Command Language

Go to the , , , section, .

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 linker script) to the linker either explicitly through the `` `-T' `` option, or implicitly as an ordinary file. Normally you should use the `` `-T' `` option. An implicit linker script should only be used when you want to augment, rather than replace, the default linker script; typically an implicit linker script would consist only of `INPUT` or `GROUP` commands.

If the linker opens a file which it cannot recognize as a supported object or archive format, nor as a linker script, 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;
        }
    ... }
```

`LOADADDR(`*`section`*`)`

Return the absolute load address of the named *section*. This is normally the same as `ADDR`, but it may be different if the `AT` keyword is used in the section definition (see section Optional Section Attributes).

`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.

`MAX(`*`exp1, exp2`*`)`
: Returns the maximum of *exp1* and *exp2*.

`MIN(`*`exp1, exp2`*`)`
: Returns the minimum of *exp1* and *exp2*.

Semicolons

Semicolons ("`;`") are required in the following places. In all other places they can appear for aesthetic reasons but are otherwise ignored.

`Assignment`
: Semicolons must appear at the end of assignment expressions. See section Assignment: Defining Symbols

`PHDRS`
: Semicolons must appear at the end of a `PHDRS` statement. See section ELF Program Headers

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 that specify whether to use a particular memory to place sections that are not listed in the linker script. Valid attribute lists must be made up of the characters "`ALIRWX`" that match section attributes. If you omit the attribute list, you may omit the parentheses around it as well. The attributes currently supported are:

`Letter'	Section Attribute
`R'	Read-only sections.
`W'	Read/write sections.
`X'	Sections containing executable code.
`A'	Allocated sections.
`I'	Initialized sections.
`L'	Same as `I`.
`!'	Invert the sense of any of the following attributes.

`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. The `rom` memory region will get all sections without an explicit memory register that are either read-only or contain code, while the `ram` memory region will get the sections.

```
MEMORY
  {
  rom (rx)  : ORIGIN = 0, LENGTH = 256K
  ram (!rx) : 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 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 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). The whitespace around *secname* is required, so that the section name is unambiguous. The other 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 special *secname* `` `/DISCARD/' `` may be used to discard input sections. Any sections which are assigned to an output section named `` `/DISCARD/' `` are not included in the final link output.

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, separate the section names with 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.

In any place where you may use a specific file or section name, you may also use a wildcard pattern. The linker handles wildcards much as the Unix shell does. A `` `*' `` character matches any number of characters. A `` `?' `` character matches any single character. The sequence `` `[ ``*`chars`*`]'` will match 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. A `` `\' `` character may be used to quote 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. In a section name, the wildcard characters will match a `` `/' `` character.

Wildcards only match files which are explicitly specified on the command line. The linker does not search directories to expand wildcards. However, if you specify a simple file name--a name with no wildcard characters--in a linker script, and the file name is not also specified on the command line, the linker will attempt to open the file as though it appeared on the command line.

In the following example, the 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)
    }
}
```

This example shows how wildcard patterns might be used to partition files. All `.text` sections are placed in `.text`, and all `.bss` sections are placed in `.bss`. For all files beginning with an upper case character, the `.data` section is placed into `.DATA`; for all other files, the `.data` section is placed into `.data`.

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

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`*`)`

`SQUAD(`*`expression`*`)`

By including one of these four statements in a section definition, you can explicitly place one, two, four, eight unsigned, or eight signed bytes (respectively) at the current address of that section. When using a 64 bit host or target, `QUAD` and `SQUAD` are the same. When both host and target are 32 bits, `QUAD` uses an unsigned 32 bit value, and `SQUAD` sign extends the value. Both will use the correct endianness when writing out the value. 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 :phdr =fill
...
}
```

secname and *contents* are required. See section Section Definitions, and section Section Placement, for details on *contents*. The remaining elements---*start*, `BLOCK(`*`align`*`)`, `(NOLOAD)`, `AT ( `*`ldadr`*` )`, `>`*`region`*, `:`*`phdr`*, 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)`

The `` `(NOLOAD)' `` directive will mark a section to not be loaded at run time. The linker will process the section normally, but will mark it so that a program loader will not load it into memory. 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)  : { ... }
  ...
}
```

`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.

`:`*`phdr`*

Assign this section to a segment described by a program header. See section ELF Program Headers. 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. To prevent a section from being assigned to a segment when it would normally default to one, use `:NONE`.

`=`*`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.

Overlays

The `OVERLAY` command 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.

The `OVERLAY` command is used within a `SECTIONS` command. It appears as follows:

```
OVERLAY start : [ NOCROSSREFS ] AT ( ldaddr )
 {
   secname1 { contents } :phdr =fill
   secname2 { contents } :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 Specifying Output Sections), 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 `.`).

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 Option 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 `.` 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 (.
  .text1 0x1000 : AT (0x4000 + SIZEOF (.text0)) { o
  __load_start_text1 = LOADADDR (.text1);
  __load_stop_text1 = LOADADDR (.text1) + SIZEOF (.
  . = 0x1000 + MAX (SIZEOF (.text0), SIZEOF (.text1
```

ELF Program Headers

The ELF object file format uses *program headers*, which are read by the system loader and describe how the program should be loaded into memory. These program headers must be set correctly in order to run the program on a native ELF system. The linker will create reasonable program headers by default. However, in some cases, it is desirable to specify the program headers more precisely; the PHDRS command may be used for this purpose. When the PHDRS command is used, the linker will not generate any program headers itself.

The PHDRS command is only meaningful when generating an ELF output file. It is ignored in other cases. This manual does not describe the details of how the system loader interprets program headers; for more information, see the ELF ABI. The program headers of an ELF file may be displayed using the `` `-p' `` option of the `objdump` command.

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 does not get put into the output file.

Certain program header types describe segments of memory which are loaded from the file by the system loader. In the linker script, the contents of these segments are specified by directing allocated output sections to be placed in the segment. To do this, the command describing the output section in the `SECTIONS` command should use `` `:name' ``, where *name* is the name of the program header as it appears in the `PHDRS` command. See section Optional Section Attributes.

It is normal for certain sections to appear in more than one segment. This merely implies that one segment of memory contains another. This is specified by repeating ``:*name*'`, using it once for each program header in which the section is to appear.

If a section is placed in one or more segments using ``:*name*'`, then all subsequent allocated sections which do not specify ``:*name*'` are placed in the same segments. This is for convenience, since generally a whole set of contiguous sections will be placed in a single segment. To prevent a section from being assigned to a segment when it would normally default to one, use `:NONE`.

The `FILEHDR` and `PHDRS` keywords which may appear after the program header type also indicate contents of the segment of memory. 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.

It is possible to specify that a segment should be loaded at a particular address in memory. This is done using an `AT` expression. This is identical to the `AT` command used in the `SECTIONS` command (see section Optional Section Attributes). Using the `AT` command for a program header overrides any information in the `SECTIONS` command.

Normally the segment flags are set based on the sections. The `FLAGS` keyword may be used 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 the use 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
  ...
}
```

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 ;
```

Version Script

The linker command script includes a command specifically for specifying a version script, and is only meaningful for ELF platforms that support shared libraries. A version script can be build directly into the linker script that you are using, or you can supply the version script as just another input file to the linker at the time that you link. The command script syntax is:

```
VERSION { version script contents }
```

The version script can also be specified to the linker by means of the `--version-script' linker command line option. Version scripts are only meaningful when creating shared libraries.

The format of the version script itself is identical to that used by Sun's linker in Solaris 2.5. Versioning is done by defining a tree of version nodes with the names and interdependencies specified in the version script. The version script can specify which symbols are bound to which version nodes, and it 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;
```

In this example, three version nodes are defined. `VERS_1.1' is the first version node defined, and has no other dependencies. The symbol `foo1' is bound to this version node, and a number of symbols that have appeared within various object files are reduced in scope to local so that they are not visible outside of the shared library.

Next, the node `VERS_1.2' is defined. It depends upon `VERS_1.1'. The symbol `foo2' is bound to this version node.

Finally, the node `VERS_2.0' is defined. It depends upon `VERS_1.2'. The symbols `bar1' and `bar2' are bound to this version node.

Symbols defined in the library which aren't specifically bound to a version node are effectively bound to an unspecified base version of the library. It is possible to bind all otherwise unspecified symbols to a given version node using `global: *' somewhere in the version script.

Lexically 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. This can be done by putting something like:

```
__asm__(".symver original_foo,foo@VERS_1.1");
```

in the C source file. This renamed 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 an incompatible change to an interface can take place without increasing the major version number of the shared library, while still allowing applications linked against the old interface to continue to function.

This can only be accomplished by using multiple `` `.symver' `` directives in the assembler. An example of this would be:

```
__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. This can be accomplished with the `` `foo@@VERS_2.0' `` type of `` `.symver' `` directive. Only one version of a symbol can be declared '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.

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`

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 where this information should be placed. 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++ calls constructors from a subroutine `__main`; a call to `__main` is automatically inserted into the startup code for `main`. GNU C++ runs destructors either by using `atexit`, or directly from the function `exit`. For object file formats such as `COFF` or `ELF` which support multiple sections, 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 -
        *(.ctors)
        LONG(0)
        __CTOR_END__ = .;
        __DTOR_LIST__ = .;
        LONG((__DTOR_END__ - __DTOR_LIST__) / 4 -
        *(.dtors)
        LONG(0)
        __DTOR_END__ = .;
```

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.

`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' ``).

`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`* `... )`

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 section Command Line Options. If you use `` `-lfile' ``, `ld` will transform the name to `libfile.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 section 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 `` `-Lpath' `` 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.

`NOCROSSREFS ( section section ... )`

This command may be used to tell `ld` to issue an error about any references among certain sections. In certain types of programs, particularly on embedded systems, 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 section names. If `ld` detects any cross references between the sections, it reports an error and returns a non-zero exit status. The `NOCROSSREFS` command uses output section names, defined in the `SECTIONS` command. It does not use the names of input sections.