are '`-A`', '`-b`' (or its synonym '`-format`'), '`-defsym`', '`-L`', '`-l`', '`-R`', and '`-u`'.

The list of object files to be linked together, shown as *objfile*. . ., may follow, precede, or be mixed in with command-line options, except that an *objfile* argument may not be placed between an option and its argument.

Usually the linker is invoked with at least one object file, but other forms of binary input files can also be specified with '`-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`'.

Option arguments must either follow the option letter without intervening whitespace, or be given as separate arguments immediately following the option that requires them.

*objfile*. . . The object files to be linked.

`-b` *input-format*

Specify the binary format for input object files that follow this option on the command line. You don't usually need to specify this, as `ld` is 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. '`-format` *input-format*' has the same effect. See Chapter 4 [BFD], page 29.

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`. You can also define the input format from a script, using the command `TARGET`; see Section 3.6 [Other Commands], page 25.

`-Bstatic` Ignored. This option is accepted for command-line compatibility with the SunOS linker.

`-c` *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 35. Introduce MRI script files with the option '`-c`'; use the '`-T`' option to run linker scripts written in the general-purpose `ld` scripting language.

`-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 3.6 [Other Commands], page 25.

`-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 3.2.6 [Assignment: Symbol Definitions], page 13). *Note:* there should be no white space between *symbol*, the equals sign ("`=`"), and *expression*.

`-e` *entry*
Use *entry* as the explicit symbol for beginning execution of your program, rather than the default entry point. See Section 3.5 [Entry Point], page 24, for a discussion of defaults and other ways of specifying the entry point.

`-F`
`-F`*format*
Ignored. Some older linkers used this option throughout a compilation toolchain for specifying object-file format for both input and output object files. The mechanisms `ld` uses for this purpose (the '`-b`' or '`-format`' options for input files, the `TARGET` command in linker scripts for output files, the `GNUTARGET` environment variable) are more flexible, but `ld` accepts the '`-F`' option for compatibility with scripts written to call the old linker.

`-format` *input-format*

Synonym for '`-b` *input-format*'.

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

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

`-l`*ar*
Add archive file *ar* 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`*ar*`.a` for every *ar* specified.

-L*searchdir*

Add path *searchdir* to the list of paths that `ld` will search for archive libraries. You may use this option any number of times.

The paths can also be specified in a link script with the `SEARCH_DIR` command.

`-M`
`-m`
Print (to the standard output) a link map—diagnostic information about where symbols are mapped by `ld`, and information on global common storage allocation.

`-N`
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`.

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

`-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.

`-o` *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.

`-R` *filename*
: On some platforms, this 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.

`-relax`
: An option with machine dependent effects. Currently this option is only supported on the H8/300.

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

`-r`
: 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 as `-i`.

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

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

`{` *script* `}`
: You can, if you wish, include a script of linker commands directly in the command line instead of referring to it via an input file. When the character '`{`' occurs on the command line, the linker switches to interpreting the command language until the end of the list of commands is reached; the end is indicated with a closing brace '`}`'. `ld` does not recognize other command-line options while parsing the script. See Chapter 3 [Commands], page 9, for a description of the command language.

`-Tbss` *bssorg*
`-Tdata` *dataorg*
`-Ttext` *textorg*
: 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.

`-T` *commandfile*
`-T`*commandfile*
: Read link commands from the file *commandfile*. These commands completely override `ld`'s default link format (rather than adding to it); *commandfile* must specify everything necessary to describe the target format. See Chapter 3 [Commands], page 9.

  You may also include a script of link commands directly in the command line by bracketing it between '`{`' and '`}`'.

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

`-u` *sym*
: Force *sym* 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.

`-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`' *will* resolve references to constructors, unlike '`-r`'.

`-v`
: Display the version number for `ld`.

`-X`
: If '`-s`' or '`-S`' is also specified, delete only local symbols beginning with '`L`'.

`-x`
: If '`-s`' or '`-S`' is also specified, delete all local symbols, not just those beginning with '`L`'.

`-y`*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.

# 3 Command Language

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

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

You may supply a command file (also known as a link script) to the linker either explicitly through the '`-T`' option, or implicitly as an ordinary file. If the linker opens a file which it cannot recognize as a supported object or archive format, it tries to interpret the file as a command file.

You can also include a script directly on the `ld` command line, delimited by the characters '`{`' and '`}`'.

## 3.1 Linker Scripts

The `ld` command language is a collection of statements; some are simple keywords setting a particular option, some are used to select and group input files or name output files; and two statement types have a fundamental and pervasive impact on the linking process.

The most fundamental command of the `ld` command language is the `SECTIONS` command (see Section 3.4 [SECTIONS], page 18). Every meaningful command script must have a `SECTIONS` command: it specifies a "picture" of the output file's layout, in varying degrees of detail. No other command is required in all cases.

The `MEMORY` command complements `SECTIONS` by describing the available memory in the target architecture. This command is optional; if you don't use a `MEMORY` command, `ld` assumes sufficient memory is available in a contiguous block for all output. See Section 3.3 [MEMORY], page 16.

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

## 3.2 Expressions

Many useful commands involve arithmetic expressions. The syntax for expressions in the command language is identical to that of C expressions, with the following features:

- All expressions evaluated as integers and are of "long" or "unsigned long" type.
- All constants are integers.
- All of the C arithmetic operators are provided.
- You may reference, define, and create global variables.
- You may call special purpose built-in functions.

### 3.2.1 Integers

An octal integer is '`0`' followed by zero or more of the octal digits ('`01234567`').

```
_as_octal = 0157255;
```

A decimal integer starts with a non-zero digit followed by zero or more digits ('`0123456789`').

```
_as_decimal = 57005;
```

A hexadecimal integer is '`0x`' or '`0X`' followed by one or more hexadecimal digits chosen from '`0123456789abcdefABCDEF`'.

```
_as_hex = 0xdead;
```

To write a negative integer, use the prefix operator '`-`'; see Section 3.2.4 [Operators], page 12.

```
_as_neg = -57005;
```

Additionally the suffixes `K` and `M` may be used 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;
```

### 3.2.2 Symbol Names

Unless quoted, symbol names start with a letter, underscore, point or hyphen and may include any letters, underscores, digits, points, and minus signs. 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;
```

### 3.2.3 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 must always appear in an expression within a `SECTIONS` command. The `.` symbol may appear anywhere that an ordinary symbol is allowed in an expression, but its assignments have a side effect. Assigning a value to the `.` symbol will cause the location counter to be moved. This 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, `file1` is located at the beginning of the output section, then there is a 1000 byte gap. Then `file2` appears, also with a 1000 byte gap following before `file3` is loaded. The notation '`= 0x1234`' specifies what data to write in the gaps (see Section 3.4.3 [Section Options], page 23).

### 3.2.4 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 3.2.6 [Assignment], page 13.

### 3.2.5 Evaluation

The linker uses "lazy evaluation" for expressions; it only calculates an expression when absolutely necessary. The linker needs the value of the start address, and the 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 command file. However, other values (such as symbol values) are not known or needed until after storage allocation. Such values are evaluated later, when other information (such as the sizes of output sections) is available for use in the symbol assignment expression.

### 3.2.6 Assignment: Defining Symbols

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

*symbol* `=` *expression* ;
*symbol* `&=` *expression* ;
*symbol* `+=` *expression* ;
*symbol* `-=` *expression* ;
*symbol* `*=` *expression* ;
*symbol* `/=` *expression* ;

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

- Assignment may only be used at the root of an expression; '`a=b+3;`' is allowed, but '`a+b=3;`' is an error.
- You must place a trailing semicolon ("`;`") at the end of an assignment statement.

Assignment statements may appear:

- as commands in their own right in an `ld` script; or
- as independent statements within a `SECTIONS` command; or
- as part of the contents of a section definition in a `SECTIONS` command.

The first two cases are equivalent in effect—both define a symbol with an absolute address. The last case defines a symbol whose address is relative to a particular section (see Section 3.4 [SECTIONS], page 18).

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

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

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

The linker tries to put off the evaluation of an assignment until all the terms in the source expression are known (see Section 3.2.5 [Evaluation], page 12). For instance, 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 *dot*, '`.`' must be evaluated during 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 :
                { ...
                }
... }
```

will cause the error message "`Non constant expression for initial address`".

### 3.2.7 Built-In Functions

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

`ABSOLUTE(`*exp*`)`

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

`ADDR(`*section*`)`

Return the absolute address 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 result of the current 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. As an example, to align the output `.data` section to the next `0x2000` byte boundary after the preceding section and to set a variable within the section to the next `0x8000` boundary after the input sections:

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

The first use of `ALIGN` in this example specifies the location of a section because it is used as the optional *start* attribute of a section definition (see Section 3.4.3 [Section Options], page 23). The second use simply defines the value of a variable.

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

`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 command-file fragment shows how to set a global symbol `begin` to the first location in the `.text` section—but if a symbol called `begin` already existed, its value is preserved:

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

`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. 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. You can use this number as the start address of the first section, if you choose, to facilitate paging.

## 3.3 MEMORY Command

The linker's default configuration permits allocation of all available memory. You can override this configuration by using the `MEMORY` command. The `MEMORY` command describes the location and size of blocks of memory in the target. By using it carefully, you can describe which memory regions may be used by the linker, and which memory regions it must avoid. The linker does not shuffle sections to fit into the available regions, but does move the requested sections into the correct regions and issue errors when the regions become too full.

The command files 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
       ...
      }
```

| | |
|---|---|
| *name* | is a name used internally by the linker to refer to the region. Any symbol name may be used. The region names are stored in a separate name space, and will not conflict with symbols, file names or section names. Use distinct names to specify multiple regions. |
| (*attr*) | is an optional list of attributes, permitted for compatibility with the AT&T linker but not used by `ld` beyond checking that the attribute list is valid. Valid attribute lists must be made up of the characters "`LIRWX`". If you omit the attribute list, you may omit the parentheses around it as well. |
| *origin* | is the start address of the region in physical memory. It is an expression that must evaluate to a constant before memory allocation is performed. The keyword `ORIGIN` may be abbreviated to `org` or `o`. |
| *len* | is the size in bytes of the region (an expression). The keyword `LENGTH` may be abbreviated to `len` or `l`. |

For example, to specify that memory has two regions available for allocation—one starting at 0 for 256 kilobytes, and the other starting at `0x40000000` for four megabytes:

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

Once you have defined a region of memory named *mem*, you can direct specific output sections there by using a command ending in '`>`*mem*' within the `SECTIONS` command (see Section 3.4.3 [Section Options], page 23). If the combined output sections directed to a region are too big for the region, the linker will issue an error message.

## 3.4 SECTIONS Command

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

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

- define the entry point;
- assign a value to a symbol;
- describe the placement of a named output section, and what input sections make it up.

The first two possibilities—defining the entry point, and defining symbols—can also be done outside the `SECTIONS` command: see Section 3.5 [Entry Point], page 24, see Section 3.2.6 [Assignment], page 13. They are permitted here as well for your convenience in reading the script, so that symbols or the entry point can be defined at meaningful points in your output-file layout.

When no `SECTIONS` command is specified, the default action of the linker is to 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.

### 3.4.1 Section Definitions

The most frequently used statement in the `SECTIONS` command is the *section definition*, which you can use to specify the properties of an output section: its location, alignment, contents, fill pattern, and target memory region. Most of these specifications are optional; the simplest form of a section definition is

```
SECTIONS { ...
secname : {
                 contents
                }
... }
```

*secname* is the name of the output section, and *contents* a specification of what goes there—for example, a list of input files or sections of input files. As you might assume, the whitespace shown is optional. You do need the colon '`:`' and the braces '`{}`', however.

*secname* 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 characters, but any name which does not conform to the standard `ld` symbol name syntax must be quoted. See Section 3.2.2 [Symbol Names], page 11.

### 3.4.2 Section Contents

In a section definition, you can specify the contents of an output section by listing particular object files, by listing particular input-file sections, or by a combination of the two. You can also place arbitrary data in the section, and define symbols relative to the beginning of the section.

The *contents* of a section definition may include any of the following kinds of statement. You can include as many of these as you like in a single section definition, separated from one another by whitespace.

*filename*
: You may simply name a particular input file to be placed in the current output section; *all* sections from that file are placed in the current section definition. To specify a list of particular files by name:

```
.data : { afile.o bfile.o cfile.o }
```

The example also illustrates that multiple statements can be included in the contents of a section definition, since each file name is a separate statement.

If the file name has already been mentioned in another section definition, with an explicit section name list, then only those sections which have not yet been allocated are used.

*filename*`(` *section* `)`
*filename*`(` *section*`,` *section*`,` ... `)`
*filename*`(` *section section* ... `)`

: You can name one or more sections from your input files, for insertion in the current output section. If you wish to specify a list of input-file sections inside the parentheses, you may separate the section names by either commas or whitespace.

`* (`*section*`)`
`* (`*section*`,` *section*`,` ...`)`
`* (`*section section* ...

: Instead of explicitly naming particular input files in a link control script, you can refer to *all* files from the `ld` command line: use '`*`' instead of a particular file name before the parenthesized input-file section list.

For example, to copy sections `1` through `4` from an Oasys file into the `.text` section of an `a.out` file, and sections `13` and `14` into the `.data` section:

```
SECTIONS {
        .text :{
                *("1" "2" "3" "4")
        }

        .data :{
                *("13" "14")
        }
}
```

If you have already explicitly included some files by name, '`*`' refers to all *remaining* files—those whose places in the output file have not yet been defined.

`[` *section* `]`
`[` *section*`,` *section*`,` ... `]`
`[` *section section* ... `]`

This is an alternate notation to specify named sections from all unallocated input files; its effect is exactly the same as that of '`* (`*section*...`)`'

*filename*`( COMMON )`
`( COMMON )`

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

For example, the following command script arranges the output file into three consecutive sections, named `.text`, `.data`, and `.bss`, taking the input for each from the correspondingly named sections of all the input files:

```
SECTIONS {
  .text : { *(.text) }
  .data : { *(.data) }
  .bss :  { *(.bss)  *(COMMON) }
}
```

The following example reads all of the sections from file `all.o` and places 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)
                }
}
```

There are still more kinds of statements permitted in the contents of output section definitions. The foregoing statements permitted you to arrange, in your output file, data originating from your input files. You can also place data directly in an output section from the link command script. Most of these additional statements involve expressions; see Section 3.2 [Expressions], page 10. Although these statements are shown separately here for ease of presentation, no such segregation is needed within a section definition in the `SECTIONS` command; you can intermix them freely with any of the statements we've just described.

`CREATE_OBJECT_SYMBOLS`

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

```
SECTIONS {
        .text 0x2020 :
                 {
                CREATE_OBJECT_SYMBOLS
                *(.text)
                _etext = ALIGN(0x2000);
                }
        ...
}
```

If `objsym` is a file containing this script, and `a.o`, `b.o`, `c.o`, and `d.o` are four input files with contents like the following—

```
/* a.c */

afunction() { }
int adata=1;
int abss;
```

'`ld -M sample a.o b.o c.o d.o`' would create a map like this, containing symbols matching the object file names:

```
00000000 A __DYNAMIC
00004020 B _abss
00004000 D _adata
00002020 T _afunction
00004024 B _bbss
00004008 D _bdata
00002038 T _bfunction
00004028 B _cbss
00004010 D _cdata
00002050 T _cfunction
0000402c B _dbss
00004018 D _ddata
00002068 T _dfunction
00004020 D _edata
00004030 B _end
00004000 T _etext
00002020 t a.o
00002038 t b.o
00002050 t c.o
00002068 t d.o
```

*symbol* = *expression* ;
*symbol f*= *expression* ;

*symbol* is any symbol name (see Section 3.2.2 [Symbols], page 11). "*f*=" refers to any of the operators `&= += -= *= /=` which combine arithmetic and assignment.

When you assign a value to a symbol within a particular section definition, the value is relative to the beginning of the section (see Section 3.2.6 [Assignment], page 13). If you write

```
SECTIONS {
  abs = 14 ;
  ...
  .data : { ... rel = 14 ; ... }
  abs2 = 14 + ADDR(.data);
  ...
}
```

`abs` and `rel` do not have the same value; `rel` has the same value as `abs2`.

`BYTE(`*expression*`)`
`SHORT(`*expression*`)`
`LONG(`*expression*`)`

By including one of these three statements in a section definition, you can explicitly place one, two, or four bytes (respectively) at the current address of that section.

Multiple-byte quantities are represented in whatever byte order is appropriate for the output file format (see Chapter 4 [BFD], page 29).

`FILL(`*expression*`)`

Specifies the "fill pattern" for the current section. Any otherwise unspecified regions of memory within the section (for example, regions you skip over by assigning a new value to the location counter '`.`') are filled with the two least significant bytes from the *expression* argument. A `FILL` statement covers memory locations *after* the point 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.

## 3.4.3 Optional Section Attributes

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

```
SECTIONS {
...
secname start BLOCK(align) (NOLOAD) : { contents } =fill >region
...
}
```

*secname* and *contents* are required. See Section 3.4.1 [Section Definition], page 18, and see Section 3.4.2 [Section Contents], page 19 for details on *contents*. The remaining elements—*start*, `BLOCK(`*align*`)`, `(NOLOAD) =`*fill*, and `>`*region*—are all optional.

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

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

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

`(NOLOAD)`
: Use '`(NOLOAD)`' to prevent a section from being loaded into memory each time it is accessed. For example, in the script sample below, the `ROM` segment is addressed at memory location '`0`' and does not need to be loaded into each object file:

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

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

`>`*region*
: Assign this section to a previously defined region of memory. See Section 3.3 [MEMORY], page 16.

## 3.5 The Entry Point

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

`ENTRY(`*symbol*`)`

Like symbol assignments, the `ENTRY` command may be placed either as an independent command in the command file, or among the section definitions within the `SECTIONS` command—whatever makes the most sense for your layout.

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

- the '`-e`' *entry* command-line option;
- the `ENTRY(`*symbol* command in a linker control script;

- the value of the symbol `start`, if present;
- the value of the symbol `_main`, if present;
- the address of the first byte of the `.text` section, if present;
- The address `0`.

For example, you can use these rules to generate an entry point with an assignment statement: if no symbol `start` is defined within your input files, you can simply define it, assigning it an appropriate value—

```
start = 0x2020;
```

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

```
start = other_symbol ;
```

## 3.6 Other Commands

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

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

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

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

Use this command to include binary input files in the link, without including them in a particular section definition. Files specified this way are treated identically to object files listed on the command line.

`OUTPUT (` *filename* `)`

Use this command to name the link output file *filename*. The effect of `OUTPUT(`*filename*`)` is identical to the effect of '`-o` *filename*', and whichever is encountered last will control the name actually used to name the output file. In particular, you can use this command to supply a default output-file name other than `a.out`.

`OUTPUT_ARCH (` *bfdname* `)`

Specify a particular output machine architecture, with one of the names used by the BFD back-end routines (see Chapter 4 [BFD], page 29). This command is often unnecessary; the architecture is most often set implicitly by either the system BFD configuration or as a side effect of the `OUTPUT_FORMAT` command.

`OUTPUT_FORMAT (` *bfdname* `)`

Specify a particular output format, with one of the names used by the BFD back-end routines (see Chapter 4 [BFD], page 29). This selection will only affect the output file; the related command `TARGET` affects primarily input files.

`SEARCH_DIR (` *path* `)`

Add *path* to the list of paths where `ld` looks for archive libraries. `SEARCH_DIR(`*path*`)` has the same effect as '`-L`*path*' on the command line.

`STARTUP (` *filename* `)`

Ensure that *filename* is the first input file used in the link process.

`TARGET (` *format* `)`

Change the input-file object code format (like the command-line option '`-b`' or its synonym '`-format`'). The argument *format* is one of the strings used by BFD to name binary formats. In the current `ld` implementation, if `TARGET` is specified but `OUTPUT_FORMAT` is not, the last `TARGET` argument is also used as the default format for the `ld` output file. See Chapter 4 [BFD], page 29.

If you don't use the `TARGET` command, `ld` uses the value of the environment variable `GNUTARGET`, if available, to select the output file format. If that variable is also absent, `ld` uses the default format configured for your machine in the BFD libraries.

# 4 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. You can use `objdump -i` (see section "objdump" in *The GNU Binary Utilities*) to list all the formats available for each architecture under BFD. This was the list of formats, and of architectures supported for each format, as of the time this manual was prepared:

```
BFD header file version 0.18
a.out-i386
 (header big endian, data big endian)
  m68k:68020
  a29k
  sparc
  i386
a.out-sunos-big
 (header big endian, data big endian)
  m68k:68020
  a29k
  sparc
  i386
b.out.big
 (header big endian, data little endian)
  i960:core
b.out.little
 (header little endian, data little endian)
  i960:core
coff-a29k-big
 (header big endian, data big endian)
  a29k
coff-h8300
 (header big endian, data big endian)
  H8/300
coff-i386
 (header little endian, data little endian)
  i386
coff-Intel-big
 (header big endian, data little endian)
  i960:core
coff-Intel-little
 (header little endian, data little endian)
  i960:core
coff-m68k
 (header big endian, data big endian)
  m68k:68020
```

```
coff-m88kbcs
 (header big endian, data big endian)
  m88k:88100
ecoff-bigmips
 (header big endian, data big endian)
  mips
ecoff-littlemips
 (header little endian, data little endian)
  mips
elf-big
 (header big endian, data big endian)
  m68k:68020
  vax
  i960:core
  a29k
  sparc
  mips
  i386
  m88k:88100
  H8/300
  rs6000:6000
elf-little
 (header little endian, data little endian)
  m68k:68020
  vax
  i960:core
  a29k
  sparc
  mips
  i386
  m88k:88100
  H8/300
  rs6000:6000
ieee
 (header big endian, data big endian)
  m68k:68020
  vax
  i960:core
  a29k
  sparc
  mips
  i386
  m88k:88100
  H8/300
  rs6000:6000
```

```
srec
 (header big endian, data big endian)
  m68k:68020
  vax
  i960:core
  a29k
  sparc
  mips
  i386
  m88k:88100
  H8/300
  rs6000:6000
```

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 4.2 [BFD information loss], page 32.

## 4.1 How it works: an outline of BFD

When an object file is opened, BFD subroutines automatically determine the format of the input object file, and 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 the memory pointer to the BFD back end routine which reads and converts the table into a canonical form. The linker then operates upon the common form. When the link is finished and the linker writes the symbol table of the output file, another BFD back end routine is called which takes the newly created symbol table and converts it into the chosen output format.

## 4.2 Information Loss

*Information can be lost during output.* The output formats supported by BFD do not provide identical facilities, and information which may 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 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 using the linker to read one format and write 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 linker. At the same time, the back end saves away any information which would 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 big endian COFF to little endian COFF, or from `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.

## 4.3 Mechanism

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 on target machine architecture, particular implementation, and format type are stored on a per-file basis. Other information includes a demand pagable 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 pagable 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 linked with one another.

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

*symbols* Each symbol contains a pointer to the object file which originally defined it, its name, its value, and various option bits. When a BFD back end reads in a symbol table, the back end 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 and may be thrown away with appropriate command line switches. (The GNU debugger `gdb` does support `a.out` style debugging information in COFF).

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 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).

# 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 [Command Language], page 9. 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.

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.

`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

The body of this manual is set in
cmr10 at 10.95pt,
with headings in **cmb10 at 10.95pt**
and examples in `cmtt10 at 10.95pt`.
*cmti10 at 10.95pt* and
*cmsl10 at 10.95pt*
are used for emphasis.

# Table of Contents