ELF Assembler User's Guide

Version 4.2

SA14-2562-00

Fourth edition (August 2000)

This edition of the IBM ELF Assembler User's Guide for PowerPC applies to the IBM ELF Assembler version 4.2 and to subsequent versions until otherwise indicated in new versions or technical newsletters.

Address comments about this document to:

IBM Corporation
Department 5OB/Building 667
Box 12195
RTP, NC 27709

Printed in the United States of America.

4 3 2 1

Contents

Tables

About This Book

The **ELF Assembler User's Guide for PowerPC** describes how to use the ELF (Executable and Linkable Format) assembler for the PowerPC microprocessor.

This manual does not contain information about PowerPC assembler mnemonics. It is intended to be used along with your PowerPC microprocessor documentation.

Notational and Typographic Conventions

This manual uses several notational and typographic conventions to visually differentiate text.

Convention	Meaning
`Courier`	Indicates program text, input, output, file names
`Bold Courier`	Indicates commands, keywords, literal options
`Italic Courier`	Indicates formal parameters to be replaced by user-specified names or values; user input on the command line
Emphasized text	Indicates special terms and term definitions
{x**|**y**|**z**}**	Instructs user to choose one and only one of the options enclosed in curly braces and separated by vertical bars
[x**|**y**|**z**]**	Instructs user to choose none, one, some, or all of the options enclosed in square brackets and separated by vertical bars
...	Indicates multiple entries of the same type
|	Separates choices within brackets

Where to Go for More Information

The main `readme` file describes any special files and provides information that was not available at the time the manuals were published.

The *Where to Go for More Information* section in the *About This Book* chapter of the **High C/C++ Programmer's Guide** describes the following:

- Documents in the High C/C++ Toolset
- C and C++ Programming Documents
- Processor-Specific Documents
- Specifications and ABI Documents

For information about the Executable and Linkable Format (ELF), see the **TIS Portable Formats Specification Version 1.1**. TIS Committee, 1993.

Using the Assembler 1

This chapter describes how to use the ELF assembler; it covers the following topics:

Overview The assembler takes assembly-language text files as input and generates relocatable object files conforming to the Executable and Linkable Format (ELF). The assembler accepts input files containing instruction mnemonics as described in the documentation for your particular microprocessor.

1.1 Invoking the Assembler

This is the command-line syntax for invoking the assembler:

asppc [*options*] *source_file*.s

- *options* is an optional series of command-line options. (Assembler options are described in detail in §1.2: *Command-Line Options*.)
- *source_file*.s is the name of the assembly source file (or files) being assembled, with a default extension .s.

Note:	If you enter **asppc** without a source-file name, you get the command-line help screen.

Note:	Whitespace is required between elements in a command line, except between an option and its argument; for example, either **-o** hello.o or **-o**hello.o is acceptable.

The Output File Name

By default, the assembler generates an object file with the same file name as the assembly source file, but with a .o extension:

source_file.o

To specify a different output-file name, use command-line option **-o**.

You can assemble more than one source file with a single command. The assembler concatenates all the source files into one object file with the same file name as the last source file specified. For example, the following command generates one output file, `srcfile3.o`, which contains object code for all three source files:

```
asppc srcfile1.s srcfile2.s srcfile3.s
```

Generating an Output Listing

The assembler generates an assembly-language output listing only if you specify command-line option **-l**. The assembler directs the listing to standard output, unless you redirect the listing to an output file. See §1.2.2: *Command-Line Option Reference* for more information about option **-l**.

Preprocessing

The ELF assembler includes a macro preprocessor. If your source files contain C-style preprocessing directives, you must invoke the High C/C++ driver with driver command-line option **-Hasmcpp** to preprocess these files. For example, the following command preprocesses C-style directives in `srcfile.s` and invokes the assembler to assemble the preprocessed output:

```
hcppc -c -Hasmcpp srcfile.s
```

Caution:	Comments in assembly code can cause the preprocessor to fail if they contain C preprocessing tokens such as **#if** or **#end**, C comment delimiters, or invalid C tokens.

See the **High C/C++ Programmer's Guide** for information about driver option **-Hasmcpp**.

1.2 Command-Line Options

Assembler command-line options determine how to assemble the source file and what output to generate.

You must place assembler options on the command line before the name of the source files you want them to affect.

Note:	Names of command-line options are case-sensitive.

To see a listing of all the assembler command-line options, enter **`asppc`** with no file name at the command prompt.

1.2.1 Passing Assembler Options from the Driver

You can use assembler command-line options with the **`asppc`** command or with the **`hcppc`** driver command. To pass assembler options to the assembler using the driver, use driver command-line option **`-Hasopt`**:

```
hcppc -Hasopt=-l hello.c
```

See the **High C/C++ Programmer's Guide** for more information about driver option **`-Hasopt`**.

1.2.2 Command-Line Option Reference

`-%reg` — Specify that register names must be preceded by a percent sign

Specifies that register names must be preceded by a percent sign (`%`), the register-name identifier, causing the assembler to recognize any name that does not begin with `%` as belonging to a user-defined identifier, not a register. By default, the assembler recognizes the name of any register, with or without the percent sign, as a register name.

Same as option **`-percent_reg`**.

See §2.1.4.2: *Using Register Names as Identifiers* for more information.

`-be` — Assemble using big-endian format

Causes the assembler to generate object code in big-endian format (the default).

`-big_si` — Suppress warning if a signed integer's value exceeds 32,767

Tells the assembler not to warn if the value of a signed integer in a signed-integer instruction exceeds 32,767 but is less than 65,536. The

assembler converts a signed integer larger than 32,767 to an unsigned integer, and normally warns when it does so, because there are situations where an unsigned integer can generate an error if a signed integer is expected.

`-c` — Suppress display of the copyright message

Suppresses the display of the copyright message that normally appears when you invoke the assembler.

`-D`*name*`[=`*n*`]` — Define an identifier and assign a value to it

Defines identifier *name* and, if you specify a value *n*, assigns *n* to *name*. If you do not specify *n*, the value of *name* defaults to `1` (one). Specifying option **`-D`***name*`=`*n* is the same as putting one of the following directives in your assembly source file:

```
.set name, n
.define name, n
```

`-diab` — Make ELF assembler compatible with Diab-style assembly language

Makes the ELF assembler compatible with programs written in Diab-style assembly language. When you invoke the assembler with option **`-diab`**, you place the following restrictions on assembly source code:

- The semicolon (`;`) introduces a comment. You can no longer use it as a statement separator. When you assemble with option **`-diab`**, each statement must be on a separate line.
- In macro definitions (with the **`.macro`** directive), the comma separating the macro name and the first parameter is optional:

  ```
  .macro name[,] param, param
  ```

- In section definitions (with the **`.section`** directive), the second argument specifies the alignment rather than the class, and the third argument specifies the class rather than the entry. (See the entry for directive **`.section`** for more information.)
- Directive **`.string`** does not terminate strings with a null character; that is, **`.string`** becomes a synonym for **`.ascii`** instead of for **`.asciz`**.

Note:	Option **`-diab`** causes the assembler to recognize instruction mnemonics without regard to case (upper, lower, or mixed).

`-eabi` — Make ELF assembler compatible with EABI-style assembly language

Makes the ELF assembler compatible with programs conforming to EABI specifications. When you invoke the assembler with option **`-eabi`**, the comma separating the macro name and the first parameter in macro definitions (with the **`.macro`** directive) becomes optional:

`.macro` *name*`[,]` *param, param*

`-Eo` — **Send error messages to standard output**

Causes the assembler to write error messages to `stdout` instead of to `stderr`.

`-errors` *n* — **Set the assembler error limit**

Sets the assembler error limit; that is, the maximum number of errors that the assembler allows before quitting. The default assembler error limit is 25. Specifying a higher error limit is useful when, for example, you debug source with multiple complex macros — it is difficult to determine where an error ocurrs without the help of the listing, which will not be produced if the error limit is reached.

`-f`*flag*`[`*flag*`...]` — **Set listing flags**

Sets listing flags that control the contents and appearance of the assembly source listing. Table 1.1 shows these flags and their default settings.

Table 1.1 Assembler Listing Flags

Flag	Function	Default
`c`	List instructions that were not assembled because of conditional assembly statements.	`Off`
`g`	List local label symbols in the symbol and cross-reference tables.	`Off`
`i`	List include files in the source listing.	`Off`
`l`*n*	Set line width of the source listing to *n* spaces ($40 \le n \le 255$).	*n*`=132`
`m`	List macros and show expansions in source listing.	`Off`
`o`	List data-storage overflow.	`Off`

Flag	Function	Default
p*n*	Set page length of the source listing to *n* lines ($20 \le n \le 255$; setting $n = 0$ (zero) means no pagination).	*n*=55
s	Show the symbol table in the source listing.	Off
x	Show the cross-reference table in the source listing.	Off

Setting and Toggling Listing Flags

To turn On a flag, list it after option **-f**. To turn Off a flag, list it after option **-f** and put an n before it. To set the l and p flags, insert an integer value after them in the flag list.

For example, the command **-f**imnsl80 causes the assembler to do the following:

- List include files in the source listing (flag i).
- Include macro expansions in the source listing (flag m).
- Not show the symbol table in the source listing (flag ns).
- Set source listing line-width to 80 characters (flag l80).

Option **-f** has the same effect as assembler directive **.lflags**, except that it can be applied only to the module as a whole.

Note: Option **-f** takes effect only if you also specify option **-l**.

-g — Generate debugging information for assembly source files

Causes the assembler to annotate the assembled object file with line-number information. This information allows the debugger to display the actual assembly source file, complete with comments and declarations, instead of disassembled instructions (which can look quite different from the original instructions in the assembly source file).

Option **-g** provides no symbolic or type information.

Debugging assembly source

The IBM RISCWatch debugger requires type and size attributes for functions in order to properly debug assembly source:

```
        .globl  MyFunction
MyFunction:
        li      %r0,0
        blr
        .type MyFunction, @function
        .size MyFunction, . - MyFunction
```

The assembler cannot determine which labels in a `.text` section are functions, and it cannot determine the size of a function. For the IBM debugger to reliably single step and display assembly language source files, you must explicitely define functions using the **`.type`** and **`.size`** directives.

`-h` — Display command-line option help screen

Displays a screen listing of the assembler command-line options, their arguments, and what they do.

`-I`*`include_dir`* — Specify directory to be searched for `.include` files

Directs the assembler to search directory *`include_dir`* for `.include` files with relative addresses. The assembler searches first the current working directory, then any directories specified with option **`-I`**.

Note: There is no limit to the number of directories you can specify with option **`-I`**; however, each directory must be specified with a separate instance of **`-I`**.

`-l` — Generate an assembly output listing

Directs the assembler to generate an assembly output listing.

The assembler writes this listing to standard output unless you redirect it to a list file. For example, the following command specifies an output listing and redirects it to a list file called `output.lst`:

```
asppc -l testfile.s > output.lst
```

`-L` — Place private labels in the output symbol table

Causes the assembler to place private labels in the output-file symbol table.

A private label is one that begins with a period; for example, `.my_label`.

`-le` — Assemble using little-endian format

Causes the assembler to generate object code in little-endian format (the default).

`-no_cpu_tag` — Do not encode target CPU in object file

When you specify a PowerPC target processor using one of the **`-Hppc*`** driver options, the assembler encodes the CPU type in the `e_flags` field of the ELF file header. Option **`-no_cpu_tag`** tells the assembler not to encode the CPU type in the `e_flags` field. (For information about the **`-Hppc*`** driver options, see the **High C/C++ Programmer's Guide**.)

`-o` *`object_file`* — Override the default object-file name

Overrides the default assembler-generated object-file name. The default is the name of the assembly source file with a `.o` extension.

For example, the following command generates an object file `sort_1.obj` (rather than the default, `sort.o`):

```
asppc -o sort_1.obj sort.s
```

`-pa` — Recognize POWER (not PowerPC) assembly

Causes the assembler to recognize POWER (non-PowerPC) assembly only.

`-percent_reg` — Specify that register names must be preceded by a percent sign

Same as option **`-%reg`**.

`-Q {y|n}` — Specify whether assembler version-number information appears in the object file

Option **`-Q`** `y` places assembler version-number information in the comment section of the generated object file. Option **`-Q`** `n` suppresses placement of this information.

Option **`-Q`** `n` is the default.

`-svr4` — Adhere to SVR4 assembler syntax

Causes the assembler to adhere to strict SVR4 syntax; for example, all register names must begin with percent sign (%).

Currently, option **`-svr4`** has the same effect as **`-%reg`** and **`-percent_reg`**.

`-tALTIVEC` — Assemble AltiVec instructions

Directs the assembler to assemble AltiVec instructions.

`-tMAC` — Assemble IBM Multiply Accumulate instructions

Directs the assembler to assemble IBM Multiply Accumulate instructions.

`-tPPC*` — Generate code for specified PowerPC processor

The **`-tPPC*`** options configure the assembler for a specific PowerPC processor by enabling recognition of specific register names for that processor's Special Purpose Registers, as follows:

`-tPPC400`	Generates code for IBM PowerPC 400 series
`-tPPC401`	Generates code for PowerPC 401
`-tPPC403`	Generates code for PowerPC 403
`-tPPC405`	Generates code for PowerPC 405
`-tPPC407`	Generates code for PowerPC 407
`-tPPC505`	Generates code for PowerPC 505
`-tPPC509`	Generates code for PowerPC 509
`-tPPC555`	Generates code for PowerPC 555
`-tPPC601`	Generates code for PowerPC 601
`-tPPC602`	Generates code for PowerPC 602
`-tPPC603`	Generates code for PowerPC 603
`-tPPC604`	Generates code for PowerPC 604
`-tPPC740`	Generates code for PowerPC 740
`-tPPC750`	Generates code for PowerPC 750
`-tPPC801`	Generates code for PowerPC 801
`-tPPC821`	Generates code for PowerPC 821
`-tPPC823`	Generates code for PowerPC 823
`-tPPC850`	Generates code for PowerPC 850
`-tPPC860`	Generates code for PowerPC 860
`-tPPC8240`	Generates code for MPC8240
`-tPPC8260`	Generates code for MPC8260

Note: New processor options are added frequently; see the `readme` file for a complete listing.

`-U`*`name`* — **Undefine an identifier**

Undefines an identifier *`name`* previously defined with option **`-D`**. In the following example, `abc` is defined for `file1.s`, but not defined for `file2.s`:

```
asppc -Dabc file1.s -Uabc file2.s
```

`-v` — **Print summary of assembler statistics**

Causes the assembler to write to standard output a summary of statistics about the program being assembled.

`-w` — **Suppress warning messages**

Tells the assembler not to emit warning messages. You should use this option only after you have determined that the conditions warned about are acceptable.

The ELF Assembly Language 2

This chapter describes the lexical features of the ELF assembly language; it covers the following topics:

2.1 Lexical Features of the ELF Assembler

A program for the ELF assembler is made up of statements written in the symbolic machine language specific to the target microprocessor(s).

2.1.1 Statement Fields

An assembly language statement contains the following fields:

[*label***:]** *opcode* **[***operands***]** **[{**!**|**#**}** *comment***]**

You can put more than one statement on a line, separated by semicolons.

Note:	If you invoke the assembler with option **`-diab`**, a semicolon introduces a comment. In that case, you must put statements on separate lines.

The Label Field

A *label* is a location marker. See §2.1.5: *Labels* for more information.

The Opcode Field

An *opcode* is typically an assembly-language mnemonic for a microprocessor instruction.

The opcode field can also contain an assembler directive (also called a pseudo-operation) or a user-defined macro instead of an instruction mnemonic. The opcode field can begin in any column.

See Chapter 4: *Assembler Directives* for a listing of directives available for the assembler. See Chapter 3: *Writing Assembler Macros* for instructions on how to write your own macros.

The Operands Field

An *operand* is an argument for the instruction in the opcode field. Whitespace separates the operands field from the opcode field.

An operand can be an identifier or a constant, or an expression containing one or more identifiers or constants.

The Comment Field

A *comment* contains information about the statement or group of statements to which it is attached. The comment field is optional. If a comment follows an opcode or operand, whitespace separates it from the opcode or operands field. A comment by itself on a line can begin in any column, provided it is preceded by the comment delimiter.

A comment begins with an exclamation point (`!`) or a pound sign (#). An asterisk (`*`) in the first column indicates that the rest of the line is a comment.

Note:	If you invoke the assembler with option **`-diab`**, a comment can also begin with a semicolon (`;`).

The comment continues for the rest of the line and ends with a line feed.

2.1.2 The ELF Assembler Character Set

The assembler recognizes the following characters:

- alphabetic characters: `A` through `Z`, `a` through `z`
- numeric characters (decimal digits): `0` (zero) through `9`
- special characters listed here:

`&`	Ampersand	`.`	Period
`*`	Asterisk	`+`	Plus sign
`@`	At sign	`#`	Pound sign
`\`	Backslash	`?`	Question mark
`^`	Caret	`>`	Right angle bracket
`:`	Colon	`)`	Right parenthesis
`,`	Comma	`]`	Right square bracket
`$`	Dollar sign	`;`	Semicolon
`"`	Double quote	`'`	Single quote
`=`	Equal sign	`/`	Slash
`!`	Exclamation point		Space
`<`	Left angle bracket		Tab
`(`	Left parenthesis	`~`	Tilde
`[`	Left square bracket	`_`	Underscore
`-`	Minus sign	`|`	Vertical bar
`%`	Percent sign		

Some special characters have a predefined function in assembly language:

- A single period (`.`) represents the current location counter.
- A semicolon (`;`) separates statements on a line, unless you have invoked the assembler with option **`-diab`**, in which case each statement must occupy a separate line.
- An exclamation point (`!`) or a pound sign (#) marks the beginning of a comment. If you invoke the assembler with option **`-diab`**, you can also use the semicolon (`;`) to mark the beginning of a comment.

- Because the exclamation point is the comment delimiter, it must be "escaped" with a backslash (\) if you use it in any other context; for example, when it is part of the "not equal to" operator: `\!=`.
- A backslash (\) at the end of a line is an escaped newline — the line continues onto the next line.

2.1.3 Identifiers

Identifiers are names of variables, labels, functions, registers, and instruction mnemonics. This section describes the rules to which the assembler requires identifiers to conform.

Identifiers can contain the following:

- alphabetic characters: `A` through `Z`, `a` through `z`
- numeric characters: `0` (zero) through `9`
- these special characters:
  - `$` (dollar sign)
  - `.` (period)
  - `_` (underscore)
  - `%` (percent sign)

Identifiers cannot start with numerals `0` (zero) through `9`.

Identifiers are case-sensitive, with the exception of register names and opcode mnemonics.

These are examples of valid identifiers:

```
month
.size
L14
_main
display__4DateFv
```

2.1.4 Symbols

A *symbol* is an identifier that can be used as an operand in an assembly statement. The assembler places an entry for each symbol in the symbol

table the first time it encounters that symbol. The assembler supports forward referencing of symbols; that is, a symbol can be placed in the symbol table before it is defined.

2.1.4.1 Reserved Symbols

The following symbols are *reserved*; that is, you cannot redefine them.

- A period (`.`) or dollar sign (`$`) by itself, which represents the location counter
- The register names and register-field names:
  - general-purpose registers `r0` through `r31`
  - floating-point registers `f0` through `f31`
  - condition-register fields `cr0` through `cr7`
  - AltiVec registers `v0` through `v31`

Note:	Register names do not have to be preceded by the `%` (percent sign) unless you have specified command-line option **`-%reg`** or **`-percent_reg`**, or directive **`.option`** with the `%reg` or `percent_reg` argument.

Note:	Register names are not case-sensitive.

2.1.4.2 Using Register Names as Identifiers

By default, the ELF assembler recognizes any register name as a register name, whether or not it it is preceded by a `%` (percent sign).

Any register name preceded by `%` is unambiguous and can only be a register name; it can never be mistaken for an identifier, because C and C++ do not allow identifier names that start with a percent sign.

To make the assembler accept as a user-defined symbol something that it would otherwise recognize as a register name without the percent sign, use one of these methods:

- Specify option **`-%reg`** or option **`-percent_reg`** on the command line.
- Place the directive **`.option`** `"%reg"` or **`.option`** `"percent_reg"` in the assembly code.

You can then use symbols with names like `r0`, `r5`, and so on in your assembly code.

Note: By default, assembly source code generated by the High C/C++ compiler contains only the percent-sign form of register names. The compiler passes option **`-%reg`** to the assembler to ensure that the assembler accepts names without the percent sign as user-defined identifiers. That is, for compiler-generated assembly code, all names without the percent sign are by default identifier names, not register names.

2.1.4.3 Built-In Symbols

The assembler supports the following built-in symbols:

Variable	Value
`$endian`	One of two string values, `big` or `little`, depending on the endian mode currently active
`$false`	Integer value `0` (zero)
`$macro`	Name of the macro currently being expanded (see §3.1.2.1: *Symbol $macro* for more information)
`$narg`	Number of arguments with which the current macro was called (see §3.1.2.2: *Symbol $narg* for more information)
`$true`	Integer value `1` (one)

These symbols can be used as operands in assembly statements or macro definitions; for example:

```
.ifeqs $endian, "big"
   .set BIG_ENDIAN, $true
.else
   .set BIG_ENDIAN, $false
.endif
```

2.1.5 Labels

A *label* is an identifier that marks the location of an instruction or a data location. When it encounters a valid label, the assembler assigns to the label the current value of the instruction counter. A label on an instruction marks a branch location, and is assigned the address of the instruction.

A label definition can begin in any column, but it must be the first item on the line. The definition terminates with a colon (`:`).

There are three types of labels: regular, local, and numeric.

There are two types of labels, regular and numeric.

2.1.5.1 Regular Labels

A *regular label* can be defined only once, because it is global to its module and must retain the same value throughout the module.

To make a regular label accessible to other modules, include it in a **`.global`** or **`.comm`** directive inside its module, or place a double colon after it:

```
main::
```

Names of regular labels follow the general syntax for identifiers, as defined in §2.1.3: *Identifiers*.

These are examples of regular labels:

```
main:
L00DATA:
L208.day:
display__4DateFv:
```

2.1.5.2 Local Labels

A *local label* begins with a `$`, followed by one to six decimal digits.

This is an example of a local label:

```
$00010:
```

A local label has limited scope, so you can redefine it as often as you need to. The scope of a local label is
one of the following:

- the body of a macro
- the body of a **.rep**, **.irep**, or **.irepc** directive
- an **.include** file

The scope of a local label ends with the next regular label.

2.1.5.3 Numeric Labels

A *numeric label* is a single digit in the range `0` (zero) to `9`; for example:

```
1:      mov %r12, %r15
7:      nop
```

A numeric label has limited scope, so you can redefine it as often as you need to.

A reference to a numeric label consists of a single digit followed by either `b` (for backward) or `f` (for forward):

- *n*`b` refers to the nearest numeric label *n* defined before the reference.
- *n*`f` refers to the nearest numeric label *n* defined after the reference.

For example, the code in Example 1 has the same meaning as the code in Example 2:

Example 1

```
        b 1f
        nop
1:      b 3f
        nop
2:      b 1f
3:      b 2b
1:      nop
```

Example 2

```
        b L1
        nop
L1:     b L3
        nop
L2:     b L1a
L3:     b L2
L1a:    nop
```

2.2 The Location Counter

The *location counter* is a variable in which the address of the current byte being assembled is stored. The assembler uses the location counter to assign addresses to assembled bytes.

You can also make use of the location counter. You use it like any other variable, except that you cannot place it in the label field of an instruction.

The symbol for the location counter is a period (`.`) or a dollar sign (`$`). When the source code is assembled, the assembler replaces the symbol with the address of the current byte.

Caution: The assembler warns if it must force the alignment of an instruction to a four-byte boundary. It does not warn when you use the location counter to make a jump to a misaligned address; it just jumps to the previous word boundary.

2.3 Constants

The assembler recognizes the following types of constants:

- integer constants
- floating-point (real) constants
- string constants

2.3.1 Integer Constants

The assembler recognizes binary, decimal, octal, and hexadecimal integer constants. Integer constants have the following prefixes:

Base	Prefix	Example
Binary	`0B` or `0b`	`0B101011`, `0b101011`
Octal	`0` (zero)	`053`

Base	Prefix	Example
Decimal	None	`43`
Hexadecimal	`0X` or `0x`	`0X2B`, `0x2b`

If an integer has no leading `0` (zero) or other prefix, the assembler assumes it is decimal.

Integer constants can be preceded by a unary plus or minus.

The assembler converts integers of any base to a two's-complement binary representation.

2.3.2 Floating-Point (Real) Constants

Floating-point constants do not require a prefix. The assembler recognizes both standard decimal formats and exponential formats as floating-point constants.

These are all floating-point constants:

```
4.0
3.14159
9e+7
5.374E-4
```

In any floating-point constant, if a decimal point is present, at least one digit must appear to the left of the decimal point. The digit can be `0` (zero).

You can put an underscore before the exponent to increase readability: `1.0782_e+2`. Embedded whitespace (space or tab) is not allowed.

Floating-point constants can be preceded by a unary plus or minus.

The assembler converts floating-point constants to an IEEE-format floating-point representation.

Use floating-point constants only with floating-point assembler directives **`.float`** and **`.double`**.

Note: It is possible to use a numeric constant without a decimal point or an exponential designation (for example, `7`) with the directives **`.float`** and **`.double`**.

The assembler interprets such a constant as an integer, and does not convert it to a floating-point format. This allows you to enter floating-point constants as hex or decimal bit patterns.

2.3.3 String Constants

A *string constant* is a sequence of characters of any length, enclosed in double quotes; for example, `"This is a string constant!"`.

You use string constants as arguments with the following assembler directives:

`.ascii`	**`.ident`**	**`.pushsect`**	**`.version`**
`.asciz`	**`.include`**	**`.section`**	**`.warn`**
`.err`	**`.machine`**	**`.seg`**	
`.file`	**`.print`**	**`.string`**	

2.3.3.1 Restrictions

String constants can contain any ASCII character, with the following restrictions:

- If single or double quotes are to be interpreted as ordinary characters rather than as delimiters, they must be preceded by a backslash (\): `"This is a \"character\" string."`
- If the backslash is to be interpreted as an ordinary character, it must be preceded by another backslash: `"This \\ is a character."`
- You must express ASCII control characters with these character combinations:

Control Character	ASCII Value	Character Combination	Control Character	ASCII Value	Character Combination
Alert	`0x07`	`\a`	Carriage return	`0x0d`	`\r`
Backslash	`0x08`	`\b`	Tab	`0x09`	`\t`
Form feed	`0x0c`	`\f`	Vertical tab	`0xob`	`\v`
Newline	`0x0a`	`\n`	NULL	`0x00`	`\0`

2.3.3.2 Octal or Hexadecimal Notation

In addition to using ASCII characters themselves in string constants, you can specify the ASCII value of the character in either octal or hexadecimal notation.

- An octal value is expressed as up to three octal characters preceded by a backslash (\).
- A hexadecimal value is expressed as up to two hexadecimal characters preceded by a backslash and a lowercase `x`.

For example, instead of `Q` in a string constant, you can use either `\121` or `\x51`.

2.4 Expressions

An *expression* is a series of one or more *operands* (identifiers, constants, and subexpressions) separated by arithmetic, logical, and/or relational operators; for example:

```
index <= 255
a + b
```

Expressions are used as operands with assembler instruction mnemonics and some assembler directives. See the documentation for your microprocessor for details about specific assembler instruction mnemonics. See Chapter 4: *Assembler Directives* for information about assembler directives.

2.4.1 Forcing Evaluation Order

The assembler evaluates an expression from left to right, taking into account the precedence of the operators. Once it has completed the evaluation, the assembler replaces the expression with the resulting value.

You can force the assembler to evaluate the expression in a specific order by enclosing subexpressions in parentheses; these subexpressions are evaluated before the rest of the expression.

2.5 Operators and Operator Precedence

Table 2.1 shows the arithmetic operators supported by the assembler, in their order of precedence, where a lower precedence number indicates a higher precedence of evaluation.

Table 2.1 Arithmetic Operators in Order of Precedence

	Precedence	Operator	Operation
Highest precedence	1	( )	Overrides precedence of any other operator
	2	+ - ~ \!	Unary plus Unary minus Unary bitwise logical NOT Unary logical NOT ***Note:*** Because ! is the syntactic delimiter for a comment, the NOT operator must be distinguished by an escape character.
	3	/ % *	Division Modulus Multiplication
	4	+ -	Addition Subtraction
	5	<< >>	Left shift Right shift
	6	< > <= >=	Less than Greater than Less than or equal to Greater than or equal to
	7	== <> \!=	Equal to Not equal to Not equal to ***Note:*** Because ! is the syntactic delimiter for a comment, the "not equal to" operator must be distinguished by an escape character.
	8	&	Bitwise logical AND
	9	^	Bitwise logical XOR
	10	|	Bitwise logical OR
	11	&&	Logical AND

	Precedence	Operator	Operation
	12	^^	Logical exclusive OR
Lowest precedence	13	||	Logical OR

2.5.1 Register Expressions

Register names can be used as operands in register expressions. Register arithmetic, however, is allowed only for addition with integers; for example:

```
%r4+3
```

2.6 Assignment Syntax

An assignment passes the value of an expression to an identifier.

The assembler recognizes the following forms of assignment syntax:

```
identifier  =  expression
identifier  := expression
identifier  =: expression

identifier: =  expression
identifier: := expression
identifier: =: expression
```

You can use the last three of these forms to assign a value to more than one identifier at a time. Each of the identifier names must be followed by a colon; for example:

```
name_1: name_2: name_3: = a + b
```

2.7 Attributes

Attributes modify references to identifiers. You can use them to indicate different relocation options.

Identifier attributes use either of the following syntaxes:

```
identifier@attribute
%attribute(identifier)
```

The ELF assembler supports the attributes listed in Table 2.2.

Table 2.2 Identifier Attributes

Attribute	Description	Relocation Entry Generated
`@got`	Address of the Global Offset Table (GOT) entry for the identifier	`R_PPC_GOT16`
`@local`	Address of a local identifier, as opposed to a global identifier of the same name	`R_PPC_LOCAL24PC`
`@off`	Offset of the identifier in the section where the identifier resides	`R_PPC_SECTOFF16`
`@plt`	Address of a function's Procedure Linkage Table (PLT) entry	`R_PPC_PLT32` for a data directive `R_PPC_PLT24` for a branch instruction
`@sda`	Offset of an identifier in the `.sdata` or `.sbss` section	`R_PPC_SDAREL16`
`@sda0`	Offset of an identifier in the `.sdata0` or `.sbss0` section	#121
`@sda0i`	Offset of a pointer to an identifier in the `.sdata0` or `.sbss0` section	#122
`@sda2`	Offset of an identifier in the `.sdata2` or `.sbss2` section	`R_PPC_EMB_SDA2REL`
`@sda2i`	Offset of a pointer to an identifier in the `.sdata2` or `.sbss2` section	`R_PPC_EMB_SDA2I16`
`@sdai`	Offset of a pointer to an identifier in the `.sdata` or `.sbss` section	`R_PPC_EMB_SDAI16`
`@sdax`	Offset of the identifier from the base address of the `.sdata` section	`R_PPC_EMB_RELSDA`

Attribute	Description	Relocation Entry Generated
@sdaxr	Combination of the base register for accessing the identifier and the offset of the identifier in the .sdata or .sbss section	R_PPC_EMB_SDA21
@sect	Base address of the section in which the identifier is stored	R_PPC_EMB_RELST
@sectoff	Offset of the identifier; same as @off	R_PPC_SECTOFF16
@stridx	Index of the entry for the identifier in the ELF string table	None
@symidx	Index of the entry for the identifier in the ELF symbol table	None

See the **High C/C++ Programmer's Guide** for more information about the .sdata, .sdata0, and .sdata2 sections.

Table 2.3 lists attributes used either by themselves or with other attributes to indicate high and low half-words of the address of the identifier referred to:

Table 2.3 Identifier Attributes That Specify High or Low Address Halves

Attribute	Description
@h	Upper half of the identifier address
@ha	Upper half of the identifier address, adjusted so the lower half can be used by instructions that interpret the lower half as signed
@l	Lower half of the identifier address

The @h, @ha, and @l attributes are used with the following attributes:

```
@got    @plt    @sect
@off    @sda    @sectoff
```

For example:

```
@got@h
@plt@ha
@sda@l
```

The `@h`, `@ha`, and `@l` attributes cause the assembler to generate the following relocation entries:

- `*16_LO`, `*16_HI`, or `*16_HA`, if the preceding attribute generates an entry of form `*32`
- `*_LO`, `*_HI`, or `*_HA`, if the preceding attribute generates an entry of form `*`

For example:

- `ident@got@l` generates relocation entry `R_PPC_GOT16_LO`.
- `ident@off@h` generates relocation entry `R_PPC_SECTOFF_HI`.
- `ident@sda@ha` generates relocation entry `R_PPC_SDAREL16_HA`.

Refer to the **AT&T UNIX System V Release 4 Programmer's Guide: ANSI C and Programming Support Tools** for the following information:

- how the ELF assembler handles dynamic relocation
- detailed information about the `@got` and `@plt` identifier attributes

Writing Assembler Macros 3

This chapter explains how to define and use assembly-language macros; it covers the following topics:

Overview A *macro* is a named block of assembly-language statements that the assembler inserts automatically into the assembly source code at any point where you put a special statement known as a *macro call*. You define a given macro only once, but you can call it as often as you want. A macro can have formal parameters. You can pass a different value to a macro parameter with each call to the macro.

3.1 Defining a Macro

A *macro definition* contains the actual code for the macro operation. It consists of three parts: the macro heading, the macro body, and the macro teminator.

3.1.1 Macro Heading

You introduce a macro heading with the **.macro** directive, followed by the name of the macro; then any parameters, separated from the macro name and from each other by commas, as in the following example:

```
.macro min_max, num1, num2
```

If you give a macro the name of an assembler instruction or directive, that instruction or directive is redefined to be the macro. You can return the name to its original use only by using the **.purgem** directive with the macro name.

Note: Macro names are case-sensitive.

Note: If you assemble with option **`-diab`** or option **`-eabi`**, the syntax for the macro heading is slightly different. The comma between the name of the macro and the first parameter is optional; that is, the following is acceptable:

```
.macro min_max num1, num2
```

See §4.2: *Assembler Directive Reference* for a discussion of the **`.macro`** directive.

3.1.2 Macro Body

The macro body begins with the first assembly-language statement following the **`.macro`** directive.

The name of any formal parameter you specified with the **`.macro`** directive can appear in any field in the macro body. If the name of the parameter is embedded in alphanumeric text or in a character string, it must be escaped from the surrounding text with an ampersand preceded by a backslash (`\&`).

For example, if your macro has a parameter `parm1`, the assembler recognizes `parm1` in each of the following contexts:

```
labl\&parm1: .word   blk\&parm1\&fp
             .print "parm1 = \&parm1\&"
```

At assembly time, the assembler first inserts the macro body in place of the macro call, then replaces each recognized reference to a parameter with the actual value passed to the parameter.

Note: Macro parameters that occur in comments are not expanded.

3.1.2.1 Symbol $macro

The symbol `$macro` in a macro definition is replaced at assembly time by the name of the macro currently being expanded. For example, the following macro definition:

```
.macro fred
   .ascii $macro
.endm
```

expands to this at assembly time:

```
.ascii "fred"
```

By default, `$macro` expands as a string. To interpret `$macro` as a symbol instead of as a string, precede the expansion with an escape sequence (\&), as follows:

```
.macro fred
   .long \&$macro
.endm
```

The preceding macro expands to this at assembly time:

```
.long fred
```

Using $macro

Symbol `$macro` is useful primarily when making macro constructors. For example, this macro will not work as expected, due to macro expansion rules:

```
.macro make_macro,arg
   .macro arg
      .ascii  "\&arg"
   .endm
.endm
```

This macro creates a macro constructor, `make_macro`. The following call to `make_macro`:

```
make_macro  fred
```

creates a new macro `fred`, which you would expect to expand to this:

```
.ascii  "fred"
```

However, this is the actual result:

```
.ascii  "arg"
```

Here is the same example, using symbol `$macro`:

```
.macro make_macro,arg
   .macro arg
      .ascii  "\&$macro"
   .endm
.endm
```

In this case, the macro created by this call to `make_macro`:

```
make_macro  fred
```

creates a new macro `fred`, which correctly expands to this:

```
.ascii  "fred"
```

3.1.2.2 Symbol $narg

The symbol `$narg` used in a macro definition is replaced at assembly time by the number of actual parameters used in a macro call; for example:

```
; Macro INST handles instructions with 0 to 4
; arguments.
.macro INST, mnemonic, arg1, arg2, arg3, arg4
   .if $narg==1
      mnemonic
   .elseif $narg==2
      mnemonic arg1
   .elseif $narg==3
      mnemonic arg1 arg2
   .elseif $narg==4
      mnemonic arg1 arg2 arg3
   .elseif $narg==5
      mnemonic arg1 arg2 arg3 arg4
   .endif
.endm
```

3.1.3 Macro Terminator

The directive **`.endm`** terminates the macro definition. See §4.2: *Assembler Directive Reference* for a discussion of this directive.

3.1.4 Redefining Macros

You can redefine one or more macros at any point in the program, but first you must purge any earlier definitions with the **`.purgem`** directive (see §4.2: *Assembler Directive Reference* for a discussion of the **`.purgem`** directive).

If you redefine a macro without first purging the earlier definition, the assembler emits an error message and aborts.

3.2 Nesting and Suppressing Macros

A macro definition is *nested* if it occurs entirely inside the body of another macro definition. A nested macro is defined when a call is made to the surrounding macro.

You can use a nested macro to redefine the surrounding macro, by preceding the new definition with a **`.purgem`** directive. This is most often done in a conditional situation, where you do not want the macro to operate if certain conditions are present.

You can suppress macro expansion at any point in the macro body with the **`.exitm`** directive. Any code occurring after this directive is not included in the macro expansion. See §4.2: *Assembler Directive Reference* for a discussion of the **`.exitm`** directive.

3.3 Calling a Macro

To call a macro, insert the macro name in the opcode field of an assembly statement. Macro names are used like assembler directives, except that they are not preceded by a period:

```
min_max x, y
```

Note:	Calls to other macros can occur within the body of a macro, but recursive calls to the macro itself cannot.

3.3.1 Arguments

Arguments in a macro call (also called actual parameters) are separated from one another by commas, and from the macro name by whitespace.

When a macro is expanded, each argument is passed as a character string into the statements of the macro definition, replacing the corresponding formal parameter.

If an argument contains a comma or semicolon, you must enclose the argument in angle brackets (<>).

If you omit an argument from the macro call, that argument defaults to a null string when the macro is expanded.

Here is an example of a macro definition with three formal parameters, followed by calls to the macro with arguments of various sorts:

```
.macro    mac, a, b, c
   add    a, b, c
.endm

mac       lr1, lr2, 3+5
mac       lr1, lr2, <x+3; reference external>
```

3.3.2 Macro Parameter Substitution

The following rules govern macro parameter substitution:

- Formal parameter names in the macro body are replaced by actual parameter strings. That is, declare a macro as follows:
  ```
  .macro min_max, parm1, parm2
  ```
  Then call the macro with the following actual parameters:
  ```
  min_max x, y
  ```
  All instances of `parm1` in the macro body are replaced with `x`, and all instances of `parm2` are replaced with `y`.
- No parameter substitution occurs in comments.
- Formal parameter names concatenated to or embedded in other text must be separated from the adjoining text with an escape sequence (`\&`); otherwise, the assembler does not recognize them as parameters.
- At macro expansion, all single `\&` sequences are treated as concatenation characters. Once the parameters they delimit are processed, the `\&` sequence vanishes.
- Two `\&` sequences become a single `\&` sequence, which is then treated as a break character for purposes of substitution.
- Within quoted strings, substitution occurs only if the parameter name is preceded and followed by `\&` sequences.

Note: The PowerPC EABI Committee decided to use the sequence `><` instead of `\&`. These operators are synonymous.

Assembler Directives 4

This chapter discusses assembler directives supported by the ELF assembler; it contains the following sections:

Overview The ELF assembler supports the directives listed alphabetically in §4.2: *Assembler Directive Reference*. With assembler directives, also called *pseudo operations*, you can control program organization and manipulate data.

4.1 Assembler Directives Listed by Operation

The tables in this section list assembler directives according to the type of operation they perform:

Directives in a given category are listed alphabetically.

Table 4.1 Directives for Conditional Assembly

Directive	What It Does
.else	Indicate alternative code to be assembled if corresponding **.if*** condition is `false`.
.elsec	Indicate alternative code to be assembled if corresponding **.if*** condition is `false`.
.elseif	Indicate code to be assembled if conditional expression is `true` and corresponding **.if*** condition is `false`.
.endc	Terminate conditional block.
.endif	Terminate conditional block.
.if	Indicate code to be assembled if conditional expression is `true`.
.ifdef	Indicate code to be assembled if an identifier is defined.
.ife	Indicate code to be assembled if conditional expression is `true`.
.ifeq	Indicate code to be assembled if conditional expression is `true`.
.ifeqs	Indicate code to be assembled if two strings are equal.
.ifn	Indicate code to be assembled if conditional expression is `false`.
.ifndef	Indicate code to be assembled if an identifier is not defined.
.ifne	Indicate code to be assembled if conditional expression is `false`.
.ifnes	Indicate code to be assembled if two strings are not equal.
.ifnotdef	Indicate code to be assembled if an identifier is not defined.

Table 4.2 Directives for Data-Storage Declaration

Directive	What It Does
.2byte	Store initialized 16-bit value(s) (half words) in current section.
.3byte	Store initialized 24-bit value(s) in current section.

Directive	What It Does
`.4byte`	Store initialized 32-bit value(s) (full words) in current section.
`.align`	Advance current location counter to specified boundary.
`.ascii`	Place string(s) without terminating null character in current section.
`.asciz`	Place string(s) with terminating null character in current section.
`.block`	Generate a block of initialized or uninitialized bytes.
`.byte`	Store initialized eight-bit value(s) (bytes) in current section.
`.double`	Store double-precision floating-point constant(s) in current section.
`.endian`	Change byte order of generated code.
`.even`	Advance current location counter to an even two-byte boundary.
`.float`	Store single-precision floating-point constant(s) in current section.
`.half`	StoreStore initialized 16-bit value(s) (half words) in current section.
`.long`	Store initialized 32-bit value(s) (full words) in current section.
`.short`	Store initialized 16-bit value(s) (half words) in current section.
`.skip`	Generate a block of initialized or uninitialized bytes.
`.space`	Generate a block of initialized or uninitialized bytes.
`.string`	Place string(s) with terminating null character in current section.
`.word`	Store initialized 32-bit value(s) (full words) in current section.

Table 4.3 Directives for Listing Control

Directive	What It Does
`.blank`	Insert blank lines in source-code listing.
`.eject`	Advance listing to top of page.

Directive	What It Does
`.lflags`	Set listing flags.
`.list`	Enable source-code listing.
`.nolist`	Disable source-code listing.
`.page`	Advance listing to top of page.
`.sbttl`	Specify subtitle for source-code listing.
`.title`	Specify main title for source-code listing.

Table 4.4 Directives for Macro Definition

Directive	What It Does
`.define`	Define a macro variable.
`.endm`	Terminate macro definition.
`.exitm`	Terminate macro expansion.
`.macro`	Declare macro name and parameters.
`.purgem`	Discard current macro definition.
`.undef`	Undefine one or more macro variables.

Table 4.5 Directives for Repeat Blocks

Directive	What It Does
`.endr`	Terminate repeat block.
`.irep`	For each item listed, assemble a repeat block and replace identifier with item.
`.irepc`	For each character in a string, assemble a repeat block and replace identifier with character.
`.rep`	Assemble a repeat block the specified number of times.
`.rept`	Assemble a repeat block the specified number of times.

Table 4.6 Directives for Section Specification

Directive	What It Does
`.bss`	Change current section to `.bss`.

Directive	What It Does
.comm, .common	Define common block (uninitialized block of storage).
.data	Change current section to default `.data` section.
.fini	Start a text section named `.fini`.
.init	Start a text section named `.init`.
.lcomm, .lcommon	Define local uninitialized block of storage.
.org	Set location counter of a section
.popsect	Pop section stack; restore most recently pushed section.
.previous	Resume prior section.
.pushsect	Push current section onto section stack; switch to new section.
.rdata	Change current section to default read-only data section.
.rodata	Change current section to default read-only data section.
.rodata1	Change current section to a secondary read-only data section.
.sbss	Change current section to `.sbss`.
.sbss2	Change current section to `.sbss2`.
.sdata	Change current section to secondary data section.
.sdata2	Change current section to secondary data section.
.sectflag	Set the `SHF_*` flags field of the specified section.
.section	Define control section and type.
.sectlink	Set the link field of one section to point to another section.
.seg	Define control section and type.
.text	Change current section to default `.text` section.

Table 4.7 Directives for Specifying Processor and Instruction Set

Directive	What It Does
`.machine`	Specify valid picoJava instruction set(s).

Table 4.8 Directives for Symbol Declaration and Binding

Directive	What It Does
`.eflags`	Bitwise OR a value with the `e_flags` field of the ELF header.
`.entry`	Set the `ENTRY` ELF assembler binding.
`.equ`	Assign a value to an identifier.
`.extern`	Designate a symbol as external.
`.global`, `.globl`	Export symbol(s).
`.reloc`	Specify relocation of the next instruction in the current section.
`.set`	Assign a value to an identifier.
`.weak`	Specify weak ELF assembler binding.

Table 4.9 Miscellaneous Directives

Directive	What It Does
`.assert`	Print an error message if assertion fails.
`.end`	Terminate assembly.
`.err`	Print an error message.
`.file`	Specify a source-file name.
`.ident`	Place string(s) into comment section of the object file.
`.include`	Include specified source file.
`.line`	Identify line number.
`.option`	Specify an assembly option
`.print`	Print string to standard output.
`.size`	Specify size of a symbol in bytes.

Directive	What It Does
.type	Specify a type.
.version	Place string(s) in comment section of the object file.
.warn	Print a warning message.

4.2 Assembler Directive Reference

The rest of this chapter describes individual assembler directives. The directives are listed alphabetically.

Names of assembler directives are not case-sensitive.

Preprocessor directives

Note: Preprocessor directives are processed by the macro preprocessor before assembly.

.2byte *expression*[, *expression*, ...] — **Store initialized 16-bit value(s) (half words) in current section**

Stores initialized half-word values (16-bit two's-complement) in the current section.

Assembles *expression* arguments into consecutive half words. For example, this directive initializes five consecutive half words; the first half-word has the value of expression `val_1`, the second has the value of `val_2`, and so on:

```
.2byte val_1, val_2, val_3, val_4, val_5
```

.2byte allows misalignment, but the assembler warns.

This directive is not valid for the `.bss` section.

Synonym for **.half** and **.short**.

.3byte *expression*[, *expression*, ...] — **Store initialized 24-bit value(s) in current section**

Stores initialized values (24-bit two's-complement) in the current section.

Assembles *expression* argument(s) into consecutive 24-bit blocks. For example, the following directive initializes four consecutive 24-bit blocks;

the first block has the value of expression `val_1`, the second has the value of `val_2`, and so on:

```
.3byte val_1, val_2, val_3, val_4
```

`.3byte` allows misalignment, but the assembler warns.

This directive is not valid for the `.bss` section.

`.4byte` *`expression`*`[, expression, ...]` — **Store initialized 32-bit value(s) (full words) in current section**

Stores initialized values (32-bit two's-complement) in the current section.

Assembles *`expression`* argument(s) into consecutive 32-bit blocks. For example, the following directive initializes four consecutive 32-bit blocks; the first block has the value of expression `val_1`, the second has the value of `val_2`, and so on:

```
.4byte val_1, val_2, val_3, val_4
```

`.4byte` allows misalignment, but the assembler warns.

This directive is not valid for the `.bss` section.

Synonym for **`.long`**.

`.align` *`number`* — **Advance location counter to specified boundary**

Advances the location counter to the boundary specified by *`number`*, where *`number`* is log base 2 of the alignment; for example:

`.align` `1`	aligns to a two-byte boundary
`.align` `2`	aligns to a four-byte boundary

`.align` `1` is the same as **`.even`**.

`.ascii` *`string`*`[, string, ...]` — **Place string(s) without terminating null character in current section**

Stores *`string`*(s) without a terminating null (`\0`) character in the current section. Each string must be enclosed in double quotes.

The **`.ascii`** directive leaves the location counter positioned after the last character in the string, whether or not it is on the current byte boundary. This means that if the next directive is **`.block`**, **`.double`**, **`.float`**, **`.half`**, or **`.word`**, the assembler emits a warning that the data item will be misaligned.

It is a good idea to follow a **.ascii** directive with a **.align** directive to force proper alignment of the next data item to be stored.

This directive is not valid for the .bss section.

.asciz *string*[*, string, ...*] — **Place string(s) with terminating null character in current section**

Places *string*(s) in the current section followed by a terminating null (\0) character.

This directive is not valid for the .bss section.

Synonym for **.string**.

.assert *expression* **— Print an error message if assertion fails**

Evaluates *expression*; if *expression* evaluates to 0 (false), generates an error message.

.blank *expression* — **Insert blank lines in source-code listing**

Tells the assembler to insert the number of blank lines specified by *expression* in the source listing. *expression* must evaluate to an absolute integer value.

Directive **.blank** works only if the assembler was invoked with option **-l**.

.block *number* — **Generate a block of initialized or uninitialized bytes**

Skips *number* bytes in the current section.

- If the section is a .bss data section, the bytes are left uninitialized.
- If the section is a text section, **nop** instructions are placed in the bytes.
- In all other sections, zeros are placed in the bytes.

This directive is valid for the .bss section.

Synonym for **.skip** and **.space**.

.bss — **Change current section to .bss**

Changes the current section to .bss, the default BSS section.

`.byte` *`expression`*`[, expression, ...]` — **Store initialized eight-bit value(s) (bytes) in current section**

Stores initialized bytes in the current section. For example, the following directive initializes three consecutive bytes; the first byte has the value of expression `val_1`, the second has the value of `val_2`, and so on:

```
.byte val_1, val_2, val_3
```

This directive is not valid for the `.bss` section.

`.comm` *`name, expression`*`[, align]`
`.common` *`name, expression`*`[, align]` — **Define a common block (uninitialized block of storage)**

Defines an uninitialized block of storage, called a *common block*, in the `.data` section. This block can be common to more than one module.

`name`	block name; references the block of storage
`expression`	block size, in bytes; must be a positive integer
`align`	specifies the alignment, which must be a positive power of 2; see **`.align`**

This directive is valid for the `.bss` section.

`.data` — **Change current section to default `.data` section**

Changes the current section to `.data`, which is the default data section.

`.define` *`name`*`[, {`*`string`*` | `*`integer`*`}]` — **Define a macro variable**

Preprocessor directive

Defines a macro variable (*`name`*) for use during macro processing. The value of *`name`* can be either a character string or an integer constant. If you do not specify a string or integer, the value of *`name`* defaults to `1` (one). *`name`* is global to the whole program.

During macro processing, the preprocessor replaces occurrences of the macro variable *`name`* in the source code with its defined value.

To redefine *`name`*, first undefine it with an **`.undef`** directive, then use it in another **`.define`** directive. (The assembler emits a warning if you redefine an already defined macro variable.)

`.double` *`floating_constant`* **`[`**`, `*`floating_constant`* **`...]`** — **Store double-precision floating-point constant(s) in current section**

Stores one or more double-precision floating-point constants in the current section. *`floating_constant`* is converted to floating-point if necessary.

`.double` allows misalignment, but the assembler warns.

This directive is not valid for the `.bss` section.

`.eflags` *`expression`* — **Bitwise OR a value with the `e_flags` field of the ELF header**

The **`.eflags`** directive directs the assembler to evaluate *`expression`*, and comine the result, using a logical OR operation, with the `e_flags` field of the ELF hearder. The `e_flags` field holds processor-specific flags associated with the objet file.

`.eject` — **Advance listing to top of page**

Tells the assembler to move to the top of the next page in the source listing form. For example, you can use **`.eject`** to start the listing of each subroutine on a new page.

Directive **`.eject`** works only if the assembler was invoked with option **`-l`**.

Synonym for **`.page`**.

`.else` — **Indicate code to be assembled if corresponding `.if*` condition is `false`**

Preprocessor directive

Indicates code to be assembled if the conditional expression of the corresponding **`.if*`** directive evaluates to `false` (zero). In that case, the assembler assembles the code following the **`.else`** directive instead of the code following the **`.if*`** directive.

Synonym for **`.elsec`**.

`.elsec` — **Indicate code to be assembled if corresponding `.if*` condition is `false`**

Preprocessor directive

Synonym for **`.else`**.

`.elseif` *`conditional_expression`* — Indicate code to be assembled if conditional expression is `true` and corresponding `.if*` condition is `false`

Preprocessor directive

Indicates that all code between the **`.elseif`** directive and the corresponding **`.else`**, **`.elseif`**, or **`.endif`** directive is to be assembled if both of the following conditions are met:

- The conditional expression of the corresponding **`.if*`** directive evaluates to `false` (zero).
- *`conditional_expression`* evaluates to `true` (non-zero).

Otherwise, the next **`.else`**, **`.elseif`**, or **`.endif`** directive is processed.

`.end` — Terminate assembly

Although the assembler accepts directive **`.end`**, it performs no operation.

`.endc` — Terminate conditional block

Preprocessor directive

Marks the end of a conditional block. If you nest **`.if*`** directives, an **`.endc`** (or **`.endif`**) directive is paired with the most recent **`.if*`** directive.

Synonym for **`.endif`**.

`.endian` {`big`|`little`} — Change byte order of generated code

Enables you to change the byte order of generated code by specifying either big-endian or little-endian mode.

`.endif` — Terminate conditional block

Preprocessor directive

Synonym for **`.endc`**.

`.endm` — Terminate macro definition

Preprocessor directive

Marks the end of a macro definition begun with the previous **`.macro`** directive. An **`.endm`** directive is paired with the most recent **`.macro`** directive. Therefore, if another **`.macro`** directive occurs before the **`.endm`** directive, the macro initiated by the second **`.macro`** directive is nested inside the first macro; it must be terminated by an **`.endm`** directive of its own. See §3.2: *Nesting and Suppressing Macros* for a discussion of nested macros.

`.endr` — Terminate repeat block

Preprocessor directive

Terminates a repeat block initiated by the **`.rep`**, **`.irep`**, or **`.irepc`** directive.

`.entry` *`name`*`[, name ...]` — **Set the `ENTRY` ELF binding**

Sets the `ENTRY` ELF binding for a symbol.

`.equ` *`name, expression`*
`name``: [`*`name`*`: ...]` **`.equ`** *`expression`* — **Assign a value to an identifier**

Assigns the value of *`expression`* to *`name`*. *`expression`* is an absolute or relocatable value.

This is the same as using an assignment operator:

```
name = expression
```

If you use the second form of **`.equ`**, you can assign *`expression`* to more than one identifier:

```
name_1: name_2: name_3: .equ expression
```

A given identifier should appear in only one **`.equ`** statement, because the **`.equ`** assignment is constant for the whole program.

`.err` `["`*`string`*`"]` — **Print an error message**

Causes the assembler to print an error message to `stderr`, or to `stdout` if you have specified option **`-Eo`**. Also increments the error count.

You can optionally specify a string to include in the error message. The string must be enclosed in double quotes.

`.even` — Advance location counter to an even two-byte boundary

Advances the location counter to the next even two-byte boundary.

Same as **`.align`** `1`.

`.exitm` — Terminate macro expansion

Preprocessor directive

Causes macro expansion to stop and all code between the **`.exitm`** directive and the **`.endm`** directive of the macro to be ignored. The **`.exitm`** directive is generally used with a **`.if*`** directive to test for a particular condition and abort macro expansion if the condition occurs.

The **.exitm** directive terminates expansion of only the macro in which it appears. If macros are nested, **.exitm** returns code generation to the previous nesting level.

.extern *name* — **Designate a symbol as external**

Identifies a symbol defined in an external module. Because undefined symbols are assumed to be external symbols, you do not need to use this directive.

.file *name* — **Specify a source-file name**

Identifies the name of a source file.

.fini — Start a text section named .fini

Starts a text section named .fini.

.float *floating_constant*[, *floating_constant,* **...**] — **Store single-precision floating-point constant(s) in current section**

Stores one or more single-precision floating-point constants in the current section. *floating_constant* is converted to floating-point if required.

.float allows misalignment, but the assembler warns.

This directive is not valid for the .bss section.

.global *name*[, *name* **...**]
.globl *name*[, *name* **...**] — **Export symbol(s)**

Exports one or more *name* symbols.

.half *expression*[, *expression,* **...**] — **Store initialized 16-bit value(s) (half words) in current section**

Synonym for **.2byte** and **.short**.

.ident *string*[, *string,* **...**] — **Place string(s) in comment section of the object file**

Places one or more strings in the comment section of the object file.

Synonym for **.version**.

`.if` *`conditional_expression`* — **Indicate code to be assembled if conditional expression is `true`**

Preprocessor directive

Indicates that all code between the **`.if`** directive and the corresponding **`.else`**, **`.elseif`**, or **`.endif`** directive is to be assembled if *`conditional_expression`* evaluates to `true` (non-zero). Otherwise, the next **`.else`**, **`.elseif`**, or **`.endif`** directive is processed.

Synonym for **`.ife`**, **`.ifeq`**.

`.ifdef` *`name`* — **Indicate code to be assembled if an identifier is defined**

Preprocessor directive

Indicates that all code between the **`.ifdef`** directive and the corresponding **`.else`**, **`.elseif`**, or **`.endif`** directive is to be assembled if *`name`* is defined. If *`name`* is not defined, the next **`.else`**, **`.elseif`**, or **`.endif`** directive is processed. *`name`* can be either an assembler variable or a macro variable defined with the **`.define`** directive.

`.ife` *`conditional_expression`* — **Indicate code to be assembled if conditional expression is `true`**

Preprocessor directive

Synonym for **`.if`**, **`.ifeq`**.

`.ifeq` *`conditional_expression`* — **Indicate code to be assembled if conditional expression is `true`**

Preprocessor directive

Synonym for **`.if`**, **`.ife`**.

`.ifeqs` `"`*`string1`*`", "`*`string2`*`"` — **Indicate code to be assembled if two strings are equal**

Preprocessor directive

Indicates that all code between the **`.ifeqs`** directive and the corresponding **`.else`**, **`.elseif`**, or **`.endif`** directive is to be assembled if *`string1`* is equal to *`string2`*. If the strings are not equal, the next **`.else`**, **`.elseif`**, or **`.endif`** directive is processed. The character strings must be enclosed in double quotes.

`.ifn` *`conditional_expression`* — **Indicate code to be assembled if conditional expression is `false`**

Preprocessor directive

Indicates that all code between the **`.ifn`** directive and the corresponding **`.else`**, **`.elseif`**, or **`.endif`** directive is to be assembled if *`conditional_expression`* evaluates to `false` (zero). Otherwise, the next **`.else`**, **`.elseif`**, or **`.endif`** directive is processed.

Synonym for **`.ifne`**.

`.ifndef` *`name`* — **Indicate code to be assembled if an identifier is not defined**

Preprocessor directive

Indicates that all code between the **`.ifndef`** directive and the corresponding **`.else`**, **`.elseif`**, or **`.endif`** directive is to be assembled if *`name`* is not defined. If *`name`* is defined, the next **`.else`**, **`.elseif`**, or **`.endif`** directive is processed. *`name`* can be either a regular assembler variable or a macro variable defined with the **`.define`** directive.

Synonym for **`.ifnotdef`**.

`.ifne` *`conditional_expression`* — **Indicate code to be assembled if conditional expression is `false`**

Preprocessor directive

Synonym for **`.ifn`**.

`.ifnes` "*`string1`*", "*`string2`*" — **Indicate code to be assembled if two strings are not equal**

Preprocessor directive

Indicates that all code between the **`.ifnes`** directive and the corresponding **`.else`**, **`.elseif`**, or **`.endif`** directive is to be assembled if *`string1`* is not equal to *`string2`*. If the strings are equal, the next **`.else`**, **`.elseif`**, or **`.endif`** directive is processed. The character strings must be enclosed in double quotes. Use this directive inside macros to test macro parameters.

`.ifnotdef` *`name`* — **Indicate code to be assembled if an identifier is not defined**

Preprocessor directive

Synonym for **`.ifndef`**.

`.include "[`*`pathname`*`]`*`file_name`*`"` — Include specified source file

Preprocessor directive

Instructs the assembler to include the specified source file in the input source-code stream at assembly time. The file name and the pathname, if any, must be enclosed in double quotes.

If you assemble with command-line option **`-I`**, the assembler looks for **`.include`** files with non-absolute pathnames first in the current directory, then in the *`pathname`* directory. If the assembler cannot find the specified source file, it emits an error message and aborts.

The assembler passes the *`pathname`* and *`file_name`* specifications to the host operating system without any conversion from lowercase to uppercase.

Note: Included source files can contain **`.include`** directives of their own, as can macros.

Note: *`file_name`* should be in the form described in §2.3.3: *String Constants*. We recommend using forward slashes as path separators, because they are supported on all platforms. However, you can use backslashes, as long as you use two backslashes to represent each single backslash in the pathname.

`.init` — Start a text section named `.init`

Starts a text section named **`.init`**.

`.irep` *`identifier, item`*`[, item ...]` — For each item listed, assemble a repeat block and replace *`identifier`* with *`item`*

Preprocessor directive

Tells the assembler to assemble instructions up to the next **`.endr`** directive once for each *`item`* listed. On each pass, the corresponding *`item`* replaces

identifier in the instruction sequence. Observe the following syntax requirements:

- Separate any occurrence of *identifier* from adjacent text with a \& (backslash-ampersand) sequence; for example:

```
              .irep  init_val, 10, 20, 30
l\&init_val:  .2byte  init_val
              .endr
```

  The assembler converts this instruction sequence to the following:

```
l10:          .2byte   10
l20:          .2byte   20
l30:          .2byte   30
```

- If *item* contains a comma or semicolon, enclose *item* in angle brackets, <like;this>.

.irepc *identifier*, "*string*" — **For each character in a string, assemble a repeat block and replace *identifier* with character**

Preprocessor directive

Tells the assembler to assemble instructions up to the next **.endr** directive once for each character in *string*. *string* must be enclosed in double quotes. On each pass, the corresponding character replaces *identifier* in the instruction sequence.

Separate any occurrence of *identifier* from adjacent text with a \& (backslash ampersand) sequence; for example:

```
              .irepc ch, "XYZ"
l_\&ch:       .byte '\&ch\&'
              .endr
```

The assembler converts this instruction sequence to the following:

```
l_X:          .byte 'X'
l_Y:          .byte 'Y'
l_Z:          .byte 'Z'
```

If *string* is empty (""), no instruction sequences are assembled.

.lcomm *name*, *expression*[, *align*]
.lcommon *name*, *expression*[, *align*] — **Define local uninitialized block of storage**

Defines a local uninitialized block of storage in the `.bss` section. The result defines *name* as a `bss` symbol.

name	block name; references the storage, cannot be predefined
expression	block size; must be a positive integer
align	specifies the alignment; must be a positive integer, log base 2 of the alignment

This directive is valid for the `.bss` section.

.lflags[*n*]*flag*[*arg*][[*n*]*flag*[*arg*]...] — **Set listing flags**

Sets listing flags that control the listing of the source file. See the description of option **-f** in §1.2.2: *Command-Line Option Reference* for these flags and their default settings. **.lflags** performs the same function as command-line option **-f**, except that you can apply the directive to portions of a module, whereas option **-f** affects the entire module.

To use the **.lflags** directive, you must also specify option **-l** on the command line.

Setting and Toggling Listing Flags

To turn `On` a flag, list it after the **.lflags** directive. To turn `Off` a flag, list it after the **.lflags** directive and put an `n` before the flag. To set the `l` and `p` flags, insert an integer value after them in the flag list.

For example, the following **.lflags** directive:

```
.lflags cgnmp66
```

tells the assembler to do the following:

- Turn `On` flags `c` and `g`.
- Turn `Off` flag `m`.
- Set the value of flag `p` to `66`.

.line *number* — **Identify line number**

Identifies a line number.

The assembler accepts this directive but ignores it.

`.list` — Enable source-code listing

Tells the assembler to output a source-code assembly listing. Every program begins with an implicit **`.list`** directive, but the assembler generates the listing only if you also specify assembler command-line option **`-l`**.

Alternate the **`.list`** directive with the **`.nolist`** directive to list selected portions of a program.

`.long` *`expression`*`[, expression, ...]` — Store initialized 32-bit value(s) (full words) in current section

Synonym for **`.4byte`**.

`.machine[`*`inst_set_name`*`]` — Specify valid PowerPC instruction set(s)

Specifies which instruction set(s) the assembler considers to be acceptable. *`inst_set_name`* can be any of the following values:

`400`	Accept only unprivileged PowerPC 400 series instructions.
`400priv`	Accept all PowerPC 400 series instructions (unprivileged and privileged).
`602`	Accept PowerPC 602 instructions.
`all`	Accept all assembler instructions known to the PowerPC family of processors.

If you specify `400`, the assembler considers any opcode not part of the unprivileged PowerPC 400 series instructions to be unacceptable. On encountering such an opcode, the assembler generates an error. Similarly, if you specify `400priv`, the assembler generates an error on encountering any non-400 series opcode.

`inst_set_name` can be either an identifier or a quoted string.

The default *`inst_set_name`* value is `all`.

`.macro` *`name`*`, `*`param`*`[, `*`param`*`...]` — Declare macro name and parameters

Preprocessor directive

Declares the name and formal parameters of a macro and marks the beginning of the macro definition. Code following the **`.macro`** directive, down to the corresponding **`.endm`** directive, constitutes the body of the macro *`name`*.

Note: Macro names are case-sensitive.

The names of the formal parameters are recognized only within the macro definition. These names can be used for other purposes outside the macro.

The actual parameters in the call to the macro are matched to the formal parameters in the macro declaration, starting with the left-most of each. There can be fewer actual parameters than formal parameters. Formal parameters that lack a corresponding actual parameter default to the null string.

Note: If you assemble with option **`-diab`**, the syntax for the macro heading is slightly different; the comma between the name of the macro and the first parameter is optional; for example, both of the following work equally well:

```
.macro min_max, num1, num2
.macro min_max num1, num2
```

`.nolist` — Disable source-code listing

Disables assembly source-code listing, except for lines flagged with errors.

Directive **`.nolist`** works only if the assembler was invoked with option **`-l`**.

`.option {"option"}` — Specify an assembly option

Enables you to specify an assembly option. These are the valid arguments:

`percent_reg`	Specify that register names must begin with a percent sign.
`%reg`	Specify that register names must begin with a percent sign.
`svr4`	Adhere to SVR4 syntax.

See §1.2: *Command-Line Options* for more detailed information about individual options.

Note: Options set with directive **`.option`** take precedence over options set at the command line.

`.org` *`address`* — Set location counter of a section

Sets the location counter of a section. If you invoke directive **`.org`** before anything has been stored in the section, the ELF section address is set to *`address`*. Otherwise, the section location counter is set to *`address`*, and the space between the stored data and the location counter is initialized.

`.page` — Advance listing to top of page

Synonym for **`.eject`**.

`.popsect` — **Pop section stack; restore most recently pushed section**

Pops the section stack, making the current section be the section most recently pushed on the section stack.

`.previous` — **Resume prior section**

Resumes the section that was active prior to the current section.

`.print [`*`"string"`*`]` — Print string to standard output

Causes the assembler to print a string to standard output. The string must be enclosed in double quotes. *`"string"`* is optional; if you do not specify a string, directive **`.print`** outputs a newline.

`.purgem` *`name`*`[`*`, name ...`*`]` — Discard current macro definition

Preprocessor directive

Discards current macro definition of all macros listed as arguments. Macros are expanded for all calls up to the **`.purgem`** directive.

`.pushsect` *`name`* — Push current section onto section stack; switch to new section

Pushes the current section onto a section stack and switches the current section to *`name`*.

`.rdata` — **Change current section to default read-only data section**

Changes the current section to `.rodata`, the default read-only data section.

Synonym for **`.rodata`**.

`.reloc` *`symbol, reltype`* — Specify relocation for the next word in the current section

Specifies the relocation type of the next word to be assembled in the current section. **`.reloc`** adds a relocation entry to the relocation table, using the

identifier *symbol*. The relocation type is designated by *reltype*, an integer value.

.rep *n* — **Assemble a repeat block the specified number of times**

Preprocessor directive

Tells the assembler to duplicate an instruction sequence ending with the next **.endr** directive the number of times represented by *n*. *n* must evaluate to an absolute integer value.

Synonym for **.rept**.

.rept *n* — **Assemble a repeat block the specified number of times**

Macro assembler directive

Synonym for **.rep**.

.rodata — **Change current section to default read-only data section**

Synonym for **.rdata**.

.rodata1 — **Change current section to secondary read-only data section**

Changes the current section to .rodata1, a secondary read-only data section.

.sbss — **Change current section to .sbss**

Changes the current section to .sbss, a secondary BSS section.

.sbss2 — **Change current section to .sbss2**

Changes the current section to .sbss2, a secondary BSS section.

.sbttl "*subtitle*" — **Specify subtitle for source-code listing**

Specifies a subtitle for the source-code listing. The subtitle string must be enclosed in double quotes. The subtitle appears below the main title at the top of each page of the source listing. When you specify a new subtitle, it appears on the page immediately following the page that contains the **.sbttl** directive.

Directive **.sbttl** works only if the assembler was invoked with option **-l**.

.sdata — **Change current section to a secondary data section**

Changes the current section to .sdata, a secondary data section.

`.sdata2` — Change current section to a secondary data section

Changes the current section to `.sdata2`, a secondary data section.

`.sectflag` *`section_name, flag`* — Set the `SHF_*` flags field of the specified section

Sets the `SHF_*` flags field of the specified section *`section_name`*. The recognized values of *`flag`* are as follows:

`begin`	sets the `SHF_BEGIN` flag
`end`	sets the `SHF_END` flag

`section_name` and *`flag`* can be either identifiers or quoted strings.

`.section` *`name`*`[`, *`class`*`][`, *`entsize`*`]` — Define control section, type, and size of entry

Defines a control section *`name`* of type *`class`*, and arranges for subsequent code to be placed within the control section.

Once the section has been defined, you can reactivate it at a later time by respecifying the **`.section`** directive. The attributes are unnecessary when you reactivate the section.

`class` is one of the following attributes:

`bss`	The section contains writable data initialized to `0` (zero).
`comdat`	The section contains common code or data.
`data`	The section contains writable data.
`directive`	The section contains additional linker command-line arguments.
`lit`	The section contains read-only data.
`note`	The section contains special information for the operating system.
`os`	The section contains special OS data.
`rodata`	The section contains read-only data.
`text`	The section contains executable instructions.

`class` can also be a string of characters, used singly or in combination, with the following meanings:

`a`	The section occupies memory during execution.
`C` or `c`	The section is allocatable and executable.
`D` or `d`	The section is allocatable and writable.

`M` or `m`	The section is allocatable.
`N` or `n`	The section is a non-allocated data section; it is not loaded to target memory. Usually used to create a debug section.
`R` or `r`	The section is allocatable.
`w`	The section is writable.
`x`	The section is executable.

For example, `aw` is equivalent to `data` and `ax` is equivalent to `text`.

The *`entsize`* argument is an integer indicating the size in bytes of entries in the section. Specify *`entsize`* if the section contains a table of entries of fixed size. *`entsize`* is used to set the `sh_entsize` field of the section in the ELF section header.

When you invoke the assembler with option **`-diab`**, the second argument of directive **`.section`** specifies the alignment rather than the class, and the third argument specifies the class rather than the entry size.

```
.section name
.section name, alignment
.section name, alignment, class
.section name, , class
```

If you omit the *`alignment`* argument, as in the last example, alignment defaults to `1`. However, the linker may override the alignment you specify, if it determines that the section requires a larger alignment.

Synonym for **`.seg`**.

`.sectlink` *`section_a, section_b`* — **Set the link field of one section to point to another section**

Sets the link field of *`section_a`* to point to *`section_b`*. Both sections must have been previously defined. The section-linking feature is necessary for COMDAT support. See the **High C/C++ Language Reference** for information about COMDAT.

`.seg` *`name`***`[`***`, class`***`][`***`, entsize`***`]`** — **Define control section, type, and size of entry**

Synonym for **`.section`**.

.set *name, expression*
name: [*name*: **...**] **.set** *expression* — **Assign a value to an identifier**

Assigns the value of *expression* to *name*. *expression* is an absolute or relocatable value.

This is the same as using the assignment operator:

```
name = expression
```

If you use the second form of **.set**, you can assign expression to more than one identifier:

```
name_1: name_2: name_3: .set expression
```

.set is a dynamic assignment operator. That is, you can use it to set the value of an identifier repeatedly, instead of just once for the entire assembly. This means that an identifier's value, if it is set with **.set**, can be different at different points in the execution.

.short *expression*[, *expression*, **...**] — **Store initialized 16-bit value(s) (half words) in current section**

Synonym for **.2byte** and **.half**.

.size *name, expression* — **Specify size of a symbol in bytes**

Sets the size (in bytes) for symbol *name* to *expression*. This value is passed to the linker.

.skip *number* — **Generate a block of initialized or uninitialized bytes**

Synonym for **.block** and **.space**.

.space *number* — **Generate a block of initialized or uninitialized bytes**

Synonym for **.block** and **.skip**.

.string *string*[, *string*, **...**] — **Place string(s) with terminating null character in current section**

Synonym for **.asciz**.

.text — **Change current section to default .text section**

Changes the current section to the default .text section, which consists of executable code.

`.title` "*`main_title`*" — **Specify main title for source-code listing**

Specifies a main title for the source-code listing. The title string must be enclosed in double quotes. The title appears at the top of each page of the source listing.

The default title defined by the assembler is blank.

For the title you specify to appear on the first page of the source listing, you must make the **`.title`** directive the first statement in the program, before all other lines, including comments.

Directive **`.title`** works only if the assembler was invoked with option **`-l`**.

`.type` *`name, type`* — **Specify a type**

Associates *`type`* with *`name`*. Type information is passed to the linker.

`type` can be one of the following:

```
@function    "function"
@import      "import"
@no_type     "no_type"
@object      "object"
```

For example:

```
.type my_function, @function
```

`.undef` *`name`*[**`,`** *`name`***`...`**] — **Undefine one or more macro variables**

Preprocessor directive

Undefines one or more macro variables. If you undefine an identifier that has not been previously defined, the assembler emits a warning.

`.version` *`string`*[**`,`** *`string`***`,...`**] — **Place string(s) in comment section of the object file**

Synonym for **`.ident`**.

`.warn` [**`"`***`string`***`"`**] — **Print a warning message**

Causes the assembler to print a warning message to `stderr`, or to `stdout` if you have specified option **`-Eo`**. Also increments the warning count.

You can optionally specify a string to include in the warning message. The string must be enclosed in double quotes.

`.weak` *`name`*`[, name, ...]` — **Specify weak ELF binding**

Sets the binding of *`name`* to `weak`.

`.word` *`expression`*`[, expression, ...]` — **Store initialized 32-bit value(s) (full words) in current section**

Synonym for **`.4byte`** and **`.long`**.

Index