Mapfile Option 7

Introduction

The link-editor automatically and intelligently maps input sections from relocatable objects to segments within the output file object. The `-M` option with an associated `mapfile` allows you to change the default mapping provided by the link-editor.

In particular, this `mapfile` option allows you to:

- Declare segments and specify values for segment attributes such as segment type, permissions, addresses, length, and alignment.
- Control mapping of input sections to segments by specifying the attribute values necessary in a section to map to a specific segment (the attributes are section name, section type, and permissions) and by specifying which object file(s) the input sections should be taken from, if necessary.
- Declare a global-absolute symbol that is assigned a value equal to the size of a specified segment (by the link-editor) and that can be referenced from object files.

The mapfile option allows users of *ifiles* (an option previously available to `ld(1)` that used link-editor command language directives) to convert to `mapfiles`. All other facilities previously available for *ifiles*, other than those mentioned above, are not available with the mapfile option.

7

Note – When using the mapfile option, be aware that you can easily create `a.out` files that do not execute. The link-editor knows how to produce a correct `a.out` without the use of the mapfile option. The mapfile option is intended for system programming use, not application programming use.

Using the Mapfile Option

To use the mapfile option, you must:

- Enter the mapfile directives into a file, for example `mapfile`
- Supply the following option on the `ld(1)` command line:

```
-M mapfile
```

If the `mapfile` is not in your current directory, include the full path name; no default search path exists.

Mapfile Structure and Syntax

You can enter four basic types of directives into a `mapfile`:

- Segment declarations.
- Mapping directives.
- Size-symbol declarations.
- File control directives.

Each directive can span more than one line and can have any amount of white space (including new-lines) as long as it is followed by a semicolon. You can enter zero or more directives in a `mapfile`. (Entering zero directives causes the link-editor to ignore the `mapfile` and use its own defaults.)

Typically, segment declarations are followed by mapping directives, that is, you declare a segment and then define the criteria by which a section becomes part of that segment. If you enter a mapping directive or size-symbol declaration without first declaring the segment to which you are mapping (except for built-in segments, explained later), the segment is given default attributes as explained below. Such segment is then an “implicitly declared segment.”

Size-symbol declarations, and file control directives can appear anywhere in a `mapfile`.

The following sections describe each directive type. For all syntax discussions, the following notations apply:

- All entries in `constant width`, all colons, semicolons, equal signs, and at `(@)` signs are typed in literally.
- All entries in italics are substitutable.
- { ... }* means “zero or more.”
- { ... }+ means “one or more.”
- [...] means “optional.”
- *section_name*s and *segment_name*s follow the same rules as C identifiers where a period (.) is treated as a letter (for example, *.bss* is a legal name).
- *section_name*s, *segment_name*s, *file_name*s, and *symbol_name*s are case sensitive; everything else is not case sensitive.
- Spaces (or new-lines) may appear anywhere except before a *number* or in the middle of a name or value.
- Comments beginning with # and ending at a new-line may appear anywhere that a space may appear.

Segment Declarations

A segment declaration creates a new segment in the `a.out` or changes the attribute values of an existing segment. (An existing segment is one that you previously defined or one of the three built-in segments described below.)

A segment declaration has the following syntax:

```
segment_name = {segment_attribute_value}*;
```

For each *segment_name*, you can specify any number of *segment_attribute_value*s in any order, each separated by a space. (Only one attribute value is allowed for each segment attribute.) The segment attributes and their valid values are as follows:

Table 7-1 Mapfile Segment Attributes

Attribute	Value
segment_type	LOAD NOTE
segment_flags	`?` [R] [W] [X] [O]
virtual_address	V*number*
physical_address	P*number*
length	L*number*
rounding	R*number*
alignment	A*number*

There are three built-in segments with the following default attribute values:

- `text` (`LOAD`, `?RX`, no *virtual_address*, *physical_address*, or *length* specified, *alignment* values set to defaults per CPU type)
- `data` (`LOAD`, `?RWX`, no *virtual_address*, *physical_address*, or *length* specified, *alignment* values set to defaults per CPU type)
- `note` (`NOTE`)

The link-editor behaves as if these segments are declared before your `mapfile` is read in. See “Mapfile Option Defaults” on page 239 for more information.

Note the following when entering segment declarations:

- A *number* can be hexadecimal, decimal, or octal, following the same rules as in the C language.
- No space is allowed between the `V`, `P`, `L`, `R`, or `A` and the *number*.
- The *segment_type* value can be either `LOAD` or `NOTE`.
- The *segment_type* value defaults to `LOAD`.

- The *segment_flags* values are `R` for readable, `W` for writable, `X` for executable, and `O` for order. No spaces are allowed between the question mark (`?`) and the individual flags that make up the *segment_flags* value.
- The *segment_flags* value for a `LOAD` segment defaults to `RWX`.
- `NOTE` segments cannot be assigned any segment attribute value other than a *segment_type*.
- Implicitly declared segments default to *segment_type* value `LOAD`, *segment_flags* value `RWX`, a default *virtual_address*, *physical_address*, and *alignment* value, and have no *length* limit.

Note – the link-editor calculates the addresses and length of the current segment based on the previous segment's attribute values. Also, even though implicitly declared segments default to "no length limit," machine memory limitations still apply.

- `LOAD` segments can have an explicitly specified *virtual_address* value and/or *physical_address* value, as well as a maximum segment *length* value.
- If a segment has a *segment_flags* value of `?` with nothing following, the value defaults to not readable, not writable, and not executable.
- The *alignment* value is used in calculating the virtual address of the beginning of the segment. This alignment only affects the segment for which it is specified; other segments still have the default alignment unless their alignments are also changed.
- If any of the *virtual_address*, *physical_address*, or *length* attribute values are not set, the link-editor calculates these values as it builds the `a.out`.
- If an *alignment* value is not specified for a segment, it is set to the built-in default. (The default differs from one CPU to another and may even differ between kernel versions. You should check the appropriate documentation for these numbers).
- If both a *virtual_address* and an *alignment* value are specified for a segment, the *virtual_address* value takes priority.
- If a *virtual_address* value is specified for a segment, the alignment field in the program header contains the default *alignment* value.

- If the rounding value is set for a segment, that segments virtual address will be rounded to the next address that conforms to the value given. This value only effects the segments that it is specified for. If no value is given no rounding is performed.

The `?N` flag lets you control whether the ELF header, and any program headers, will be included as part of the first loadable segment. By default, the ELF header and program headers are included with the first segment, as the information in these headers is used within the mapped image (commonly by the runtime linker). The use of the `?N` option causes the virtual address calculations for the image to start at the first *section* of the first segment.

The `?O` flag lets you control the order of sections in the final relocatable object, executable file or shared object. This flag is intended for use in conjunction with the `-xF` option to the compiler(s). When a file is compiled with the `-xF` option, each function in that file is placed in a separate section with the same attributes as the *.text* section. These sections are now called *.text%function_name.*

For example, a file containing three functions `main()`, `foo()` and `bar()`, when compiled with the `-xF` option, will yield an object file with text for the three functions in sections called *.text%main*, *.text%foo* and *.text%bar*. Because the `-xF` option forces one function per section, the use of the `?O` flag to control the order of sections in effect controls the order of functions.

Consider the following user-defined `mapfile`:

```
text = LOAD ?RXO;
text: .text%foo;
text: .text%bar;
text: .text%main;
```

If the order of function definitions in the source file is `main`, `foo` and `bar`, then the final executable will contain functions in the order `foo`, `bar` and `main`.

For static functions with the same name the file names must also be used. The `?O` flag forces the ordering of sections as requested in the `mapfile`. For example, if a static function `bar()` exists in files `a.o` and `b.o`, and function `bar` from file `a.o` is to be placed before function `bar` from file `b.o`, then the `mapfile` entries should read:

```
text: .text%bar: a.o;
text: .text%bar: b.o;
```

Although the syntax allows for the entry:

```
text: .text%bar: a.o b.o;
```

this entry does not guarantee that function `bar` from file `a.o` will be placed before function `bar` from file `b.o`. The second format is not recommended as the results are not reliable.

Note – If a *virtual_address* value is specified, the segment is placed at that virtual address. For the system kernel this creates a correct result. For files that start via `exec(2)`, this method creates an incorrect `a.out` file because the segments do not have correct offsets relative to their page boundaries.

Mapping Directives

A mapping directive tells the link-editor how to map input sections to output segments. Basically, you name the segment that you are mapping to and indicate what the attributes of a section must be in order to map into the named segment. The set of *section_attribute_values* that a section must have to map into a specific segment is called the “entrance criteria” for that segment. In order to be placed in a specified segment of the `a.out`, a section must meet the entrance criteria for a segment exactly.

A mapping directive has the following syntax:

```
segment_name : {section_attribute_value}* [: {file_name}+];
```

For a *segment_name*, you specify any number of *section_attribute_value*s in any order, each separated by a space. (At most one section attribute value is allowed for each section attribute.) You can also specify that the section must come from a certain `.o` file(s) via the *file_name* substitutable. The section attributes and their valid values are as follows:

Table 7-2 Section Attributes

Section Attribute	Value
section_name:	`any valid section name`

section_type:	`$PROGBITS`
	`$SYMTAB`
	`$STRTAB`
	`$REL`
	`$RELA`
	`$NOTE`
	`$NOBITS`

section_flags:	? [[!]A] [[!]W] [[!]X]

Note the following when entering mapping directives:

- You must choose at most one *section_type* from the *section_type*s listed above. The *section_type*s listed above are built-in types. For more information on *section_type*s, see "Sections" on page 148.
- The *section_flags* values are `A` for allocatable, `W` for writable, or `X` for executable. If an individual flag is preceded by an exclamation mark (`!`), the link-editor checks to make sure that the flag is not set. No spaces are allowed between the question mark, exclamation mark(s), and the individual flags that make up the *section_flags* value.
- *file_name* may be any legal file name and can be of the form *archive_name(component_name)*, for example, `/usr/lib/usr/libc.a(printf.o)`. A file name may be of the form `*filename` (see next bullet item). Note that the link-editor does not check the syntax of file names.
- If a *file_name* is of the form `*filename`, the link-editor simulates a `basename(1)` on the file name from the command line and uses that to match against the specified `filename`. In other words, the `filename` from the `mapfile` only needs to match the last part of the file name from the command line. (See "Mapping Example" on page 236.)

- If you use the `-l` option during a link-edit, and the library after the `-l` option is in the current directory, you must precede the library with `./` (or the entire path name) in the `mapfile` in order to create a match.
- More than one directive line may appear for a particular output segment, for example, the following set of directives is legal:

```
S1 : $PROGBITS;
S1 : $NOBITS;
```

Entering more than one mapping directive line for a segment is the only way to specify multiple values of a section attribute.

- A section can match more than one entrance criteria. In this case, the first segment encountered in the `mapfile` with that entrance criteria is used, for example, if a `mapfile` reads:

```
S1 : $PROGBITS;
S2 : $PROGBITS;
```

the `$PROGBITS` sections are mapped to segment `S1`.

Section-within-Segment Ordering

By using the following notation it is possible to specify the order that sections will be placed within a segment:

```
segment_name | section_name1;
segment_name | section_name2;
segment_name | section_name3;
```

The sections that are named in the above form will be placed before any unnamed sections, and in the order they are listed in the `mapfile`.

Size-Symbol Declarations

Size-symbol declarations let you define a new global-absolute symbol that represents the size, in bytes, of the specified segment. This symbol can be referenced in your object files. A size-symbol declaration has the following syntax:

```
segment_name @ symbol_name;
```

symbol_name can be any legal C identifier, although the link-editor does not check the syntax of the *symbol_name*.

File Control Directives

File control directives allow users to specify which version definitions within shared objects are to be made available during a link-edit. The file control definition has the following syntax:

```
shared_object_name - version_name [ version_name ... ];
```

version_name is a version definition name contained within the specified *shared_object_name*. For more information on version control see "Specifying a Version Binding" on page 132.

Mapping Example

Following is an example of a user-defined `mapfile`. The numbers on the left are included in the example for tutorial purposes. Only the information to the right of the numbers actually appears in the `mapfile`.

Code Example 7-1 User-Defined Mapfile

```
1.      elephant : .data : peanuts.o *popcorn.o;
2.      monkey : $PROGBITS ?AX;
3.      monkey : .data;
4.      monkey = LOAD V0x80000000 L0x4000;
5.      donkey : .data;
6.      donkey = ?RX A0x1000;
7.      text = V0x80008000;
```

Four separate segments are manipulated in this example. The implicitly declared segment `elephant` (line 1) receives all of the *.data* sections from the files `peanuts.o` and `popcorn.o`. Note that `*popcorn.o` matches any `popcorn.o` file that may be supplied to the link-edit; the file need not be in the current directory. On the other hand, if `/var/tmp/peanuts.o` was supplied to the link-edit, it will not match `peanuts.o` because it is not preceded by a `*`.

The implicitly declared segment `monkey` (line 2) receives all sections that are both `$PROGBITS` and allocatable-executable (`?AX`), as well as all sections (not already in the segment `elephant`) with the name *.data* (line 3). The *.data* sections entering the `monkey` segment need not be `$PROGBITS` or allocatable-executable because the *section_type* and *section_flags* values are entered on a separate line from the *section_name* value. (An “and” relationship exists between attributes on the same line as illustrated by `$PROGBITS` “and” `?AX` on line 2. An “or” relationship exists between attributes for the same segment that span more than one line as illustrated by `$PROGBITS ?AX` on line 2 “or” *.data* on line 3.)

The `monkey` segment is implicitly declared in line 2 with *segment_type* value `LOAD`, *segment_flags* value `RWX`, and no *virtual_address*, *physical_address*, *length* or *alignment* values specified (defaults are used). In line 4 the *segment_type* value of `monkey` is set to `LOAD` (since the *segment_type* attribute value does not change, no warning is issued), *virtual_address* value to 0x80000000 and maximum *length* value to 0x4000.

Line 5 implicitly declares the `donkey` segment. The entrance criteria are designed to route all *.data* sections to this segment. Actually, no sections fall into this segment because the entrance criteria for `monkey` in line 3 capture all of these sections. In line 6, the *segment_flags* value is set to `?RX` and the *alignment* value is set to 0x1000 (since both of these attribute values changed, a warning is issued).

Line 7 sets the *virtual_address* value of the `text` segment to 0x80008000.

The example of a user-defined `mapfile` is designed to cause warnings for illustration purposes. If you want to change the order of the directives to avoid warnings, use the following example:

```
1.      elephant : .data : peanuts.o *popcorn.o;
4.      monkey = LOAD V0x80000000 L0x4000;
2.      monkey : $PROGBITS ?AX;
3.      monkey : .data;
6.      donkey = ?RX A0x1000;
5.      donkey : .data;
7.      text = V0x80008000;
```

The following `mapfile` example uses the segment within section ordering:

```
1.      text = LOAD ?RXN V0xf0004000;
2.      text | .text;
3.      text | .rodata;
4.      text : $PROGBITS ?A!W;
5.      data = LOAD ?RWX R0x1000;
```

The `text` and `data` segments are manipulated in this example. Line 1 declares the `text` segment to have a *virtual_address* of 0xf0004000 and to *not* include the ELF header or any program headers as part of this segments address calculations. Lines 2 and 3 turn on section-within-segment ordering and specify that the *.text* and *.rodata* sections will be the first two sections in this segment. The result is that the *.text* section will have a virtual address of 0xf0004000, and the *.rodata* section will immediately follow that.

Any other `$PROGBITS` section that make up the `text` segment will follow the *.rodata* section. Line 5 declares the `data` segment and specifies that its virtual address must begin on a 0x1000 byte boundary. The first section that constitutes the `data` segment will also reside on a 0x1000 byte boundary within the file image.

Mapfile Option Defaults

The link-editor defines three built-in segments (`text`, `data`, and `note`) with default *segment_attribute_value*s and corresponding default mapping directives as described in "Segment Declarations" on page 229. Even though the link-editor does not use an actual `mapfile` to provide the defaults, the model of a default `mapfile` helps illustrate what happens when the link-editor encounters your `mapfile`.

The example below shows how a `mapfile` would appear for the link-editor defaults. The link-editor begins execution behaving as if the `mapfile` has already been read in. Then the link-editor reads your `mapfile` and either augments or makes changes to the defaults.

```
text = LOAD ?RX;
text : ?A!W;
data = LOAD ?RWX;
data : ?AW;
note = NOTE;
note : $NOTE;
```

As each segment declaration in your `mapfile` is read in, it is compared to the existing list of segment declarations as follows:

1. If the segment does not already exist in the `mapfile`, but another with the same *segment-type* value exists, the segment is added before all of the existing segments of the same *segment_type*.
2. If none of the segments in the existing `mapfile` has the same *segment_type* value as the segment just read in, then the segment is added by *segment_type* value to maintain the following order:

   `INTERP`

   `LOAD`

   `DYNAMIC`

   `NOTE`

3. If the segment is of *segment_type* `LOAD` and you have defined a *virtual_address* value for this `LOAD`able segment, the segment is placed before any `LOAD`able segments without a defined *virtual_address* value or with a higher `virtual_address` value, but after any segments with a *virtual_address* value that is lower.

As each mapping directive in a `mapfile` is read in, the directive is added after any other mapping directives that you already specified for the same segment but before the default mapping directives for that segment.

Internal Map Structure

One of the most important data structures in the ELF-based link-editor is the map structure. A default map structure, corresponding to the model default `mapfile` mentioned above, is used by the link-editor when the command is executed. Then, if the mapfile option is used, the link-editor parses the `mapfile` to augment and/or override certain values in the default map structure.

A typical (although somewhat simplified) map structure is illustrated in Figure6-1. The “Entrance Criteria” boxes correspond to the information in the default mapping directives and the “Segment Attribute Descriptors” boxes correspond to the information in the default segment declarations. The “Output Section Descriptors” boxes give the detailed attributes of the sections that fall under each segment. The sections themselves are in circles.

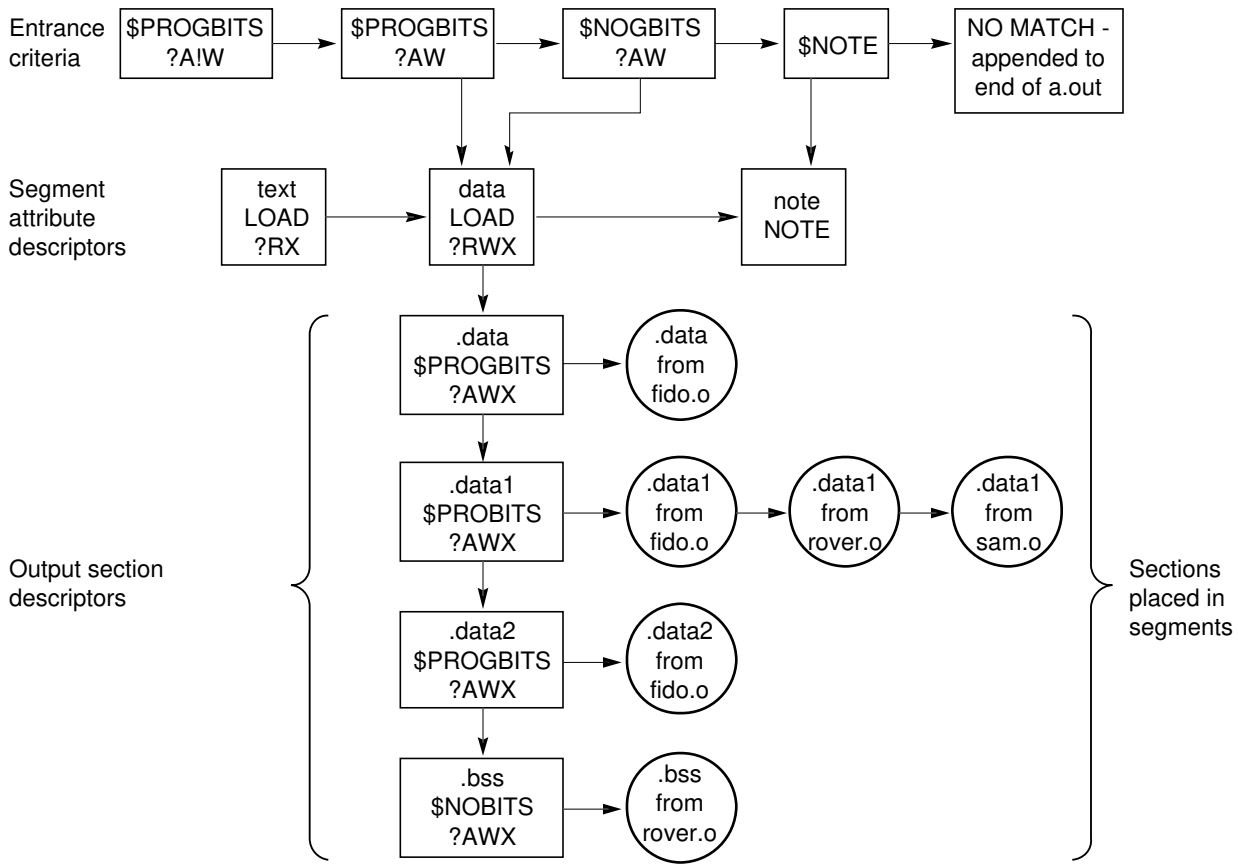

Figure 7-1 Simple Map Structure

The link-editor performs the following steps when mapping sections to segments:

1. When a section is read in, the link-editor checks the list of Entrance Criteria looking for a match. All specified criteria must be matched.

In Figure 6-1, for a section to fall into the `text` segment it must have a *section_type* value of `$PROGBITS` and have a *section_flags* value of `?A!W`. It need not have the name *.text* since no name is specified in the Entrance Criteria. The section may be either `X` or `!X` (in the *section_flags* value) since nothing was specified for the execute bit in the Entrance Criteria.

If no Entrance Criteria match is found, the section is placed at the end of the `a.out` file after all other segments. (No program header entry is created for this information. See "Program Header" on page 189 for more information.)

2. When the section falls into a segment, the link-editor checks the list of existing Output Section Descriptors in that segment as follows:

   If the section attribute values match those of an existing Output Section Descriptor exactly, the section is placed at the end of the list of sections associated with that Output Section Descriptor.

   For instance, a section with a *section_name* value of *.data1*, a *section_type* value of `$PROGBITS`, and a *section_flags* value of `?AWX` falls into the second Entrance Criteria box in Figure 6-1, placing it in the `data` segment. The section matches the second Output Section Descriptor box exactly (*.data1*, `$PROGBITS`, `?AWX`) and is added to the end of the list associated with that box. The *.data1* sections from `fido.o`, `rover.o`, and `sam.o` illustrate this point.

   If no matching Output Section Descriptor is found, but other Output Section Descriptors of the same *section_type* exist, a new Output Section Descriptor is created with the same attribute values as the section and that section is associated with the new Output Section Descriptor. The Output Section Descriptor (and the section) are placed after the last Output Section Descriptor of the same *section_type*. The *.data2* section in Figure 6-1 was placed in this manner.

   If no other Output Section Descriptors of the indicated *section_type* exist, a new Output Section Descriptor is created and the section is placed in that section.

Note – If the input section has a user-defined *section_type* value (that is, between `SHT_LOUSER` and `SHT_HIUSER`, as described in the "Sections" on page 148) it is treated as a `$PROGBITS` section. Note that no method exists for

naming this *section_type* value in the `mapfile`, but these sections can be redirected using the other attribute value specifications (*section_flags*, *section_name*) in the entrance criteria.

3. If a segment contains no sections after all of the command line object files and libraries are read in, no program header entry is produced for that segment.

Note – Input sections of type `$SYMTAB`, `$STRTAB`, `$REL`, and `$RELA` are used internally by the link-editor. Directives that refer to these *section_type*s can only map output sections produced by the link-editor to segments.

Error Messages

Warnings

Errors within this category do not stop execution of the link-editor nor do they prevent the link-editor from producing a viable `a.out`. The following conditions produce warnings:

- A *physical_address* or a *virtual_address* value or a *length* value appears for any segment other than a `LOAD` segment. (The directive is ignored.)
- A second declaration line exists for the same segment that changes an attribute value(s). (The second declaration overrides the original.)
- An attribute value(s) (*segment_type* and/or *segment_flags* for `text` and `data`; *segment_type* for `note`) was changed for one of the built-in segments
- An attribute value(s) (*segment_type*, *segment_flags*, *length* and/or *alignment*) was changed for a segment created by an implicit declaration. If only the `?O` flag has been added then the change of attribute value warning will not be generated.
- An entrance criteria was not met. If the `?O` flag has been turned on and if none of the input sections met an entrance criteria, the warning is generated.

7

Fatal Errors

Errors within this category stop execution of the link-editor at the point the fatal error occurred. The following conditions produce fatal errors:

- A `mapfile` cannot be opened or read.
- A syntax error is found in the `mapfile`.

Note – The link-editor does not return an error if a *file_name*, *section_name*, *segment_name* or *symbol_name* does not conform to the rules under the "Mapfile Structure and Syntax" section unless this condition produces a syntax error. For instance, if a name begins with a special character and this name is at the beginning of a directive line, the link-editor returns an error. If the name is a *section_name* (appearing within the directive), the link-editor does not return an error.

- More than one *segment_type*, *segment_flags*, *virtual_address*, *physical_address*, *length*, or *alignment* value appears on a single declaration line.
- You attempt to manipulate either the `interp` segment or `dynamic` segment in a `mapfile`.

Note – The `interp` and `dynamic` segments are special built-in segments that you cannot change in any way.

- A segment grows larger than the size specified by a your *length* attribute value.
- A user-defined *virtual_address* value causes a segment to overlap the previous segment.
- More than one *section_name*, *section_type*, or *section_flags* value appears on a single directive line.
- A flag and its complement (for example, `A` and `!A`) appear on a single directive line.