
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 Implicit Linker Scripts.

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.

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.

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 in the above example sets 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.

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.

Simple Linker Script Commands

In this section we describe the simple linker script commands.

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.

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 Command Line Options. 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 Command Line Options.

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 Command Line Options). If both are used, the command line option takes precedence. You can use the OUTPUT command to define a default name for the output file other than the usual default of `a.out'.

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

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

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 section BFD). Using OUTPUT_FORMAT(*bfdname*) is exactly like using `-oformat *bfdname*' on the command line (see section Command Line Options). If both are used, the command line option takes precedence. You can use OUTPUT_FORMAT with three arguments to use different formats based on the `-EB' and `-EL' command line

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

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

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

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

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 section BFD). You can see the architecture of an object file by using the `objdump` program with the `-f' option.

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.

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 Expressions in Linker Scripts.

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 The Section of an Expression.

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.

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.

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 Setting the entry point)
- a symbol assignment (see section Assigning Values to Symbols)
- 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.

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

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] [: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 Assigning Values to Symbols)
- an input section description (see section Input section description)
- data values to include directly (see section Output section data)
- a special output section keyword (see section Output section keywords)

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 Output section discarding.

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 Expressions in Linker Scripts. 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.

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.

- Input Section Basics: Input section basics
- Input Section Wildcards: Input section wildcard patterns
- Input Section Common: Input section for common symbols
- Input Section Keep: Input section and garbage collection
- Input Section Example: Input section example

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 Input section wildcard patterns).

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.

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.

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) }
}
```

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)'.

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))'.

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)
    }
}
```

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 Expressions in Linker Scripts). 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.

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 Output section fill), 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.

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.

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.

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] [:phdr :phdr ...] [=fillexp]
```

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

- Output Section Type: Output section type
- Output Section LMA: Output section LMA
- Output Section Region: Output section region
- Output Section Phdr: Output section phdr
- Output Section Fill: Output section fill

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

Output section LMA

Every section has a virtual address (VMA) and a load address (LMA); see section Basic Linker Script Concepts. The address expression which may appear in an output section description sets the VMA (see section Output section address).

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.

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;
```

Output section region

You can assign a section to a previously defined region of memory by using `` `>region' ``. See section MEMORY command.

Here is a simple example:

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

Output section phdr

You can assign a section to a previously defined program segment by using `` `:phdr' ``. See section PHDRS Command. 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 }
```

Output section fill

You can set the fill pattern for an entire section by using `` `=fillexp' ``. *fillexp* is an expression (see section Expressions in Linker Scripts). 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 Output section data.

Here is a simple example:

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

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 SECTIONS command), 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 Other linker script commands.

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));
```

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 SECTIONS command, 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 Output section region. If no address was specified for the output section, the linker will set the address to the next available address within the memory region. If the combined output sections directed to a memory region are too large for the region, the linker will issue an error message.

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 Output section phdr.

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

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

You may use the FILEHDR and PHDRS keywords 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 Output section LMA). The AT command for a program header overrides the output section attribute.

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

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

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

SECTIONS
```

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

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.

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.

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 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 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.

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 Output section fill).

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 Assigning Values to Symbols. }

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'.

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.

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 The Section of an Expression.

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 Output section address). The second use of `ALIGN` is used to defines the value of a symbol. The builtin function `NEXT` is closely related to `ALIGN`.

`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 Output section LMA).

`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 PHDRS Command).

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.

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