



ELF Assembler User's Guide

for PowerPC

92G6921

Second Edition (August 1996)

This edition of the *IBM ELF Assembler User's Guide for PowerPC* applies to IBM ELF Assembler version 2.04 and to subsequent versions until otherwise indicated in new versions or technical newsletters.

The following paragraph does not apply to the United Kingdom or any country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS MANUAL "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions; therefore, this statement may not apply to you.

IBM does not warrant that the products in this publication, whether individually or as one or more groups, will meet your requirements or that the publication or the accompanying product descriptions are error-free.

This publication could contain technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or program(s) described in this publication at any time.

It is possible that this publication may contain references to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country. Any reference to an IBM licensed program in this publication is not intended to state or imply that you can use only IBM's licensed program. You can use any functionally equivalent program instead.

No part of this publication may be reproduced or distributed in any form or by any means, or stored in a data base or retrieval system, without the written permission of IBM.

Requests for copies of this publication and for technical information about IBM products should be made to your IBM Authorized Dealer or your IBM Marketing Representative.

Address comments about this publication to:

IBM Corporation
Department YM3A
P.O. Box 12195
Research Triangle Park, NC 27709

IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Copyright © 1995-96, MetaWare® Incorporated, Santa Cruz, CA

©Copyright International Business Machines Corporation 1995-96. All rights reserved.

Printed in the United States of America.

4 3 2 1

Notice to U.S. Government Users—Documentation Related to Restricted Rights—Use, duplication, or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corporation.

Patents and Trademarks

IBM may have patents or pending patent applications covering the subject matter in this publication. The furnishing of this publication does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 208 Harbor Drive, Stamford, CT 06904, United States of America.

IBM and the IBM logo are registered trademarks of the International Business Machines Corporation.

All MetaWare product names are trademarks or registered trademarks of MetaWare Incorporated.

Other terms which are trademarks are the property of their respective owners.

Contents

About This Book	iii
Notational and Typographic Conventions.....	iii
Where to Go for More Information	iv
1 Using the Assembler.....	1
1.1 Invoking the Assembler.....	1
1.1.1 Assembler Command-Line Syntax.....	1
1.1.2 Command-Line Options	3
1.1.3 Command-Line Option Reference.....	4
1.2 Lexical Features of the ELF Assembler	7
1.2.1 The ELF Assembler Character Set	8
1.2.2 Identifiers.....	9
1.2.3 Symbols	9
1.2.3.1 Reserved Symbols.....	9
1.2.4 Labels.....	10
1.2.4.1 Regular Labels	10
1.2.4.2 Numeric Labels.....	10
1.3 The Location Counter.....	11
1.4 Constants	12
1.4.1 Integer Constants	12
1.4.2 Floating-Point (Real) Constants	12
1.4.3 String Constants.....	13
1.5 Operators and Expressions	15
1.5.1 Operators and Operator Precedence	15
1.5.2 Register Expressions.....	16
1.6 Identifier Attributes	16
1.6.1 Dynamic Relocation	19
2 Writing Assembler Macros	21
2.1 Defining a Macro.....	21
2.1.1 Redefining Macros	23
2.1.2 Nesting Macros.....	23
2.1.3 Suppressing Macro Expansion	23
2.2 Calling a Macro	24
2.2.1 Arguments	24
2.2.2 Macro Parameter Substitution	25

3	Assembler Directives	27
3.1	Quick-Reference Table of Assembler Directives	27
3.2	Assembler Directives Listed by Operation	31
3.3	Assembler Directive Reference	32
	Index	49

About This Book

This **ELF Assembler User's Guide** describes how to use the ELF (Executable and Linking Format) Assembler for PowerPC.

This guide contains all the system-specific information you need to use the assembler effectively.

Notational and Typographic Conventions

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

Convention	Meaning
<code>Courier</code>	Program text, input, output, file names.
Courier	Commands, keywords, literal options.
<i>Courier</i>	Formal parameters to be replaced by user-specified names or values; user input on the command line.
<i>Italics</i>	Special terms; term definitions.
{ x y z }	Choose one and only one of the options separated by vertical bars and enclosed in curly braces.
[x y z]	Choose none, one, some, or all of the options separated by vertical bars and enclosed in brackets.
...	Indicates multiple entries of the same type.
	Separates choices within brackets.

Where to Go for More Information

The file `readme`, in your main High C/C++ directory, identifies last-minute changes and describes any special files.

The **High C/C++ Programmer's Guide** contains all the system-specific information you need to use the High C/C++ compiler.

The **High C/C++ Language Reference** describes the syntax and semantics of the C and C++ languages, and the MetaWare High C/C++ language extensions.

If your distribution includes MetaWare libraries, see the **High C Library Reference** for descriptions of C functions, and the **High C++ I/O Streams Library Reference** for information about C++ input and output streams.

The **ELF Linker/Locator and Archiver User's Guide** describes the options and syntax of the ELF linker.

For documentation about the PowerPC architecture and the Executable and Linking Format (ELF), see *About This Book* in the **High C/C++ Programmer's Guide**.

1

Using the Assembler

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

This chapter describes how to use the ELF assembler. It covers the following topics:

§1.1: *Invoking the Assembler*

§1.2: *Lexical Features of the ELF Assembler*

§1.3: *The Location Counter*

§1.4: *Constants*

§1.5: *Operators and Expressions*

§1.6: *Identifier Attributes*

1.1 Invoking the Assembler

This section explains how to invoke the ELF assembler and how to use the assembler command-line options.

1.1.1 Assembler Command-Line Syntax

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

```
asppc [options] [-o object_file.o] source_file.s  
[> list_file]
```

The **asppc** assembler command is followed by:

- an optional series of command-line options
- the name of an optional output object file, specified with command-line option **-o**
- the name of the assembly source file being assembled
- the name of an optional list file to which the assembly-language output listing is redirected; the list-file name must be preceded by a redirection operator (**>**)

The ELF assembler command-line options are described in detail in §1.1.2.

The assembly source file has a default extension **.s**. You must specify a source file with the **asppc** command, or assembly fails.

Output file name Unless you specify otherwise with option **-o**, the assembler generates an object file with the same file name as the assembly source file, but with a default **.o** extension: *source_file.o*. You can assemble more than one source file with a single command; the assembler concatenates all the source files into one object file. For example:

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

The default name for this output file is the same as the name of the last source file, with a **.o** extension: *srcfile3.o*

An assembly-language output listing is generated only if you specify command-line option **-l**. If you specify option **-l** and do not specify a list-file name, the listing is directed to standard output.

The following example tells the assembler to assemble source file *prog1.s*, generate an assembly output listing, and redirect the listing to *prog1.lst*:

```
asppc -l prog1.s > prog1.lst
```

Preprocessing The ELF assembler includes a macro preprocessor. See Chapter 2: *Writing Assembler Macros*, for information on how to write macros, and Chapter 3: *Assembler Directives*, for macro-specific assembler directives. However, if you have source files containing C-style preprocessing directives, you must invoke the High C/C++ driver with driver command-line option **-Hasmcpp** to preprocess these files. For example:

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

1.1.2 Command-Line Options

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

You can place these options on the command line in any order, before or after the source-file name.

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

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

On-line help To see a listing of all the ELF assembler command-line options, type **asppc** at the command prompt, and press Enter.

Table 1.1 *Command-Line Options*

Option	Meaning
-be	Assembles using big-endian format
-big_si	Suppresses warning if a signed integer's value exceeds 32,767
-c	Suppresses display of the copyright message
-D name [=n]	Defines an identifier and assigns a constant value to it
-Eo	Sends error messages to standard output
-f flag [flag...]	Sets listing flags
-h	Displays command-line option help screen
-l	Generates an assembly output listing
-L	Places private labels in the symbol table
-le	Assembles using little-endian format
-o object_file	Overrides default object-file name
-pa	Performs Power (not PowerPC) assembly

Table 1.1 *Command-Line Options (Continued)*

Option	Meaning
-Qy	Places information about the assembler in object file
-v	Prints a summary of assembler statistics
-w	Suppresses warning messages

Assembler command-line options can be used either directly with the **asppc** command or with the **hcppc** driver command. To pass assembler options to the assembler via the **hcppc** 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 options.

1.1.3 Command-Line Option Reference

-be — Assemble using big-endian format

Causes the assembler to generate object code in big-endian format. Big-endian mode is the default for the ELF assembler.

-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 constant value to it

Defines identifier *name* and, if you specify a constant value *n*, assigns *n* to *name*. If you specify *n*, *n* must be an integer; otherwise, the value of *name* defaults to 1 (one).

This is the same as putting the directive `.equ name, n` on the first line of the assembly source file.

-Eo — Send error messages to standard output

Causes error messages to be written to standard output instead of to `stderr`.

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

Sets listing flags that control the contents and appearance of the assembly source listing. Table 1.2 shows these flags and their default settings.

Table 1.2 Assembler Listing Flags

Flag	Function	Default
c	Lists instructions not assembled because of conditional assembly statements	Off
g	Lists local label symbols in the symbol and cross-reference tables	Off
i	Lists include files in the source listing	Off
ln	Sets line width of the source listing to <i>n</i> spaces ($40 \leq n \leq 255$)	132
m	Lists macros and shows expansions in source listing	Off
o	Lists data-storage overflow	Off
pn	Sets page length of the source listing to <i>n</i> lines ($20 \leq n \leq 255$; 0 = no pagination)	55
s	Shows the symbol table in the source listing	Off
x	Shows the cross-reference table in the source listing	Off

Turn On a flag by listing it after option **-f**. Turn it Off by inserting it after option **-f** and putting an *n* before it.

Set the *l* and *p* flags by inserting an integer value after them in the flag list.

For example, the following use of option **-f** tells the assembler to list include files and include macro expansions in the source listing, not to show

the symbol table in the source listing, and to set line width of the source listing to 80 characters:

```
-f imnsl80
```

Option **-f** does not take effect unless option **-l** also appears on the command line.

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

-h — Display command-line option help screen

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

-l — Generate an assembly output listing

Tells the assembler to generate an assembly output listing. This listing is written to standard output unless you redirect it to a list file. The following example 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 symbol table

Causes private (compiler-generated) labels to be placed in the symbol table.

-le — Assemble using little-endian format

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

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

Overrides the default object-file name, which is the name of the assembly source file with a `.o` extension. The following example generates an object file `sort_1.obj`:

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

-pa — Perform Power (not PowerPC) assembly

Causes the assembler to perform Power (non-PowerPC) assembly only.

-Qy — Place information about the assembler in object file

Causes information about the assembler to be placed in the comment section of the assembled object file.

-v — Print summary of assembler statistics

Causes a summary of statistics about the program being assembled to be written to standard output.

-w — Suppress warning messages

Tells the assembler not to emit warning messages.

1.2 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 PowerPC microprocessor.

An assembly language statement contains up to four fields:

```
label:      opcode      operands      ; comment
```

label field A *label* is a location marker. See §1.2.4 for a discussion of labels.

opcode field An *opcode* is typically a two- to eight-letter assembly-language mnemonic for a PowerPC microprocessor instruction. For information about PowerPC assembly-language mnemonics, see the documentation for your particular PowerPC microprocessor.

The opcode field can also contain an assembler *directive* (also called a pseudo-operation) or a user-defined *macro* instead of an instruction mnemonic. See Chapter 3: *Assembler Directives* for a listing of directives available for the ELF assembler. See Chapter 2: *Writing Assembler Macros* for instructions on how to write your own macros.

The opcode field can begin in any column.

operands field An *operand* is an argument for the instruction in the opcode field. The operand field is separated from the opcode field by whitespace.

An operand can be an identifier or a constant, or an expression containing an identifier or constant.

comment field The comment field is optional. A *comment* contains information about the statement or group of statements to which it is attached. If it is present, a comment is separated from the operands field by whitespace and always begins with an exclamation point (!) or a pound sign (#). The comment continues for the rest of the line and ends with a line feed.

1.2.1 The ELF Assembler Character Set

The ELF assembler recognizes the following characters:

- alphabetic characters: A through Z, a through z
- numeric characters (decimal digits): 0 (zero) through 9
- special characters listed in the following table:

&	Ampersand	%	Percent sign
*	Asterisk	.	Period
@	At sign	+	Plus 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

Some special characters have a predefined function in PowerPC assembly language:

- A single period (.) represents the current location counter.
- An exclamation point (!) or pound sign (#) marks the beginning of a comment. The comment continues to the end of the line.
- A semicolon (;) separates statements on a line.

1.2.2 Identifiers

Identifiers are names of *variables*, *labels*, *functions*, and *registers*. The ELF assembler requires identifiers to conform to the following rules:

- Identifiers can contain the characters A through Z, a through z, 0 (zero) through 9, and the following special characters: \$, . (period), ?, and _ (underscore). Register names begin with a percent sign (%).
- Identifiers *cannot* start with numerals 0 (zero) through 9.

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

These are valid identifiers:

```
mon
.size
L14
_main
display__4DateFv
%r26
```

1.2.3 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.

1.2.3.1 Reserved Symbols

The following symbols are *reserved*. You cannot redefine them.

- a period (.) by itself; this represents the location counter
- register names and register-field names:
 - general-purpose registers %r0 through %r31
 - floating-point registers %f0 through %f31
 - condition-register fields %cr0 through %cr7

Note: Register names are not case-sensitive.

1.2.4 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 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. It terminates with a colon (:).

There are two types of labels, regular and numeric.

1.2.4.1 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. In any other module that refers to it, reference the label with an **.extern** directive.

Names of regular labels follow the general syntax for identifiers, as defined in §1.2.2: *Identifiers*.

These are examples of regular labels:

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

1.2.4.2 Numeric Labels

A *numeric label* is a single digit in the range 0 (zero) to 9, followed by a colon. For example:

```
1:      mov %ah, %al
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):

- `nb` refers to the nearest numeric label `n` defined before the reference.
- `nf` refers to the nearest numeric label `n` defined after the reference.

For example, this code:

```

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

```

has the same meaning as this code:

```

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

```

1.3 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 (`.`). When the source code is assembled, the assembler replaces the period with the address of the current byte.

WARNING: The assembler warns if it has to force the alignment of an instruction to a four-byte boundary. It does *not* warn when a jump is made to a misaligned address; it just jumps to the previous word boundary.

1.4 Constants

The assembler recognizes the following types of constants:

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

1.4.1 Integer Constants

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

Base	Prefix	Example
Binary	0B <i>or</i> 0b	0B101011, 0b101011
Octal	0 (<i>zero</i>)	053
Decimal	<i>None</i>	43
Hexadecimal	0X <i>or</i> 0x	0X2B, 0x2b

If an integer has no leading 0 (zero) or other prefix, the assembler understands it to be 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.

1.4.2 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. The following 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 optional 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`.

Caution: 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.

1.4.3 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 with the following assembler directives:

<code>.ascii</code>	<code>.pushsect</code>
<code>.asciz</code>	<code>.section</code>
<code>.file</code>	<code>.seg</code>
<code>.ident</code>	<code>.string</code>
<code>.machine</code>	<code>.version</code>

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 an ordinary character."
- You must express ASCII control characters with these character combinations:

Character Combination	Control	ASCII Value
<code>\b</code>	Backslash	0x08
<code>\f</code>	Form feed	0x0c
<code>\n</code>	Newline	0x0a
<code>\r</code>	Carriage return	0x0d
<code>\t</code>	Tab	0x09
<code>\0</code>	NULL	0x00

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 three hexadecimal characters preceded by a backslash and a lowercase x.

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

1.5 Operators and Expressions

An *expression* is a series of one or more *operands* (identifiers, constants, and subexpressions) separated by arithmetic 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 3: *Assembler Directives* for information about ELF assembler directives.

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.

The assembler uses 32-bit arithmetic.

1.5.1 Operators and Operator Precedence

Table 1.3 shows the arithmetic operators supported by the ELF assembler in their order of precedence, where a lower precedence number indicates a higher precedence.

Table 1.3 *Arithmetic Operators in Order of Precedence*

	Precedence	Operator	Operation
<i>Highest precedence</i>	1	()	Overrides other precedence
	2	~	Unary bitwise logical NOT
		+	Unary plus
		-	Unary minus

Table 1.3 Arithmetic Operators in Order of Precedence (Continued)

Precedence	Operator	Operation
3	/	Division
	%	Modulus
	*	Multiplication
	<<	Left shift
	>>	Right shift
4		Bitwise logical OR
	^	Bitwise logical XOR
	&	Bitwise logical AND
5	+	Addition
	-	Subtraction

Lowest precedence

1.5.2 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
```

1.6 Identifier Attributes

To indicate different relocation options, you can use *identifier attributes* to modify references to identifiers.

Identifier attributes begin with the “at” sign (@) and use the following syntax:

```
identifier@attribute
```

For example, the following identifier-attribute combination refers to the index of *ident*’s entry in the symbol table, rather than to its address in memory:

```
ident@symidx
```

Table 1.4 describes the identifier attributes used in the ELF assembler for PowerPC.

Table 1.4 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
@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

Table 1.4 Identifier Attributes (Continued)

Attribute	Description	Relocation Entry Generated
@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 1.5 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 1.5 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.

1.6.1 Dynamic Relocation

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

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

§2.1: *Defining a Macro*

§2.2: *Calling a Macro*

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.

2.1 Defining a Macro

A *macro definition* contains the actual code for the macro operation. It consists of three parts:

- heading
- macro body
- terminator

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. For 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. The name can be returned to its original use only by using the **.purgem** directive with the macro name.

Note: Macro names are case-sensitive.

See Chapter 3: *Assembler Directives* for a discussion of the **.macro** directive.

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 set off 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:

```
lab1\&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.

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 6
; arguments.
.macro INST mnemonic arg1 arg2 arg3 arg4 arg5 arg6
    .if $narg==1
        mnemonic
    .endif
    .if $narg==2
        mnemonic arg1
    .endif
    .if $narg==3
        mnemonic arg1 arg2
    .endif
    .if $narg==4
        mnemonic arg1 arg2 arg3
    .endif
    .if $narg==5
        mnemonic arg1 arg2 arg3 arg4
    .endif
```

```
.if $narg==6
    mnemonic arg1 arg2 arg3 arg4 arg5
.endif
.if $narg==7
    mnemonic arg1 arg2 arg3 arg4 arg5 arg6
.endif
.endm
```

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

Macro terminator The directive **.endm** terminates the macro definition. See Chapter 3: *Assembler Directives* for a discussion of this directive.

2.1.1 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. For example:

```
.purgem load_regs, store_data
```

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

2.1.2 Nesting 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, 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.

2.1.3 Suppressing Macro Expansion

You can suppress macro expansion at any point in the macro body by means of the **.exitm** directive. Any code occurring after this directive is not

included in the macro expansion. See Chapter 3: *Assembler Directives* for a discussion of the `.exitm` directive.

2.2 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
```

2.2.1 Arguments

Arguments (also called actual parameters) in a macro call 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 leave an argument off 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>
```

2.2.2 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, if a macro is declared as follows:

```
.macro min_max, parm1, parm2
```

and called 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 ampersand preceded by a backslash (`\&`); 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.

3 Assembler Directives

This chapter discusses assembler directives supported by the ELF assembler for PowerPC. It contains the following sections:

§3.1: *Quick-Reference Table of Assembler Directives*

§3.2: *Assembler Directives Listed by Operation*

§3.3: *Assembler Directive Reference*

The ELF assembler supports the directives listed alphabetically in Table 3.1. With assembler directives, also called *pseudo operations*, you can control program organization and manipulate data.

3.1 Quick-Reference Table of Assembler Directives

Table 3.1 provides a quick overview of assembler directives available for the ELF assembler. You can find more detailed descriptions of these directives in the reference section that starts on page 32.

Table 3.1 Assembler Directives (Pseudo Operations)

Directive	Description	Synonym	Valid for .bss Section?
.2byte	Generate initialized 16-bit value(s) (half-words) in current section.	.half, .short	No
.3byte	Generate initialized 24-bit value(s) in current section.		No
.4byte	Generate initialized 32-bit value(s) (full words) in current section.	.long, .word	No
.align	Advance current location counter to specified boundary.		
.ascii	Place string(s) without terminating null character in current section.		No

Table 3.1 Assembler Directives (Pseudo Operations) (Continued)

Directive	Description	Synonym	Valid for .bss Section?
.asciz	Place null-terminating string(s) in current section.	.string	No
.blank	Insert blank lines in source-code listing.		
.bss	Change current section to <code>.bss</code> .		
.byte	Generate initialized eight-bit value(s) in current section.		No
.comm	Define common block (uninitialized block of storage).	.common	
.common	Define common block (uninitialized block of storage).	.comm	
.data	Change current section to default <code>.data</code> section.		
.define	Define a macro variable.		
.double	Store double-precision floating-point constant(s) in current section.		No
.eject	Advance listing to top of page.		
.else	Indicate alternative code to be assembled if corresponding <code>.if*</code> condition is <code>false</code> .		
.elseif	Indicate code to be assembled if conditional expression is <code>true</code> and corresponding <code>.if*</code> condition is <code>false</code> .		
.endian	Change byte order of generated code.		
.endif	Terminate conditional block.		
.endm	Terminate macro definition.		
.endr	Terminate repeat block.		
.entry	Set the <code>ENTRY</code> ELF binding.		
.equ	Assign a value.	.set	
.even	Advance current location counter to an even two-byte boundary.	.align 1	
.exitm	Terminate macro expansion.		

Table 3.1 Assembler Directives (Pseudo Operations) (Continued)

Directive	Description	Synonym	Valid for .bss Section?
.extern	Designate a symbol as external.		
.file	Specify a source-file name.		
.float	Store single-precision floating-point constant(s) in current section.		No
.global	Export symbol(s).	.globl	
.globl	Export symbol(s).	.global	
.half	Generate initialized 16-bit value(s) (half-words) in current section.	.2byte , .short	No
.ident	Place string(s) into comment section of the object file.	.version	
.if	Indicate code to be assembled if conditional expression is true.		
.ifdef	Indicate code to be assembled if an identifier is defined.		
.ifeqs	Indicate code to be assembled if two strings are equal.		
.ifndef	Indicate code to be assembled if an identifier is not defined.	.ifnotdef	
.ifnes	Indicate code to be assembled if two strings are not equal.		
.ifnotdef	Indicate code to be assembled if an identifier is not defined.	.ifndef	
.include	Include specified source file.		
.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.		
.lcomm	Define local uninitialized block of storage.	.lcommon	
.lcommon	Define local uninitialized block of storage.	.lcomm	
.lflags	Set listing flags.		

Table 3.1 Assembler Directives (Pseudo Operations) (Continued)

Directive	Description	Synonym	Valid for .bss Section?
.line	Identify line number.		
.list	Enable source-code listing.		
.long	Generate initialized 32-bit value(s) in current section.	.4byte , .word	No
.machine	Specify valid PowerPC instruction set(s).		
.macro	Declare macro name and parameters.		
.nolist	Disable source-code listing.		
.popsect	Pop section stack; restore most recently pushed section.		
.previous	Resume prior section.		
.print	Print string to standard output.		
.purgem	Discard current macro definition.		
.pushsect	Push current section onto section stack; switch to new section.		
.rdata	Change current section to default read-only data section	.rodata	
.reloc	Specify relocation of the next instruction in the current section.		
.rep	Assemble a repeat block the specified number of times.		
.rodata	Change current section to default read-only data section	.rdata	
.sbttl	Specify subtitle for source-code listing.		
.sectflag	Set the SHF_* flags field of the specified section.		
.section	Define control section and type.	.seg	
.sectlink	Set the link field of one section to point to another section.		
.seg	Define control section and type.	.section	

Table 3.1 Assembler Directives (Pseudo Operations) (Continued)

Directive	Description	Synonym	Valid for .bss Section?
.set	Assign a value.	.equ	
.short	Generate initialized 16-bit value(s) (half-words) in current section.	.2byte, .half	No
.size	Specify a size in bytes.		
.skip	Skip bytes in current section.	.space	
.space	Skip bytes in current section.	.skip	
.string	Place null-terminated string(s) in current section.		No
.text	Change current section to default <code>.text</code> section.		
.title	Specify main title for source-code listing.		
.type	Specify a type.		
.undef	Undefine one or more macro variables.		
.version	Place string(s) in comment section.	.ident	
.weak	Specify weak ELF binding.		
.word	Generate initialized 32-bit value(s) (full words) in current section.	.4byte, .long	No

3.2 Assembler Directives Listed by Operation

Table 3.2 lists assembler directives according to the type of operation they perform.

Table 3.2 Assembler Directives Grouped by Function

Function	Directives		
Conditional assembly	.else	.if	.ifndef
	.elseif	.ifdef	.ifnes
	.endif	.ifeqs	.ifnotdef

Table 3.2 Assembler Directives Grouped by Function (Continued)

Function	Directives		
Data-storage declaration	<code>.2byte</code> <code>.3byte</code> <code>.4byte</code> <code>.align</code> <code>.ascii</code> <code>.asciz</code>	<code>.byte</code> <code>.double</code> <code>.endian</code> <code>.even</code> <code>.float</code> <code>.half</code>	<code>.long</code> <code>.short</code> <code>.skip</code> <code>.space</code> <code>.string</code> <code>.word</code>
File processing	<code>.ident</code>	<code>.version</code>	
High-level language (HLL) debugging	<code>.file</code> <code>.line</code>	<code>.size</code> <code>.type</code>	
Instruction-set specification	<code>.machine</code>		
Listing control	<code>.blank</code> <code>.eject</code> <code>.lflags</code>	<code>.list</code> <code>.nolist</code> <code>.print</code>	<code>.sbttl</code> <code>.title</code>
Macro definition	<code>.define</code> <code>.endm</code>	<code>.exitm</code> <code>.macro</code>	<code>.purgem</code> <code>.undef</code>
Repeat block	<code>.endr</code> <code>.irep</code>	<code>.irepc</code>	<code>.rep</code>
Section	<code>.bss</code> <code>.comm</code> <code>.common</code> <code>.data</code> <code>.lcomm</code> <code>.lcommon</code>	<code>.popsect</code> <code>.previous</code> <code>.pushsect</code> <code>.rdata</code> <code>.rodata</code>	<code>.sectflag</code> <code>.section</code> <code>.sectlink</code> <code>.seg</code> <code>.text</code>
Symbol declaration and binding	<code>.equ</code> <code>.entry</code> <code>.extern</code>	<code>.global</code> <code>.globl</code> <code>.reloc</code>	<code>.set</code> <code>.weak</code>

3.3 Assembler Directive Reference

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

Note: Names of assembler directives are not case-sensitive.

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

Generates initialized values (16-bit two's-complement) in the current section. Assembles *expression* arguments into consecutive half words.

.2byte allows misalignment; the assembler does not warn.

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

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

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

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

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

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

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

.4byte allows misalignment; the assembler does not warn.

Synonym for **.long** and **.word**.

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

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

Advances the current location counter to the boundary specified by *number*, where *number* is \log_2 of the alignment. For example:

.align 1 aligns to a two-byte boundary

.align 2 aligns to a four-byte boundary

.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.

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

.asciz *string* [, *string*, ...] — Place null-terminated string(s) in current section

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

Synonym for **.string**.

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

.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.

.bss — Change current section to .bss

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

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

Generates initialized bytes in the current section.

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

.comm *name*, *expression* [, *align*]**.common *name*, *expression* [, *align*] — Define 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**

.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

Macro assembler 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 identifier defaults to 1. The value of *name* is global to the whole program.

During macro processing, occurrences of the macro variable in the source code are replaced with its defined value.

To redefine *name*, use it in another **.define** directive. (The assembler emits a warning when 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; the assembler does not warn.

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

.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 it to start the listing of each subroutine on a new page.

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

Macro assembler directive

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

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

Macro assembler directive

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

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

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

.endif — Terminate conditional block

Macro assembler directive

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

.endm — Terminate macro definition

Macro assembler directive

Marks the end of a macro definition begun with the previous **.macro** directive. If another **.macro** directive occurs before the **.endm** directive, the macro initiated by that **.macro** directive is nested inside the first macro; it must be terminated by an **.endm** directive of its own. See §2.1.2: *Nesting Macros* for a discussion of nested macros.

.endr — Terminate repeat block

Macro assembler 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 — Assign a value

Synonym for **.set**.

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

Same as **.align 1**.

.exitm — Terminate macro expansion

Macro assembler 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 macro expansion only of 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.

.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; the assembler does not warn.

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

Generates initialized values (16-bit two's-complement) in the current section. Assembles *expression* arguments into consecutive half words.

.half allows misalignment; the assembler does not warn.

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

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

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

Places one or more *string*(s) in the comment section of the object file.

Synonym for **.version**.

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

Macro assembler 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.

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

Macro assembler directive

Indicates that all code between the **.ifdef** directive and the corresponding **.else** or **.endif** directive is to be assembled if *name* is defined. Otherwise 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.

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

Macro assembler directive

Indicates that all code between the **.ifeqs** directive and the corresponding **.else** or **.endif** directive is to be assembled if *string1* is equal to *string2*. Otherwise the next **.else**, **.elseif**, or **.endif** directive is processed. The two character strings must be enclosed in double quotes.

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

Macro assembler directive

Indicates that all code between the **.ifndef** directive and the corresponding **.else** or **.endif** directive is to be assembled if *name* is not defined. Otherwise 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.

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

Macro assembler directive

Indicates that all code between the **.ifnes** directive and the corresponding **.else** or **.endif** directive is to be assembled if *string1* is not equal to *string2*. Otherwise the next **.else**, **.elseif**, or **.endif** directive is

processed. The two character strings must be enclosed in double quotes. Use this directive inside macros to test macro parameters.

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

Macro assembler directive

Indicates that all code between the **.ifdef** directive and the corresponding **.else** or **.endif** directive is to be assembled if *name* is not defined. Otherwise 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.

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

Macro assembler 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 it cannot find the specified source file, it emits an error message and aborts.

The assembler passes the *pathname* and *filename* 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.

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

Macro assembler 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 *name* in the instruction sequence. Separate any occurrence of *name* from adjacent text with a **&** sequence. If *item* contains a comma or semicolon, enclose *item* in angle brackets: **<>**. For example:

```

                .irep  init_val, 10, 20, 30
1\&init_val:  .hword init_val
                .endr

```

The assembler converts this instruction sequence to the following:

```
110:      .hword 10
120:      .hword 20
130:      .hword 30
```

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

Macro assembler 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 *name* in the instruction sequence. Separate any occurrence of *name* from adjacent text with a `\&` sequence. For example:

```
l_\&ch:      .irepc ch, "XYZ"
              .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.

<i>name</i>	block name; references the storage, cannot be predefined
<i>expression</i>	block size; must be a positive integer
<i>align</i>	specifies the alignment, interpreted as in .align

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

Sets listing flags that control the listing of the source file. Table 3.3 shows these flags and their default settings.

Table 3.3 Assembler Listing Flags

Flag	Function	Default
<code>c</code>	Lists instructions not assembled because of conditional assembly statements	Off
<code>g</code>	Lists local label symbols in the symbol and cross-reference tables	Off
<code>i</code>	Lists include files in the source listing	Off
<code>ln</code>	Sets line width of the source listing to <i>n</i> spaces ($40 \leq n \leq 255$)	132
<code>m</code>	Lists macros and show expansions in source listing	Off
<code>o</code>	Lists data-storage overflow	Off
<code>pn</code>	Sets page length of the source listing to <i>n</i> lines ($20 \leq n \leq 255$; 0 (zero) = no pagination)	55
<code>s</code>	Shows the symbol table in the source listing	Off
<code>x</code>	Shows the cross-reference table in the source listing	Off

Turn On a flag by listing it after the **.lflag** directive. Turn it Off by inserting another **.lflag** directive and putting an *n* before the flag. Change the setting of the *l* and *p* flags by inserting an integer value after them in the flag list. The following example turns On the *c* and *g* flags, turns Off the *m* flag, and sets the value of the *p* flag to 66:

```
.lflag cgnmp66
```

None of the **.lflags** directives take effect unless you also specify option **-l** on the command line.

The **.lflags** directive is identical to assembler command-line option **-f**, except that you can apply the directive to portions of a module, whereas option **-f** affects the entire module.

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

Identifies a line number.

.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* , . . .] — **Generate initialized 32-bit value(s) (full words) in current section**

Generates initialized values (32-bit two's-complement) in the current section. Assembles *expression* arguments into consecutive full words; each *expression* is a non-floating-point constant.

.long allows misalignment; the assembler does not warn.

Synonym for **.4byte** and **.word**.

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

.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).
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**

Macro assembler 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.

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 leftmost of each. There can be fewer actual parameters than formal parameters. Formal parameters that lack a corresponding actual parameter default to the null string.

.nolist — Disable source-code listing

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

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

Pops the section stack, making the current section 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 a string to be printed to standard output. The string must be enclosed in double quotes. Use the **.print** directive to output an error message, with an **.if*** directive to test for a user-defined error condition and the **.err** directive to flag the error.

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

Macro assembler 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.

.reloc *symbol*, *reltype* — Specify relocation of the next instruction in the current section

Specifies the relocation type of the next instruction 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

Macro assembler 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.

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

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

.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	Set the SHF_BEGIN flag
end	Set the SHF_END flag

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

.section *name* [, *class*] — Define control section and type

Defines a control section *name* of type *class*, and arranges for subsequent code to be placed within the control section. *class* is one of the following attributes:

<code>text</code>	contains executable instructions
<code>data</code>	contains writable data
<code>rodata</code>	is a read-only section
<code>bss</code>	contains writable data initialized to 0 (zero)
<code>os</code>	contains special OS data

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

<code>a</code>	section occupies memory during execution
<code>w</code>	section is writable
<code>x</code>	section is executable

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

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

`.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.

`.seg name[, class]`— Define control section and type

Synonym for `.section`.

`.set name, expression`— Assign a value

Assigns the value of `expression` to `name`. This is the same as

`name = expression`

Synonym for `.equ`.

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

Generates initialized values (16-bit two's-complement) in the current section. Assembles `expression` arguments into consecutive half words.

`.short` allows misalignment; the assembler does not warn.

Synonym for `.2byte` and `.half`.

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

`.size name, expression` — Specify a size in bytes

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

`.skip number`

`.space number` — Skip bytes in current section

Skip *number* bytes in the current section.

- If the section is not a `text` section, zeros are placed in the section.
- If the section is a `text` section, `nop` instructions are placed in the section.

`.string string[, string, ...]` — Place null-terminated string(s) in current section

Places the characters in *string*(s) in the current section and terminates the string with a null (0) character.

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

`.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.

`.type name, type` — Specify a type

Associates *type* with *name*. Type information is passed to the linker. *type* can be one of the following:

<code>@function</code>	<code>"function"</code>
<code>@object</code>	<code>"object"</code>
<code>@no_type</code>	<code>"no_type"</code>

For example:

```
.type    some_function,@function
```

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

Macro assembler 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**.

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

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

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

Generates initialized values (32-bit two's-complement) in the current section. Assembles *expression* arguments into consecutive full words; each *expression* is a non-floating-point constant.

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

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

.word allows misalignment; the assembler does not warn.

Index

directive .2byte — generate initialized	16-bit values (half words) in current section	33
directive .half — generate initialized	16-bit values (half words) in current section	37
directive .short — generate initialized	16-bit values (half words) in current section	45
directive .3byte — generate initialized	24-bit values in current section	33
directive	.2byte — generate initialized 16-bit values (half words) in current section	33
directive .4byte — generate initialized	32-bit values (full words) in current section	33
directive .word — generate initialized	32-bit values (full words) in current section	47
directive .long — generate initialized	32-bit values (full words) in current section	42
directive	.3byte — generate initialized 24-bit values in current section	33
PowerPC	400 series privileged and unprivileged instructions .	42
directive	.4byte — generate initialized 32-bit values (full words) in current section	33

A

	actual parameters	24
directive	.align — advance current location counter to specified boundary	33
	arguments	24
	arithmetic operators	15
directive	.ascii — place strings without terminating null character in current section	33
directive	.asciz — place null-terminated strings in current section	34
asppc	assembler command	2
C-style preprocessing directives in	assembly code	2
option -l — generate an	assembly output listing	2, 6, 41
driver option -Hasmcpp — process C-style preprocessing directives in	assembly source file	2
	assembly source files	2
	assembly-language output listings	2
	assembly-language statements	7
identifier	attributes	16
	attributes of directive .section	44

B

option	-be — assemble using big-endian format	4
--------	--	---

option -be — assemble using	big-endian format	4
option	-big_si — suppress warning if a signed integer's value exceeds 32,767	4
directive .entry — set the ENTRY ELF	binding	36
directive .weak — specify weak ELF	binding	47
directive	.blank — insert blank lines in source-code listing	34
directive .blank — insert	blank lines in source-code listing	34
directive .endif — terminate conditional	block	36
directive .endr — terminate repeat	block	36
directive .comm — define common	block (uninitialized block of storage)	34
directive .common — define common	block (uninitialized block of storage)	34
directive .irepc — for each character in a string, assemble a repeat	block and replace identifier with character	40
directive .irep — for each item listed, assemble a repeat	block and replace identifier with item	39
directive .lcomm — define local uninitialized	block of storage	40
directive .lcommon — define local uninitialized	block of storage	40
directive .rep — assemble a repeat	block the specified number of times	44
directive .align — advance current location counter to specified	boundary	33
directive	.bss — change current section to .bss	34
directive	.byte — generate initialized eight-bit values in current section	34
directive .endian — change	byte order of generated code	36
directive .size — specialize a size in	bytes	46

C

option	-c — suppress display of the copyright message	4
	C-style preprocessing directives in assembly code	2
	calling a macro	24
	case sensitivity	3, 9, 21, 33, 43
directive .irepc — for each	character in a string, assemble a repeat block and replace identifier with character	40
the assembler	character set	8
directive	.comm — define common block (uninitialized block of storage)	34
whitespace in the	command line	3
option -h — display	command-line option help screen	6
	command-line options	3
assembler	command-line syntax	1
	comment delimiters	8

assembly code	comment field	7
directive .ident — place strings in	comment section of the object file	37
directive	.common — define common block (uninitialized	
directive .comm — define	block of storage)	34
directive .common — define	common block (uninitialized block of storage)	34
directive .else — indicate alternative code	common block (uninitialized block of storage)	34
to be assembled if corresponding .if*	condition is false	35
directive .elseif — indicate code to be	condition is false	35
assembled if a conditional expression is true	conditional block	36
and corresponding .if*	conditional expression is true	38
directive .endif — terminate	conditional expression is true and corresponding	
directive .if — indicate code to be assembled	.if* condition is false	35
if	conditional-assembly directives	31
directive .elseif — indicate code to be	constant value to it	5
assembled if a	constants	12
option -D — define an identifier and assign a	constants	12
floating-point	constants	13
integer	constants in current section	35
string	constants in current section	37
directive .double — store double-precision	control section and type	44
floating-point	control section and type	45
directive .float — store single-precision	copyright message	4
floating-point		
directive .section — define		
directive .seg — define		
option -c — suppress display of the		

D

option	-D — define an identifier and assign a constant	
	value to it	5
directive	.data — change current section to default .data	
	section	34
directive .rdata — change current section to	data section	44
default read-only	data section	44
directive .rodata — change current section to	data-storage declaration directives	32
default read-only	debugging directives	32
high-level language (HLL)	default .data section	34
directive .data — change current section to	default .text section	46
directive .text — change current section to	default object-file name	6
option -o — override		

directive .rdata — change current section to	default read-only data section	44
directive .rodata — change current section to	default read-only data section	44
directive .comment	.define — define a macro variable	34
	delimiters	8
	directive .2byte — generate initialized 16-bit values (half words) in current section	33
	directive .3byte — generate initialized 24-bit values in current section	33
	directive .4byte — generate initialized 32-bit values (full words) in current section	33
	directive .align — advance current location counter to specified boundary	33
	directive .ascii — place strings without terminating null character in current section	33
	directive .asciz — place null-terminated strings in current section	34
	directive .blank — insert blank lines in source-code listing	34
	directive .bss — change current section to .bss	34
	directive .byte — generate initialized eight-bit values in current section	34
	directive .comm — define common block (uninitialized block of storage)	34
	directive .common — define common block (uninitialized block of storage)	34
	directive .data — change current section to default .data section	34
	directive .define — define a macro variable	34
	directive .double — store double-precision floating-point constants in current section	35
	directive .eject — advance listing to top of page	35
	directive .else — indicate alternative code to be assembled if corresponding .if* condition is false	35
	directive .elseif — indicate code to be assembled if a conditional expression is true and corresponding .if* condition is false	35
	directive .endian — change byte order of generated code	36
	directive .endif — terminate conditional block	36
	directive .endm — terminate macro definition	36
	directive .endr — terminate repeat block	36
	directive .entry — set the ENTRY ELF binding	36
	directive .exitm — terminate macro expansion	36
	directive .extern — designate symbol as external	37

directive .file — specify a source-file name	37
directive .float — store single-precision floating-point constants in current section	37
directive .global — export symbols	37
directive .globl — export symbols	37
directive .half — generate initialized 16-bit values (half words) in current section	37
directive .ident — place strings in comment section of the object file	37
directive .if — indicate code to be assembled if conditional expression is true	38
directive .ifdef — indicate code to be assembled if an identifier is defined	38
directive .ifeqs — indicate code to be assembled if two strings are equal	38
directive .ifndef — indicate code to be assembled if an identifier is not defined	38
directive .ifnes — indicate code to be assembled if two strings are not equal	38
directive .ifndef — indicate code to be assembled if an identifier is not defined	39
directive .include — include specified source file ...	39
directive .irep — for each item listed, assemble a repeat block and replace identifier with item	39
directive .irepc — for each character in a string, assemble a repeat block and replace identifier with character	40
directive .lcomm — define local uninitialized block of storage	40
directive .lcommon — define local uninitialized block of storage	40
directive .lflags — set listing flags	41
directive .line — identify line number	41
directive .list — enable source-code listing	42
directive .long — generate initialized 32-bit values (full words) in current section	42
directive .machine — specify valid PowerPC instruction set(s)	42
directive .macro — declare macro name and parameters	42
directive .nolist — disable source-code listing	43
directive .popsect — pop section stack; restore most recently pushed section	43
directive .previous — resume prior section	43
directive .print — print string to standard output	43

	directive .purgem — discard current macro definition	43
	directive .pushsect — push current section onto section stack; switch to new section	43
	directive .rdata — change current section to default read-only data section	44
	directive .reloc — specify relocation of the next instruction in the current section	44
	directive .rep — assemble a repeat block the specified number of times	44
	directive .rodata — change current section to default read-only data section	44
	directive .sbttl — specify subtitle for source-code listing	44
	directive .sectflag — set the SHF_* flags field of the specified section	44
	directive .section — define control section and type	44
	directive .sectlink — set the link field of one section to point to another section	45
	directive .seg — define control section and type	45
	directive .set — assign a value	45
	directive .short — generate initialized 16-bit values (half words) in current section	45
	directive .size — specialize a size in bytes	46
	directive .skip — skip bytes in current section	46
	directive .space — skip bytes in current section	46
	directive .string — place null-terminated strings in current section	46
	directive .text — change current section to default .text section	46
	directive .title — specify main title for source-code listing	46
	directive .type — specify a type	46
	directive .undef — undefine one or more macro variables	47
	directive .weak — specify weak ELF binding	47
	directive .word — generate initialized 32-bit values (full words) in current section	47
assembler	directives	27
conditional-assembly	directives	31
data-storage declaration	directives	32
file-processing	directives	32
high-level language (HLL) debugging	directives	32
listing-control	directives	32

macro-definition	directives	32
repeat-block	directives	32
section	directives	32
symbol-declaration	directives	32
summary table of assembler	directives (pseudo operations)	27
using integers with	directives .float and .double	13
assembler	directives by category	31
C-style preprocessing	directives in assembly code	2
using integers with directives .float and	.double	13
directive	.double — store double-precision floating-point constants in current section	35
	driver option -Hasmcpp — process C-style preprocessing directives in assembly source file	2
	driver option -Hasopt — pass assembler options to the assembler	4

E

directive .byte — generate initialized	eight-bit values in current section	34
directive	.eject — advance listing to top of page	35
directive .entry — set the ENTRY	ELF binding	36
directive .weak — specify weak	ELF binding	47
directive	.else — indicate alternative code to be assembled if corresponding .if* condition is false	35
directive	.elseif — indicate code to be assembled if a conditional expression is true and corresponding .if* condition is false	35
directive	.endian — change byte order of generated code	36
directive	.endif — terminate conditional block	36
directive	.endm — terminate macro definition	36
directive	.endr — terminate repeat block	36
relocation	entries	17, 19
directive	.entry — set the ENTRY ELF binding	36
option	-Eo — send error messages to standard output	5
option -Eo — send	error messages to standard output	5
directive	.exitm — terminate macro expansion	36
directive .exitm — terminate macro	expansion	36
directive .global —	export symbols	37
directive .globl —	export symbols	37
directive	.extern — designate symbol as external	37

F

option	-f — set listing flags	5
--------	------------------------------	---

directive .sectlink — set the link	field of one section to point to another section	45
directive .sectflag — set the SHF_* flags	field of the specified section	44
option -Qy — place information about the assembler in object	file	6
readme	file — identifies last-minute changes and describes special files	iv
directive .file — specify a source-file name	file generation	37
object	file names	2
output	file-processing directives	32
assembly source	files	2
directive .lflags — set listing	flags	41
option -f — set listing	flags	5
directive .sectflag — set the SHF_*	flags field of the specified section	44
directive .float — store single-precision floating-point constants in current section		37
using integers with directives	.float and .double	13
directive .double — store double-precision	floating-point constants	12
directive .float — store single-precision	floating-point constants in current section	35
	floating-point constants in current section	37
	formal macro parameters	22
	forward referencing of symbols	9

G

directive .global — export symbols	37
directive .globl — export symbols	37

H

option -h — display command-line option help screen	6
directive .half — generate initialized 16-bit values (half words) in current section	37
directive .2byte — generate initialized 16-bit values (half words) in current section	33
directive .half — generate initialized 16-bit values (half words) in current section	37
directive .short — generate initialized 16-bit values (half words) in current section	45
driver option -Hasmcpp — process C-style preprocessing directives in assembly source file	2
driver option -Hasopt — pass assembler options to the assembler	4
macro heading	21
option -h — display command-line option help screen	6

high-level language	high-level language (HLL) debugging directives	32
	(HLL) debugging directives	32
I		
directive	.ident — place strings in comment section of the object file	37
option -D — define an	identifier and assign a constant value to it	5
	identifier attributes	16
directive .ifdef — indicate code to be assembled if an	identifier is defined	38
directive .ifndef — indicate code to be assembled if an	identifier is not defined	38
directive .ifnotdef — indicate code to be assembled if an	identifier is not defined	39
directive .irepc — for each character in a string, assemble a repeat block and replace	identifier with character	40
directive .irep — for each item listed, assemble a repeat block and replace	identifier with item	39
	identifiers	9
directive	.if — indicate code to be assembled if conditional expression is true	38
directive .else — indicate alternative code to be assembled if corresponding	.if* condition is false	35
directive .elseif — indicate code to be assembled if a conditional expression is true and corresponding	.if* condition is false	35
directive	.ifeqs — indicate code to be assembled if two strings are equal	38
directive	.ifndef — indicate code to be assembled if an identifier is not defined	38
directive	.ifnes — indicate code to be assembled if two strings are not equal	38
directive	.ifnotdef — indicate code to be assembled if an identifier is not defined	39
directive	.include — include specified source file	39
directive .machine — specify valid PowerPC	instruction set(s)	42
	instruction-set specification	32
PowerPC 400 series privileged and unprivileged	instructions	42
	integer constants	12
using	integers with directives .float and .double	13
directive	.irep — for each item listed, assemble a repeat block and replace identifier with item	39

directive	.irepc — for each character in a string, assemble a repeat block and replace identifier with character	40
L		
option	-l — generate an assembly output listing	2, 6, 41
option	-L — place private labels in the symbol table	6
assembly code	label field	7
regular and numeric	labels	10
	labels accessible to other modules	10
option -L — place private	labels in the symbol table	6
directive	.lcomm — define local uninitialized block of storage	40
directive	.lcommon — define local uninitialized block of storage	40
option	-le — assemble using little-endian format	6
directive	.lflags — set listing flags	41
directive	.line — identify line number	41
directive .sectlink — set the	link field of one section to point to another section	45
directive	.list — enable source-code listing	42
directive .blank — insert blank lines in source-code	listing	34
directive .list — enable source-code	listing	42
directive .nolist — disable source-code	listing	43
directive .sbttl — specify subtitle for source-code	listing	44
directive .title — specify main title for source-code	listing	46
generating a source-code	listing	2
option -l — generate an assembly output	listing	2, 6, 41
directive .lflags — set	listing flags	41
option -f — set	listing flags	5
directive .eject — advance	listing to top of page	35
	listing-control directives	32
assembly-language output	listings	2
option -le — assemble using	little-endian format	6
directive .lcomm — define	local uninitialized block of storage	40
directive .lcommon — define	local uninitialized block of storage	40
	location counter	8, 9, 11
directive .align — advance current	location counter to specified boundary	33
directive	.long — generate initialized 32-bit values (full words) in current section	42

M

directive	.machine — specify valid PowerPC instruction	
	set(s)	42
calling a	macro	24
directive	.macro — declare macro name and parameters	42
	macro body	22
recursive	macro calls	23
directive .endm — terminate	macro definition	36
directive .purgem — discard current	macro definition	43
	macro definition	21
directive .exitm — terminate	macro expansion	36
	macro heading	21
directive .macro — declare	macro name and parameters	42
	macro parameter substitution	25
formal	macro parameters	22
	macro terminator	23
directive .define — define a	macro variable	34
directive .undef — undefine one or more	macro variables	47
	macro-definition directives	32
	macros	21
redefining	macros	23

N

symbol	\$narg	22
directive	.nolist — disable source-code listing	43
	notational and typographic conventions	iii
directive .ascii — place strings without terminating	null character in current section	33
directive .asciz — place	null-terminated strings in current section	34
directive .string — place	null-terminated strings in current section	46
regular and	numeric labels	10

O

option	-o — override default object-file name	6
directive .ident — place strings in comment section of the	object file	37
option -Qy — place information about the assembler in	object file	6
	object file generation	2
option -o — override default	object-file name	6
assembly code	opcode field	7

	operands	15
assembly code	operands field	7
pseudo	operations	27
	operator precedence	15
arithmetic	operators	15
	option -be — assemble using big-endian format	4
	option -big_si — suppress warning if a signed integer's value exceeds 32,767	4
	option -c — suppress display of the copyright message	4
	option -D — define an identifier and assign a constant value to it	5
	option -Eo — send error messages to standard output	5
	option -f — set listing flags	5
	option -h — display command-line option help screen	6
driver	option -Hasmcpp — process C-style preprocessing directives in assembly source file	2
driver	option -Hasopt — pass assembler options to the assembler	4
	option -l — generate an assembly output listing	2, 6, 41
	option -L — place private labels in the symbol table	6
	option -le — assemble using little-endian format	6
	option -o — override default object-file name	6
	option -pa — perform Power (not PowerPC) assembly	6
	option -Qy — place information about the assembler in object file	6
	option -v — print summary of assembler statistics	7
	option -w — suppress warning messages	7
command-line	options	3
driver option -Hasopt — pass assembler	options to the assembler	4

P

option	-pa — perform Power (not PowerPC) assembly	6
directive .eject — advance listing to top of	page	35
macro	parameter substitution	25
actual	parameters	24
directive .macro — declare macro name and	parameters	42

formal macro	parameters	22
directive	.popsect — pop section stack; restore most recently pushed section	43
option -pa — perform	Power (not PowerPC) assembly	6
	PowerPC 400 series privileged and unprivileged instructions	42
directive .machine — specify valid	PowerPC instruction set(s)	42
option -pa — perform Power (not	PowerPC) assembly	6
operator	precedence	15
C-style	preprocessing directives in assembly code	2
driver option -Hasmcpp — process C-style	preprocessing directives in assembly source file	2
directive	.previous — resume prior section	43
directive	.print — print string to standard output	43
option -L — place	private labels in the symbol table	6
PowerPC 400 series	privileged and unprivileged instructions	42
driver option -Hasmcpp —	process C-style preprocessing directives in assembly source file	2
	program counter	11
assembler directives	(pseudo operations)	27
directive	.purge — discard current macro definition	43
directive .popsect — pop section stack; restore most recently	pushed section	43
directive	.pushsect — push current section onto section stack; switch to new section	43

Q

option	-Qy — place information about the assembler in object file	6
--------	---	---

R

directive	.rdata — change current section to default read-only data section	44
directive .rdata — change current section to default	read-only data section	44
directive .rodata — change current section to default	read-only data section	44
	readme file — identifies last-minute changes and describes special files	iv
	recursive macro calls	23
	redefining macros	23
	register and register-field names	9
	register expressions	16
register and	register-field names	9

directive	.reloc — specify relocation of the next instruction in the current section	44
	relocation entries	17, 19
directive	.rep — assemble a repeat block the specified number of times	44
directive .endr — terminate	repeat block	36
directive .irepc — for each character in a string, assemble a	repeat block and replace identifier with character	40
directive .irep — for each item listed, assemble a	repeat block and replace identifier with item	39
directive .rep — assemble a	repeat block the specified number of times	44
	repeat-block directives	32
	reserved symbols	9
directive	.rodata — change current section to default read-only data section	44

S

directive	.sbtbl — specify subtitle for source-code listing	44
directive	.sectflag — set the SHF_* flags field of the specified section	44
directive	.section — define control section and type	44
directive	.sectlink — set the link field of one section to point to another section	45
directive	.seg — define control section and type	45
directive	.set — assign a value	45
directive .sectflag — set the	SHF_* flags field of the specified section	44
directive	.short — generate initialized 16-bit values (half words) in current section	45
option -big_si — suppress warning if a	signed integer's value exceeds 32,767	4
directive .float — store	single-precision floating-point constants in current section	37
directive	.size — specialize a size in bytes	46
directive .size — specialize a	size in bytes	46
directive	.skip — skip bytes in current section	46
directive .include — include specified	source file	39
assembly	source files	2
directive .blank — insert blank lines in	source-code listing	34
directive .list — enable	source-code listing	42
directive .nolist — disable	source-code listing	43
directive .sbtbl — specify subtitle for	source-code listing	44
directive .title — specify main title for	source-code listing	46
generating a	source-code listing	2
directive .file — specify a	source-file name	37

directive .space — skip bytes in current section	46
directive .size — specialize a size in bytes	46
directive .popsect — pop section onto section	43
directive .pushsect — push current section onto section	43
directive .print — print string to standard output	43
option -Eo — send error messages to standard output	5
assembly-language statements	7
option -v — print summary of assembler statistics	7
directive .lcomm — define local uninitialized block of storage	40
directive .lcommon — define local uninitialized block of storage	40
directive .comm — define common block (uninitialized block of storage)	34
directive .common — define common block (uninitialized block of storage)	34
directive .string — place null-terminated strings in current section	46
string constants	13
string to standard output	43
string, assemble a repeat block and replace identifier with character	40
strings are equal	38
strings are not equal	38
strings in comment section of the object file	37
strings in current section	34
strings in current section	46
strings without terminating null character in current section	33
substitution	25
subtitle for source-code listing	44
symbol \$narg	22
symbol as external	37
symbol table	6
symbol-declaration directives	32
symbols	37
symbols	37
symbols	9
syntax	1
directive .ifeqs — indicate code to be assembled if two	
directive .ifnes — indicate code to be assembled if two	
directive .ident — place	
directive .asciz — place null-terminated	
directive .string — place null-terminated	
directive .ascii — place	
macro parameter	
directive .sbttl — specify	
directive .extern — designate	
option -L — place private labels in the	
directive .global — export	
directive .globl — export	
forward referencing of	
assembler command-line	

T

directive .ascii — place strings without macro	terminating null character in current section	33
directive	terminator	23
directive	.text — change current section to default .text section	46
directive	.title — specify main title for source-code listing	46
directive .title — specify main	title for source-code listing	46
directive	.type — specify a type	46
notational and	typographic conventions	iii

U

directive	.undef — undefine one or more macro variables	47
directive .lcomm — define local	uninitialized block of storage	40
directive .lcommon — define local	uninitialized block of storage	40
directive .comm — define common block	(uninitialized block of storage)	34
directive .common — define common block	(uninitialized block of storage)	34
PowerPC 400 series privileged and	unprivileged instructions	42

V

option	-v — print summary of assembler statistics	7
directive .define — define a macro	variable	34
directive .undef — undefine one or more macro	variables	47

W

option	-w — suppress warning messages	7
option -big_si — suppress	warning if a signed integer's value exceeds 32,767	4
option -w — suppress	warning messages	7
directive	.weak — specify weak ELF binding	47
directive	whitespace in the command line	3
directive	.word — generate initialized 32-bit values (full words) in current section	47
directive .4byte — generate initialized 32-bit values (full	words) in current section	33
directive .long — generate initialized 32-bit values (full	words) in current section	42
directive .word — generate initialized 32-bit values (full	words) in current section	47