## 7.86 `.section name`

Use the `.section` directive to assemble the following code into a section named *name*.

This directive is only supported for targets that actually support arbitrarily named sections; on `a.out` targets, for example, it is not accepted, even with a standard `a.out` section name.

### COFF Version

For COFF targets, the `.section` directive is used in one of the following ways:

```
.section name[, "flags"]
.section name[, subsection]
```

If the optional argument is quoted, it is taken as flags to use for the section. Each flag is a single character. The following flags are recognized:

`b`

bss section (uninitialized data)

`n`

section is not loaded

`w`

writable section

`d`

data section

`e`

exclude section from linking

`r`

read-only section

`x`

executable section

`s`

shared section (meaningful for PE targets)

`a`

ignored. (For compatibility with the ELF version)

`y`

section is not readable (meaningful for PE targets)

`0-9`

single-digit power-of-two section alignment (GNU extension)

If no flags are specified, the default flags depend upon the section name. If the section name is not recognized, the default will be for the section to be loaded and writable. Note the `n` and `w` flags remove attributes from the section, rather than adding them, so if they are used on their own it will be as if no flags had been specified at all.

If the optional argument to the `.section` directive is not quoted, it is taken as a subsection number (see Sub-Sections).

## ELF Version

This is one of the ELF section stack manipulation directives. The others are `.subsection` (see `.subsection name`), `.pushsection` (see `.pushsection name [, subsection] [, "flags"[, @type[,arguments]]]`), `.popsection` (see `.popsection`), and `.previous` (see `.previous`).

For ELF targets, the `.section` directive is used like this:

```
.section name [, "flags"[, @type[,flag_specific_arguments]]]
```

If the ‘`--sectname-subst`’ command-line option is provided, the *name* argument may contain a substitution sequence. Only `%S` is supported at the moment, and substitutes the current section name. For example:

```
        .macro exception_code
        .section %S.exception
        [exception code here]
        .previous
        .endm

        .text
        [code]
        exception_code
        [...]

        .section .init
        [init code]
        exception_code
        [...]
```

The two `exception_code` invocations above would create the `.text.exception` and `.init.exception` sections respectively. This is useful e.g. to discriminate between ancillary sections that are tied to setup code to be discarded after use from ancillary sections that need to stay resident without having to define multiple `exception_code` macros just for that purpose.

The optional *flags* argument is a quoted string which may contain any combination of the following characters:

`a`

section is allocatable

`d`

section is a GNU_MBIND section

`e`

section is excluded from executable and shared library.

`o`

section references a symbol defined in another section (the linked-to section) in the same file.

`w`

section is writable

`x`

section is executable

`M`

section is mergeable

`S`

section contains zero terminated strings

`G`

section is a member of a section group

`T`

section is used for thread-local-storage

`?`

section is a member of the previously-current section’s group, if any

`+`

section inherits attributes and (unless explicitly specified) type from the previously-current section, adding other attributes as specified

`-`

section inherits attributes and (unless explicitly specified) type from the previously-current section, removing other attributes as specified

`R`

retained section (apply SHF_GNU_RETAIN to prevent linker garbage collection, GNU ELF extension)

`<number>`

a numeric value indicating the bits to be set in the ELF section header’s flags field. Note - if one or more of the alphabetic characters described above is also included in the flags field, their bit values will be ORed into the resulting value.

`<target specific>`

some targets extend this list with their own flag characters

Note - once a section’s flags have been set they cannot be changed. There are a few exceptions to this rule however. Processor and application specific flags can be added to an already defined section. The `.interp`, `.strtab` and `.symtab` sections can have the allocate flag (`a`) set after they are initially defined, and the `.note-GNU-stack` section may have the executable (`x`) flag added. Also note that the `.attach_to_group` directive can be used to add a section to a group even if the section was not originally declared to be part of that group.

Note further that `+` and `-` need to come first and can only take the effect described here unless overridden by a target. The attributes inherited are those in effect at the time the directive is processed. Attributes added later (see above) will not be inherited. Using either together with `?` is undefined at this point.

The optional *type* argument may contain one of the following constants:

`@progbits`

section contains data

`@nobits`

section does not contain data (i.e., section only occupies space)

`@note`

section contains data which is used by things other than the program

`@init_array`

section contains an array of pointers to init functions

`@fini_array`

section contains an array of pointers to finish functions

`@preinit_array`

section contains an array of pointers to pre-init functions

`@<number>`

a numeric value to be set as the ELF section header’s type field.

`@<target specific>`

some targets extend this list with their own types

Many targets only support the first three section types. The type may be enclosed in double quotes if necessary.

Note on targets where the `@` character is the start of a comment (eg ARM) then another character is used instead. For example the ARM port uses the `%` character.

Note - some sections, eg `.text` and `.data` are considered to be special and have fixed types. Any attempt to declare them with a different type will generate an error from the assembler.

If *flags* contains the `M` symbol then the *type* argument must be specified as well as an extra argument—*entsize*—like this:

```
        .section name , "flags"M, @type, entsize
```

Sections with the `M` flag but not `S` flag must contain fixed size constants, each *entsize* octets long. Sections with both `M` and `S` must contain zero terminated strings where each character is *entsize* bytes long. The linker may remove duplicates within sections with the same name, same entity size and same flags. *entsize* must be an absolute expression. For sections with both `M` and `S`, a string which is a suffix of a larger string is considered a duplicate. Thus `"def"` will be merged with `"abcdef"`; A reference to the first `"def"` will be changed to a reference to `"abcdef"+3`.

If *flags* contains the `o` flag, then the *type* argument must be present along with an additional field like this:

```
        .section name,"flags"o,@type,SymbolName|SectionIndex
```

The *SymbolName* field specifies the symbol name which the section references. Alternatively a numeric *SectionIndex* can be provided. This is not generally a good idea as section indices are rarely known at assembly time, but the facility is provided for testing purposes. An index of zero is allowed. It indicates that the linked-to section has already been discarded.

Note: If both the *M* and *o* flags are present, then the fields for the Merge flag should come first, like this:

```
        .section name,"flags"Mo,@type,entsize,SymbolName
```

If *flags* contains the `G` symbol then the *type* argument must be present along with an additional field like this:

```
        .section name , "flags"G, @type, GroupName[, linkage]
```

The *GroupName* field specifies the name of the section group to which this particular section belongs. The optional linkage field can contain:

`comdat`

indicates that only one copy of this section should be retained

`.gnu.linkonce`

an alias for comdat

Note: if both the *M* and *G* flags are present then the fields for the Merge flag should come first, like this:

```
.section name , "flags"MG, @type, entsize, GroupName[, linkage]
```

If both `o` flag and `G` flag are present, then the *SymbolName* field for `o` comes first, like this:

```
.section name,"flags"oG,@type,SymbolName,GroupName[,linkage]
```

If *flags* contains the `?` symbol then it may not also contain the `G` symbol and the *GroupName* or *linkage* fields should not be present. Instead, `?` says to consider the section that's current before this directive. If that section used `G`, then the new section will use `G` with those same *GroupName* and *linkage* fields implicitly. If not, then the `?` symbol has no effect.

The optional *unique,<number>* argument must come last. It assigns *<number>* as a unique section ID to distinguish different sections with the same section name like these:

```
.section name,"flags",@type,unique,<number>
.section name,"flags"G,@type,GroupName,[linkage],unique,<number>
.section name,"flags"MG,@type,entsize,GroupName[,linkage],unique,<number>
```

The valid values of *<number>* are between 0 and 4294967295.

If no flags are specified, the default flags depend upon the section name. If the section name is not recognized, the default will be for the section to have none of the above flags: it will not be allocated in memory, nor writable, nor executable. The section will contain data.

For SPARC ELF targets, the assembler supports another type of `.section` directive for compatibility with the Solaris assembler:

```
.section "name"[, flags...]
```

Note that the section name is quoted. There may be a sequence of comma separated flags:

`#alloc`

    section is allocatable

`#write`

    section is writable

`#execinstr`

    section is executable

`#exclude`

section is excluded from executable and shared library.

`#tls`

section is used for thread local storage

This directive replaces the current section and subsection. See the contents of the gas testsuite directory `gas/testsuite/gas/elf` for some examples of how this directive and the other section stack directives work.

---

Next: `.set *symbol*, *expression*`, Previous: `.scl *class*`, Up: Assembler Directives [Contents][Index]