Go to the first, previous, next, last section, table of contents.

Command Language

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

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

You may supply a command file (also known as a 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 reports an error.

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 Specifying Output Sections). 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 Memory Layout.

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

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.

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 Operators.

```
_as_neg = -57005;
```

Additionally the suffixes `K' and `M' may be used to scale a constant by respectively. For example, the following all refer to the same quantity:

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

Symbol Names

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

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 Optional Section Attributes).

@vfill

Operators

The linker recognizes the standard C set of arithmetic operators, with the standard bindings and precedence levels: { @obeylines@parskip=0pt@parindent=0pt @dag@quad Prefix operators. @ddag@quad See section Assignment: Defining Symbols. }

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.

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 Specifying Output Sections).

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 Evaluation). 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".

In some cases, it is desirable for a linker script to define a symbol only if it is referenced, and only if it 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*).

Arithmetic 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 Optional Section Attributes). 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.

Memory Layout

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.

A command file 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 (but not, for example, `ORG').

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 Optional Section Attributes). If the combined output sections directed to a region are too big for the region, the linker will issue an error message.

Specifying Output Sections

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

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

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

You can also use the first two operations--defining the entry point and defining symbols--outside the SECTIONS command: see section The Entry Point, and see section Assignment: Defining Symbols. They are permitted here as well for your convenience in reading the script, so that symbols and the entry point can be defined at meaningful points in your output-file layout.

If you do not use a SECTIONS command, the linker places 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.

- Section Definition: Section Definitions
- Section Placement: Section Placement
- Section Data Expressions: Section Data Expressions
- Section Options: Optional Section Attributes

Section Definitions

The most frequently used statement in the `SECTIONS` command is the **section definition**, which specifies 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 (see section Section Placement). 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 of characters, but any name which does not conform to the standard `ld` symbol name syntax must be quoted. See section Symbol Names.

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

Section Placement

In a section definition, you can specify the contents of an output section by listing particular input 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. 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. 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.

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. 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. 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")
  }
}
```

`[*section* ...]' used to be accepted as an alternate way to specify named sections from all unallocated input files. Because some operating systems (VMS) allow brackets in file names, that notation is no longer supported.

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)
    }
}
```

Section Data Expressions

The foregoing statements 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 Expressions. 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 that 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 sample.ld 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 -T sample.ld 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 Symbol Names). "*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 Assignment: Defining Symbols). 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*)
QUAD(*expression*)

By including one of these four statements in a section definition, you can explicitly place one, two, four, or eight bytes (respectively) at the current address of that section. QUAD is only supported when using a 64 bit host or target. Multiple-byte quantities are represented in whatever byte order is appropriate for the output file format (see section BFD).

FILL(*expression*)

Specify 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.

Optional Section Attributes

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

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

secname and *contents* are required. See section Section Definitions, and see section Section Placement for details on *contents*. The remaining elements---*start*, BLOCK(*align*), (NOLOAD), AT (*ldadr*), >*region*, and =*fill*---are all optional.

start

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

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

BLOCK(*align*)

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

(NOLOAD)

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)  : { ... }
  ...
}
```

AT (*ldadr*)

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

```
SECTIONS
  {
  .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 C programs, usually `crt0`) for use with a ROM generated this way has to include something like the following, to copy the initialized data from the ROM image to its runtime address:

```
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;
```

>*region*

Assign this section to a previously defined region of memory. See section Memory Layout.

`=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.

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 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 ;
```

Option 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.

`CONSTRUCTORS`

This command ties up C++ style constructor and destructor records. The details of the constructor representation vary from one object format to another, but usually lists of constructors and destructors appear as special sections. The `CONSTRUCTORS` command specifies where the linker is to place the data from these sections, relative to the rest

of the linked output. Constructor data is marked by the symbol `__CTOR_LIST__` at the start, and `__CTOR_LIST_END` at the end; destructor data is bracketed similarly, between `__DTOR_LIST__` and `__DTOR_LIST_END`. (The compiler must arrange to actually run this code; GNU C++ calls constructors from a subroutine `__main`, which it inserts automatically into the startup code for `main`, and destructors from `_exit`.)

`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. Specify the full name for each *file*, including `.a' if required. `ld` searches for each *file* through the archive-library search path, just as for files you specify on the command line. See the description of `-L' in @xref{Options,,Command Line Options}. If you use `-l*file*', `ld` will transform the name to `lib`*file*`.a` as with the command line argument `-l'.

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

This 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 @xref{Options,,Command Line Options}.

`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*', which overrides it. 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 section BFD). 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*` )`

When `ld` is configured to support multiple object code formats, you can use this command to specify a particular output format. *bfdname* is one of the names used by the BFD back-end routines (see section BFD). The effect is identical to the effect of the `-oformat' command-line option. This selection affects only 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*` )`

When `ld` is configured to support multiple object code formats, you can use this command to 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. 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 section BFD. 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.

Go to the first, previous, next, last section, table of contents.