

CodeWarrior™

Development Tools

Assembler Reference

Revised 2003/08/19

Metrowerks, the Metrowerks insignia, and CodeWarrior are registered trademarks of Metrowerks Corp. in the US and/or other countries. All other trade names, trademarks and registered trademarks are the property of their respective owners.

© Copyright. 2003. Metrowerks Corp. ALL RIGHTS RESERVED.

Metrowerks reserves the right to make changes without further notice to any products herein to improve reliability, function or design. Metrowerks does not assume any liability arising out of the application or use of any product described herein. Metrowerks software is not authorized for and has not been designed, tested, manufactured, or intended for use in developing applications where the failure, malfunction, or any inaccuracy of the application carries a risk of death, serious bodily injury, or damage to tangible property, including, but not limited to, use in factory control systems, medical devices or facilities, nuclear facilities, aircraft or automobile navigation or communication, emergency systems, or other applications with a similar degree of potential hazard.

Documentation stored on electronic media may be printed for personal use only. Except for the forgoing, no portion of this documentation may be reproduced or transmitted in any form or by any means, electronic or mechanical, without prior written permission from Metrowerks.

ALL SOFTWARE, DOCUMENTATION AND RELATED MATERIALS ARE SUBJECT TO THE METROWERKS END USER LICENSE AGREEMENT FOR SUCH PRODUCT.

How to Contact Metrowerks:

Corporate Headquarters	Metrowerks Corporation 7700 West Parmer Lane Austin, TX 78729 U.S.A.
World Wide Web	http://www.metrowerks.com
Ordering & Technical Support	Voice: (800) 377-5416 Fax: (512) 996-4910

Table of Contents

1 Introduction	7
Read the Release Notes!	7
What's in This Book.	7
Chapter Descriptions	8
Code Examples	8
Conventions Used in This Manual	9
Where to Learn More	10
 2 Assembly Language Syntax	 11
Assembly Language Statements Description	11
Assembly Language Statement Syntax.	12
Symbols	14
Labels	14
Non-local labels	15
Local labels	15
Relocatable labels	16
Equates	17
Equate names	18
Forward Equates.	18
Case-sensitive identifiers	19
Constants	19
Integer Constants	20
Floating-Point Constants	21
Character Constants	21
Expressions	22
Comments.	24
Data Alignment	25
 3 Using Directives	 27
Macro Directives	27
macro	28
endm	28
mexit.	28
#define	28
Conditional Preprocessor Directives.	29

Table of Contents

if	30
ifdef	30
ifndef	30
ifc	31
ifnc	31
endif	31
elseif	31
else	33
Other conditional preprocessor directives	33
.ifeq (if equal)	33
.ifne (if not equal)	33
.iflt (if less than)	33
.ifle (if less than or equal).	34
.ifgt (if greater than)	34
.ifge (if greater than or equal)	34
Section Control Directives	34
text	35
data	35
rodata	35
bss	35
sdata	35
sdata2	35
sbss	36
debug	36
previous	36
offset	36
section	37
Scope Control Directives.	39
global	40
extern	40
public	40
Symbol Definition Directives.	40
set	41
<i>equal sign (=)</i>	41
equ	41
textequ	41
Data Declaration Directives	42

Integer Directives	42
byte	42
short	42
long	42
space	43
fill	43
String Directives	43
ascii	44
asciz	44
Floating-Point Directives	44
float	44
double	44
Assembler Control Directives	44
align	45
endian	45
error	45
include	45
pragma	46
org	46
option	46
Debugging Directives	48
file	49
function	49
line	50
size	50
type	50
4 Using Macros	51
Defining Macros	51
Macro Definition Syntax	51
Defining a macro with the .macro directive	52
Defining a macro with the #define directive	53
Using Macro Arguments	54
Using Local Labels in a Macro	56
Creating Unique Labels and Equates	56
Referring to the Number of Arguments	57
Invoking Macros	57

5 Common Assembler Settings	59
Displaying the Assembler Target Settings Panel.	59
Common Assembler Settings Descriptions	60
Labels must end with ':'	60
Directives begin with ':'	61
Case-sensitive identifiers	61
Allow space in operand field.	62
Generate listing file.	62
Prefix file	62
6 PowerPC-Specific Information	63
Related Documentation	63
PowerPC-Specific Examples	63
Index	69

Introduction

This manual describes the assembly language syntax and the CodeWarrior IDE settings for the processor-specific assemblers provided by CodeWarrior.

This chapter includes the following topics:

- [Read the Release Notes!](#)
- [What's in This Book](#)
- [Conventions Used in This Manual](#)
- [Where to Learn More](#)

Read the Release Notes!

The release notes contain important information about new features, bug fixes, and incompatibilities and reside in the following directory:

```
{CodeWarrior directory}\Release_Notes
```

What's in This Book

CodeWarrior provides several assemblers, depending on the processor for which you are developing code. This manual describes the syntax for assembly language statements, including macros and directives, used by the CodeWarrior assemblers.

NOTE Refer to the *Targeting* manual for your target processor and *C Compilers Reference* for information on the inline assembler provided by the CodeWarrior C/C++ compiler.

The basic syntax of assembly language statements is identical among the processor-specific assemblers (which this manual describes). However, the instruction mnemonics and register names for each processor differ.

This manual assumes you are familiar with assembly language and the processor for which you are developing code.

Unless otherwise stated, all the information in this manual applies to all the assemblers.

NOTE When this manual states that information applies to *the assembler*, the information refers to all the assemblers unless otherwise stated.

Chapter Descriptions

[Table 1.1](#) describes each chapter.

Table 1.1 Chapter descriptions

Chapter Title	Description
Introduction	This chapter, which describes this manual.
Assembly Language Syntax	Describes the main syntax of assembly language statements.
Using Directives	Describes the assembler directives.
Using Macros	Describes how to define and invoke macros.
Common Assembler Settings	Describes the assembler settings that are common among the assemblers

Code Examples

The code examples in the general chapters of this manual ([Table 1.1](#)) are for x86 processors. Any processor-specific chapters included in this manual contain corresponding examples wherever the code differs for the processor discussed in that chapter. Each processor-

specific example also is cross-referenced to the corresponding example in the general chapters.

Conventions Used in This Manual

This manual includes syntax statements that describe how to use assembly language statements. [Table 1.2](#) describes how to interpret the syntax.

Table 1.2 Understanding Syntax Examples

Syntax	Description
<code>literal</code>	Include the item in your statement as shown.
<i>metasymbol</i>	Replace the symbol with an appropriate value. The text after the syntax example describes what the appropriate values are.
<code>a b c</code>	Use one of the items in the group: either a, b, or c. Do not type the character because it is not part of the statement being defined.
<code>[a]</code>	Include the item, which is optional, when needed. The text after the syntax example describes when to include it. Do not type the square brackets ([]) because they are not part of the statement being defined.
<code>a ::= b c</code>	Substitute one or more items on the right side of the ::= symbol for the item on the left side as indicated by the syntax on the right side. In the example, a is defined as either b or c.

Where to Learn More

Each assembler uses the standard assembly language mnemonics and register names defined by the manufacturer of the applicable processor. For information on related documentation, see the processor-specific chapters of this manual.

- PowerPC Assembler: *The PowerPC Architecture*, IBM Inc.

The *PowerPC Assembler* supports all instructions for the Motorola MPC8xx, Motorola MPC505, and the IBM40x processors. It also supports both big-endian and little-endian code. It generates only 32-bit code.

Assembly Language Syntax

This chapter describes the syntax of assembly language statements and includes the following topics:

- [Assembly Language Statements Description](#)
- [Assembly Language Statement Syntax](#)
- [Symbols](#)
- [Constants](#)
- [Expressions](#)
- [Comments](#)
- [Data Alignment](#)

Assembly Language Statements Description

Three types of assembly language statements exist:

- Instruction statement
- Macro statement
- Directive statement

The type of the assembly language statement differs depending on whether the operation performed by the statement is a machine instruction, a macro call, or an assembler directive.

Instruction, directive, and macro names are case insensitive. For example, `MOV`, `MOV`, and `mov` all name the same instruction.

When creating assembly language statements, you must be aware of the following information:

- The maximum length of a statement or an expanded macro is 1000 characters.
- A statement must reside on a single line. However, you can concatenate two or more lines by typing a backslash (\) character at the end of the line.
- Each line of the source file can contain only one statement unless the assembler is running in GNU mode, which allows multiple statements to reside on one line, separated by semicolons.

Refer to the processor-specific chapters of this manual for information on where to find machine instructions for a particular chip. For more information on assembler directives, refer to [“Using Directives”](#). For more information on macros, refer to [“Using Macros”](#).

Assembly Language Statement Syntax

[Listing 2.1](#) shows the syntax of an assembly language statement.

Listing 2.1 Statement syntax

statement ::= [symbol] operation [operand] [,operand] . . . [comment]

operation ::= machine_instruction | assembler_directive | macro_call

operand ::= symbol | constant | expression | register_name

[Table 2.1](#) provides information related to the syntax shown in [Listing 2.1](#).

Table 2.1 Syntax-related information

Syntax Element	Description
<i>symbol</i>	A <i>symbol</i> is a combination of characters that represents a value. For more information, see “Symbols” .
<i>machine_instruction</i>	A machine instruction for your target processor. For information on where to find machine instructions for a particular processor, see the processor-specific chapters of this manual.
<i>assembler_directive</i>	Assembler directives are special instructions that tell the assembler how to process other assembly language statements. For example, certain assembler directives tell the assembler where the beginning and end of a macro is. For more information on assembler directives, see “Using Directives” .
<i>macro_call</i>	A call to a previously specified macro. For information on macro-related assembler directives, see “Macro Directives” . For more information on macros, see “Using Macros” .
<i>constant</i>	A defined value such as a string of characters or a numeric value. For more information, see “Constants” .
<i>expression</i>	A mathematical expression. For more information, see “Expressions” .
<i>register_name</i>	The name of a register; these names are processor-specific. For information on related processor-specific documentation, see the processor-specific chapters of this manual.
<i>comment</i>	A <i>comment</i> is text that the assembler ignores. You can use comments to document your code. For more information, see “Comments” .

Symbols

A *symbol* is a group of characters that represents a value, such as an address, numeric constant, string constant, or character constant. The length of a symbol name is unlimited.

The syntax of a symbol follows:

symbol ::= *label* | *equate*

NOTE For the complete syntax of an assembly language statement, see [Listing 2.1](#).

In general, a symbol has file-wide scope. *File-wide scope* means that you can access the symbol anywhere within the file where you defined the symbol and only within that file. However, symbols sometimes have a different scope. For more information, see [“Local labels”](#).

This section discusses the following topics:

- [Labels](#)
- [Equates](#)
- [Case-sensitive identifiers](#)

Labels

A *label* is a symbol that represents an address. The assembler provides local labels and non-local labels. Whether a label is local or non-local determines its scope.

The syntax of a label follows:

label ::= *local_label* [:] | *non-local_label* [:]

By default, a label ends with a colon (:) and can begin in any column. However, if you are porting existing code that does not follow this convention, clear the **Labels must end with ':'** checkbox on the Assembler settings panel. After you clear the checkbox, a label must either begin in column 1 or end with a colon (:).

NOTE For more information, see [“Common Assembler Settings”](#).

This section contains the following topics:

- [Non-local labels](#)
- [Local labels](#)
- [Relocatable labels](#)

Non-local labels

A non-local label is a symbol that represents an address and has file-wide scope.

The first character of a non-local label must be one of the following:

- A letter (a-z or A-Z)
- A period (.)
- A question mark (?)
- An underscore (_)

The subsequent characters of a non-local label can be either a character from the preceding list or one of the following:

- A numeral between zero and nine (0-9)
- A dollar sign (\$)

Local labels

A local label is a symbol that represents an address and has local scope. *Local scope* means that the scope of the label extends forward and backward within the file until the point where the assembler encounters a non-local label.

The first character of a local label must be an at-sign (@). The subsequent characters of a local label must be one of the following:

- A letter (a-z or A-Z)
- A numeral between zero and nine (0-9)
- An underscore (_)
- A question mark (?)
- A dollar sign (\$)

- A period (.)

NOTE: You cannot export local labels. In addition, local labels do not appear in debugging tables.

Within an expanded macro, the scope of local labels works differently:

- The scope of local labels defined in macros does not extend outside the macro.
- A non-local label in an expanded macro does not end the scope of locals in the unexpanded source.

[Listing 2.2](#) shows the scope of local labels in macros.

Listing 2.2 The scope of local labels in a macro

```
MAKEPOS     .MACRO
    cmp      eax, 1
    jne      @SKIP
    neg      eax
@SKIP:      ;Scope of this label is within the macro
    .ENDM
START:
    mov      eax, COUNT
    cmp      eax, 1
    jne      @SKIP
MAKEPOS
@SKIP:      ;Scope of this label is START to END
            ;excluding lines arising from
            ;macro expansion
    add      eax, 1
END:     ret
```

In [Listing 2.2](#), the @SKIP label defined in the macro does not conflict with the @SKIP label defined in the main body of code.

Relocatable labels

The assembler assumes a flat 32-bit memory space. You can specify the relocation of a 32-bit label with the expressions shown in [Table 2.2](#).

NOTE Some expressions are not allowed in all assemblers.

Table 2.2 Relocatable label expressions

This...	Represents this
<i>label</i>	The offset from the address of the label to the base of its section, relocated by the section base address. It also is the PC-relative target of a branch or call. It is a 32-bit address.
<i>label@l</i>	The low 16-bits of the relocated address of the symbol.
<i>label@h</i>	The high 16-bits of the relocated address of the symbol. You can OR this with <i>label@l</i> to produce the full 32-bit relocated address.
<i>label@ha</i>	The adjusted high 16-bits of the relocated address of the symbol. You can add this to <i>label@l</i> to produce the full 32-bit relocated address.
<i>label@sdx</i>	For labels in a small data section, the offset from the base of the small data section to the label. This syntax is not allowed for labels in other sections.
<i>label@got</i>	For chips with a global offset table, the offset from the base of the global offset table to the 32-bit entry for label.

Equates

An *equate* is a symbol that represents any value. You can create an equate with a `.equ` or `.set` directive.

NOTE For more information, see [“equ”](#) and [“set”](#).

This section contains the following topics:

- [Equate names](#)
- [Forward Equates](#)

Equate names

The first character of an must be one of the following:

- A letter (a-z or A-Z)
- A period (.)
- A question mark (?)
- An underscore (_)

The subsequent characters of an equate can be either a character from the preceding list or one of the following:

- A numeral between zero and nine (0-9)
- A dollar sign (\$)

Forward Equates

The assembler allows *forward equates*. This means that you can refer to an equate in a file before it is defined. When an assembler encounters an expression it cannot resolve because the expression references a symbol whose value is not known, the assembler retains the expression and marks it as unresolved. After the assembler reads the entire file, it reevaluates unresolved expressions and, if necessary, repeatedly reevaluates them until it resolves them all or it cannot resolve them any further. If the assembler cannot resolve an expression, it raises an error.

However, the assembler must be able to immediately resolve any expression whose value affects the location counter.

NOTE Note that if the assembler can make a reasonable assumption about the location counter, the expression is allowed. For example, in a forward branch instruction for a 68K processor, you can specify a default assumption of 8, 16, or 32 bits.

Thus, the code in [Listing 2.3](#) is valid.

Listing 2.3 Valid forward equate

```
.data
.long   alloc_size
alloc_size .set   rec_size + 4
```

```
                ; a valid forward equate on next line
rec_size        .set    table_start-table_end
                .text;...
table_start:
    ; ...
table_end:
```

However, the code in [Listing 2.4](#) is not valid. The assembler cannot immediately resolve the expression in the `.space` directive. Consequently, the effect on the location counter is unknown.

Listing 2.4 Invalid forward equate

```
                ;invalid forward equate on next line
rec_size        .set    table_start-table_end
                .space rec_size
                .text; ...
table_start:
    ; ...
table_end:
```

Case-sensitive identifiers

The **Case-sensitive identifiers** checkbox on the Assembler settings panel lets you choose whether symbols are case-sensitive.

If you click the checkbox, symbols are case sensitive, so `SYM1`, `sym1`, and `Sym1` are three different symbols, for example.

If you clear the checkbox, symbols are *not* case-sensitive, so `SYM1`, `sym1`, and `Sym1` are the same symbol, for example. By default, this option is on.

Constants

The assembler recognizes three kinds of constants:

- [Integer Constants](#)
- [Floating-Point Constants](#)
- [Character Constants](#)

Integer Constants

[Table 2.3](#) lists the preferred notation for integer constants.

Table 2.3 Preferred integer constant notation

For numbers of this type...	Use...
Decimal	A string of decimal digits, such as 12345678.
Hexadecimal	A dollar sign (\$) followed by a string of hexadecimal digits, such as \$deadbeef.
Binary	A percent sign (%) followed by a string of binary digits, such as %01010001.

To help you port existing code, the assembler also supports the notation in [Table 2.4](#).

Table 2.4 Alternate integer constant notation

For numbers of this type...	Use...
Hexadecimal	0x followed by a string of hexadecimal digits, such as 0xdeadbeef.
Hexadecimal	0 followed by a string of hexadecimal digits, such as 0deadbeef, and ending with an h, such as 0deadbeefh.
Decimal	A string of decimal digits followed by d, such as 12345678d.
Binary	A string of binary digits followed by a b, such as 01010001b.

NOTE The assembler stores and manipulates integer constants using 32-bit signed arithmetic.

Floating-Point Constants

You can specify floating point constants in either hexadecimal or decimal format. A floating point constant in decimal format must contain either a decimal point or an exponent, e.g. 1E-10 or 1.0.

You can use floating point constants only in data generation directives like `.float` and `.double`, or in floating point instructions. You cannot use them in expressions.

Character Constants

Enclose a character constant in single quotes unless the character constant includes a single quote. In that case, enclose the character constant in double quotes.

NOTE A character constant cannot include both single and double quotes.

The maximum width of a character constant is 4 characters, depending on the context. For example, the following items are character constants:

- 'A'
- 'ABC'
- 'TEXT'

A character constant can contain any of the escape sequences shown in [Table 2.5](#).

Table 2.5 **Escape sequences**

Sequence	Description
\b	Backspace
\n	Line feed (ASCII character 10)
\r	Return (ASCII character 13)
\t	Tab
\"	Double quote

Sequence	Description
\\	Backslash
\nnn	Octal value of \nnn

A character constant is zero-extended to 32 bits during computation. You can use a character constant anywhere you can use an integer constant.

Expressions

The assembler evaluates expressions using 32-bit signed arithmetic and does not check for arithmetic overflow.

Since there is no common set of operators in the existing assemblers for different processors, the assembler uses an expression syntax similar to the one for the C language. Expressions use the C language arithmetic rules for such things as parentheses and associativity, and they use the same operators.

NOTE To refer to the program counter in an expression, use a period (.), dollar sign (\$), or asterisk (*).

The assembler supports the binary operators listed in [Table 2.6](#).

Table 2.6 Binary operators

Operator	Description
+	add
–	subtract
*	multiply
/	divide
%	modulo
	logical OR
&&	logical AND

Operator	Description
	bitwise OR
&	bitwise AND
^	bitwise XOR
<<	shift left
>>	shift right (zeros are shifted into high order bits)
==	equal to
!=	not equal to
<=	less than or equal to
>=	greater than or equal to
<	less than
>	greater than

The assembler supports the unary operators listed in [Table 2.7](#).

Table 2.7 **Unary operators**

Operator	Description
+	unary plus
-	unary minus
~	unary bitwise complement

The assembler also supports the operations listed in [Table 2.8](#).

Table 2.8 **Alternate operators**

Operator	Description
<>	not equal to
%	modulo

Operator	Description
	logical OR
	logical XOR

The operators have the following precedence, with the highest priority first:

1. unary + - ~
2. * / %
3. binary + -
4. << >>
5. < <= > >=
6. == !=
7. &
8. ^
9. |
10. &&
11. ||

Comments

Comments are text that the assembler ignores. You can use them to document your code.

There are several ways you can specify comments:

- Type a semicolon (;) followed by your text entry.
 - **In GNU Mode** - The semicolon indicates multiple assembly instructions on one line.
 - **Not in GNU Mode** - The semicolon is interpreted as a comment.
- Use the following types of C-style comments, which can start in any column:

```
/* This is a comment. */  
// This is a comment.
```


- Type an asterisk (*) as the first character of the line followed by your comment.

NOTE The asterisk (*) must be the first character of the line for it to specify a comment. The asterisk has other meanings when it occurs elsewhere in a line.

- Clear the **Allow space in operand field** checkbox on the Assembler settings panel. In this case, the assembler ignores any text between a space character in the operand field and the end of the line. Therefore, after you type a space in the operand field, you can type a comment on the remainder of the line.
- Begin a comment with a pound sign (#), which can start in any column:

```
# This is a comment.
```

NOTE The assembler distinguishes between a comment that begins with a pound sign (#) and a preprocessor directive that begins with a pound sign.

The three immediately preceding comment methods are helpful for porting existing code.

Data Alignment

By default, the assembler aligns all data on a natural boundary for the data size and for the target processor family. You can turn off alignment with the `alignment` argument to the `.option` directive, described in [“option.”](#)

The assembler does not align data automatically in the `.debug` section. For more information on the `.debug` section, see [“Debugging Directives.”](#)

Using Directives

This chapter describes the directives that are available for the assembler.

NOTE Some directives are not available for every assembler.

By default, most directives must begin with a period (.). However if you clear the **Directives begin with '.'** checkbox of the Assembler settings panel, you can omit the period.

NOTE You can specify several preprocessor directives using the C/C++ preprocessor format.

This chapter discusses the following topics:

- [Macro Directives](#)
- [Conditional Preprocessor Directives](#)
- [Section Control Directives](#)
- [Scope Control Directives](#)
- [Symbol Definition Directives](#)
- [Data Declaration Directives](#)
- [Assembler Control Directives](#)
- [Debugging Directives](#)

Macro Directives

The following directives let you create macros:

- [macro](#)
- [endm](#)

- [mexit](#)
- [#define](#)

For more information on macros, see [“Using Macros”](#).

macro

label .macro [*parameter*] [, *parameter*] ...

Begins the definition of a macro named *label*, with the specified parameters.

endm

.endm

Ends a macro definition.

mexit

.mexit

Causes the assembler to stop macro processing before the .endm statement is reached and resume execution with the statement following the macro call.

#define

#define *name* [(*parms*)] *assembly_statement* [;] [\]
assembly_statement [;] [\]
assembly_statement
parms ::= *parameter* [, *parameter*] ...

Defines a macro named *name* with the specified parameters. You can extend *assembly_statement* by typing a backslash (\) and continuing the statement on the next physical line. You also can specify multiple assembly statements in the macro by typing a semicolon (;) followed by a backslash (\) and typing a new assembly statement on the next physical line. The assembler must be in GNU mode for multiple statements to reside on one line of code (refer to [“Comments.”](#)).

NOTE For more information, see [“Defining a macro with the #define directive”](#).

Conditional Preprocessor Directives

Conditional directives create a conditional assembly block. If you wrap some code with `ifdef` and `endif` you can control whether that code is included in compilation. This is useful for making several different builds that are slightly different.

You must use conditional directives together to form a complete block. The assembler also contains several variations of `.if` to make it easier to make blocks that test strings for equality, test whether a symbol is defined, and so on.

NOTE You can specify several of the conditional preprocessor directives using the C/C++ preprocessor format:

```
#if
#ifdef
#ifndef
#else
#elif
#endif
```

These directives function identically whether preceded by a pound sign (#) or a period with two exceptions. You cannot use the pound sign form of the directive in a macro, and the period (.) form of the `#elif` directive is `.elseif`.

This section discusses the following topics:

- [if](#)
- [ifdef](#)
- [ifndef](#)
- [ifc](#)
- [ifnc](#)
- [endif](#)
- [elseif](#)
- [else](#)
- [Other conditional preprocessor directives](#)

if

`.if bool-expr`

Specifies the beginning of a conditional assembly block, where *bool-expr* is a Boolean expression. If *bool-expr* is true, the assembler processes the statements associated with the `.if` directive. If *bool-expr* is false, the assembler skips the statements associated with the `.if` directive.

Each `.if` directive must have a matching `.endif` directive.

NOTE

A Boolean expression is a special type of arithmetic expression. The assembler interprets a Boolean expression that evaluates to zero as false and a Boolean expression that evaluates to a nonzero result as true. For more information on expressions, see [“Expressions”](#).

ifdef

`#ifdef symbol`

Specifies the beginning of a conditional assembly block and tests whether *symbol* is already defined. If *symbol* was defined previously, the assembler processes the statements associated with the `.ifdef` directive. If *symbol* is not yet defined, the assembler skips the statements associated with the `.ifdef` directive.

Each `.ifdef` directive must have a matching `.endif` directive.

ifndef

`.ifndef symbol`

Specifies the beginning of a conditional assembly block and tests whether *symbol* is not yet defined. If *symbol* is *not* yet defined, the assembler processes the statements associated with the `.ifndef` directive. If *symbol* is already defined, the assembler skips the statements associated with the `.ifndef` directive.

Each `.ifndef` directive must have a matching `.endif` directive.

ifc

```
.ifc string1, string2
```

Specifies the beginning of a conditional assembly block and tests whether *string1* and *string2* are equal. The comparison is case-sensitive. If the strings are equal, the assembler processes the statements associated with the `.ifc` directive. If the strings are *not* equal, the assembler skips the statements associated with the `.ifc` directive.

Each `.ifc` directive must have a matching `.endif` directive.

ifnc

```
.ifnc string1, string2
```

Specifies the beginning of a conditional assembly block and tests whether *string1* and *string2* are *not* equal. The comparison is case-sensitive. If the strings are *not* equal, the assembler processes the statements associated with the `.ifnc` directive. If the strings are equal, the assembler skips the statements associated with the `.ifnc` directive.

Each `.ifnc` directive must have a matching `.endif` directive.

endif

```
.endif
```

Specifies the end of a conditional assembly block. Each type of `.if` directive must have a matching `.endif` directive.

elseif

```
.elseif bool-expr
```

You can use the `.elseif` directive to create a series of directives that together comprise a logical multilevel *if-then-else* statement, the syntax of which follows:

```
.if bool-expr statement-group  
[ .elseif bool-expr statement-group ]...  
[ .else statement-group ]  
.endif
```

In the preceding syntax statement, *bool-expr* is any Boolean expression and *statement-group* is any group of assembly language statements.

Expanding the syntax as follows helps to explain the flow of the statement:

```
.if bool-expr-1
statement-group-1
.elseif bool-expr-2
statement-group-2
.elseif bool-expr-3
statement-group-3
.elseif bool-expr-4
statement-group-4
.else
statement-group-5
.en
dif
```

In the preceding syntax statement, if *bool-expr-1* is true, the assembler executes *statement-group-1* (the first group of conditional assembly language statements) and goes to the `.endif` directive. If *bool-expr-1* is false, the assembler skips *statement-group-1* and tests *bool-expr-2* in the first `.elseif` directive.

If *bool-expr-2* is true, the assembler executes *statement-group-2* and goes to the `.endif` directive. If *bool-expr-2* is false, the assembler skips *statement-group-2* and tests *bool-expr-3* in the second `.elseif` directive.

The assembler continues evaluating the Boolean expressions in succeeding `.elseif` directives until it comes to a Boolean expression that evaluates to true. If none of the `.elseif` directives have a Boolean expression that evaluates to true, the assembler processes the statements associated with the `.else` directive, if there is one.

else

`.else`

Marks the beginning of a conditional assembly block to execute if the Boolean expressions for an `.if` directive and its associated `.elseif` directives are false.

NOTE Using an `.else` directive is optional.

Other conditional preprocessor directives

For compatibility with other assemblers, the assembler also supports the following directives:

`.ifeq` (if equal)

`.ifeq string1, string2`

Specifies the beginning of a conditional block, and tests whether *string1* and *string2* are equal to each other.

- If the strings are equal to each other, the assembler processes the statements associated with this directive.
- If the strings are not equal to each other, the assembler skips over all associated statements.

`.ifne` (if not equal)

Specifies the beginning of a conditional block, and tests whether *string1* is not equal to zero.

- If the string is not equal to zero, the assembler processes the statements associated with this directive.
- If the string is equal to zero, the assembler skips over all associated statements.

`.iflt` (if less than)

Specifies the beginning of a conditional block, and tests whether *string1* is less than zero.

- If the string is less than zero, the assembler processes the statements associated with this directive.

- If the string is not less than zero, the assembler skips over all associated statements.

.ifl (if less than or equal)

Specifies the beginning of a conditional block, and tests whether *string1* is less than or equal to *string2*.

- If *string1* is less than or equal to *string2*, the assembler processes the statements associated with this directive.
- If *string1* is not less than or equal to *string2*, the assembler skips over all associated statements.

.ifgt (if greater than)

Specifies the beginning of a conditional block, and tests whether *string1* is greater than *string2*.

- If *string1* is less greater than *string2*, the assembler processes the statements associated with this directive.
- If *string1* is not greater than *string2*, the assembler skips over all associated statements.

.ifge (if greater than or equal)

Specifies the beginning of a conditional block, and tests whether *string1* is greater than or equal to *string2*.

- If *string1* is greater than or equal to *string2*, the assembler processes the statements associated with this directive.
- If *string1* is not greater than or equal to *string2*, the assembler skips over all associated statements.

Section Control Directives

The following directives identify the different sections of an assembly file:

- [text](#)
- [data](#)
- [rodata](#)
- [bss](#)
- [sdata](#)

- [sdata2](#)
- [sbss](#)
- [debug](#)
- [previous](#)
- [offset](#)
- [section](#)

text

`.text`

Specifies an executable code section. This must be in front of the actual code in a file.

data

`.data`

Specifies an initialized read-write data section.

rodata

`.rodata`

Specifies an initialized read-only data section.

bss

`.bss`

Specifies an uninitialized read-write data section.

sdata

`.sdata`

Specifies a small data section as initialized and read-write.

sdata2

`.sdata2`

Specifies a small data section as initialized and read-only.

sbss

`.sbss`

Specifies a small data section as uninitialized and read-write.

debug

`.debug`

Specifies a debug section. If you enable the debugger, the assembler automatically generates some debug information for your project. However, you use special directives in the debug section that provide the debugger with more detailed information. For more information on the debug directives, see [“Debugging Directives.”](#)

previous

`.previous`

Reverts to the previous section. This switch toggles between the current section and the previous section.

offset

`.offset [expression]`

Defines a record. The optional parameter *expression* specifies the initial location counter. The record definition extends until the start of the next section.

Within a record, you can use only the following directives:

<code>.equ</code>	<code>.set</code>	<code>.textequ</code>
<code>.align</code>	<code>.org</code>	<code>.space</code>
<code>.byte</code>	<code>.short</code>	<code>.long</code>
<code>.space</code>	<code>.ascii</code>	<code>.asciz</code>
<code>.float</code>	<code>.double</code>	

The data declaration directives (like `.byte` and `.short`) update the location counter but do not allocate any storage.

[Listing 3.1](#) shows a sample record definition.

Listing 3.1 A record definition with the offset directive

```

        .offset
top:    .short 0
left:   .short 0
bottom: .short 0
right:  .short 0
rectSize .equ *
```

section

For the ELF (Executable and Linkable Format) object file format, the `.section` directive has the following syntax:

```
.section name [ ,alignment ] [ ,type ] [ ,flags
]
```

Defines a section in an object file. Use this directive to create arbitrary relocatable sections, including sections to be loaded at an absolute address.

[Table 3.1](#) describes the syntax elements for the ELF `.section` directive.

Table 3.1 Syntax descriptions for ELF `.section` directive

Syntax Element	Description
<i>name</i>	The name of the section.
<i>alignment</i>	Specifies the alignment boundary of the section.
<i>type</i>	Numeric value that indicates the ELF section type. The default value for <i>type</i> is the type of the code section. For more information, see Table 3.2 .
<i>flags</i>	Numeric value that indicates the ELF section flags. The default value for <i>flags</i> is the value of the flags of the code section. For more information, see Table 3.3 .

The following example specifies a section named `vector` with an alignment of 4 bytes:

```
.section vector,4
```

[Table 3.2](#) defines the ELF section types.

Table 3.2 ELF section types

Type	Name	Description
0	NULL	Indicates that the section header is inactive.
1	PROGBITS	Indicates that the section contains information defined by the program.
2	SYMTAB	Indicates that the section contains a symbol table.
3	STRTAB	Indicates that the section contains a string table.
4	RELA	Indicates that the section contains relocation entries with explicit addends.
5	HASH	Indicates that the section contains a symbol hash table.
6	DYNAMIC	Indicates that the section contains information used for dynamic linking.
7	NOTE	Indicates that the section contains information that marks the file, often for compatibility purposes between programs.
8	NOBITS	Indicates that the section occupies no space in the object file.
9	REL	Indicates that the section contains relocation entries without explicit addends.

Type	Name	Description
10	SHLIB	Indicates that the section has unspecified semantics and, therefore, does not conform to the Application Binary Interface (ABI) standard.
11	DYNSYM	Indicates that the section contains a minimal set of symbols used for dynamic linking.

[Table 3.3](#) defines the ELF section flags.

Table 3.3 ELF section flags

Flag	Name	Description
0x00000001	WRITE	Indicates that the section contains data that is writable during execution.
0x00000002	ALLOC	Indicates that the section occupies memory during execution.
0x00000004	EXECINSTR	Indicates that the section contains executable machine instructions.
0xF0000000	MASKPROC	Indicates that the bits specified in this mask are reserved for processor-specific purposes.

Scope Control Directives

The assembler provides the following directives that let you import and export labels:

- [global](#)
- [extern](#)
- [public](#)

For more information on labels, see [“Labels”](#).

NOTE You cannot import or export equates or local labels.

global

```
.global label [ ,label ]...
```

Instructs the assembler to export the specified labels, that is, make them available to other files.

Use the `.extern` or `.public` directive to reference the labels in another file.

extern

```
.extern label [ ,label ]...
```

Instructs the assembler to import the specified labels, that is, to find the label definitions in another file.

Use the `.global` or `.public` directive to export the labels from another file.

public

```
.public label [ ,label ]...
```

Declares that the specified labels are public. If the labels are already defined in the same file, the assembler exports them, that is, makes them available to other files. If the equates are *not* already defined, the assembler imports them, that is, finds the label definitions in another file.

Symbol Definition Directives

You can use the following directives to create equates:

- [set](#)
- [equal sign \(=\)](#)
- [equ](#)
- [textequ](#)

set

equate .set *expression*

Temporarily assigns the value *expression* to *equate*. You can change the value of *equate* after defining it.

equal sign (=)

equate = *expression*

Temporarily assigns the value *expression* to *equate*. You can change the value of *equate* after defining it.

NOTE This directive is equivalent to .set and is available only for compatibility with assemblers provided by other companies.

equ

equate .equ *expression*

Permanently assigns the value *expression* to *equate*. You cannot change the value of *equate* after defining it.

textequ

equate .textequ "*string*"

Substitutes *equate* with the text you specify in *string*. You can use this directive, which helps to port existing code, to give new names to machine instructions, directives, and operands.

Whenever you use *equate*, the assembler replaces it with *string* before performing any other processing on that source line. [Listing 3.2](#) shows examples of .textequ statements.

Listing 3.2 textequ examples

```
dc.b      .textequ    ".byte"  
endc      .textequ    ".endif"
```

Data Declaration Directives

The assembler provides the following types of directives that initialize data:

- [Integer Directives](#)
- [String Directives](#)
- [Floating-Point Directives](#)

Integer Directives

The following directives initialize blocks of integer data:

- [byte](#)
- [short](#)
- [long](#)
- [space](#)
- [fill](#)

byte

```
[ label ] .byte expression [ , expression ]...
```

Declares an initialized block of bytes with the name *label*. The assembler allocates one byte for each *expression*. Each *expression* must fit in a byte.

short

```
[ label ] .short expression [ , expression ]...
```

Declares an initialized block of 16-bit short integers with the name *label*. The assembler allocates 16 bits for each *expression*. Each *expression* must fit in 16 bits.

long

```
[ label ] .long expression [ , expression ]...
```

Declares an initialized block of 32-bit short integers with the name *label*. The assembler allocates 32 bits for each *expression*. Each *expression* must fit in 32 bits.

space

[*label*] .space *expression*

Declares a block of zero-initialized bytes with the name *label*. The assembler allocates a block *expression* bytes long and initializes each byte to zero.

fill

[*label*] .fill *expression*

Declares a block of zero-initialized bytes with the name *label*. The assembler allocates a block *expression* bytes long and initializes each byte to zero.

String Directives

The following directives initialize blocks of character data:

- [ascii](#)
- [asciz](#)

A string can contain any of the escape sequences shown in [Table 3.4](#).

Table 3.4 **Escape sequences**

Sequence	Description
\b	Backspace
\n	Line feed (ASCII character 10)
\r	Return (ASCII character 13)
\t	Tab
\"	Double quote
\\	Backslash
\nnn	Octal value of \nnn

ascii

```
[ label ] .ascii "string"
```

Declares a block of storage for the string *string* with the name *label*. The assembler allocates a byte for each character in *string*.

asciz

```
[ label ] .asciz "string"
```

Declares a zero-terminated block of storage for the string *string* with the name *label*. The assembler allocates a byte for each character in *string*. The assembler then allocates an extra byte at the end and initializes the byte to zero.

Floating-Point Directives

The following directives initialize blocks of floating-point data:

- [float](#)
- [double](#)

float

```
[ label ] .float  value [ , value ]...
```

Declares an initialized block of 32-bit floating-point numbers with the name *label*. The assembler allocates 32 bits for each value *value*. Each value must fit in the specified size.

double

```
[ label ] .double value [ , value ]...
```

Declares an initialized block of 64-bit floating-point numbers with the name *label*. The assembler allocates 64 bits for each value *value*. Each value must fit in the specified size.

Assembler Control Directives

These directives let you control how the assembler emits code:

- [align](#)
- [endian](#)

- [error](#)
- [include](#)
- [pragma](#)
- [org](#)
- [option](#)

align

`.align expression`

Aligns the location counter to the next multiple of the *expression*. The *expression* must be a power of 2, such as 2, 4, 8, 16, or 32.

endian

`.endian big | little`

Specifies the byte ordering for the target processor.

NOTE You can use this directive only for processors that allow you to change the byte ordering.

error

`.error "error"`

Prints *error* to the Errors & Warnings window in the CodeWarrior IDE.

include

`.include filename`

Causes the assembler to switch input to *filename*. The assembler takes input from the specified file. When the assembler reaches the end of the file, it begins taking input from the assembly statement line that follows the `.include` directive.

The file specified by *filename* can contain an `.include` directive for another file.

pragma

`.pragma pragma-type setting`

Tells the assembler to assemble the code using a particular pragma setting.

org

`.org expression`

Changes the location counter to the value of *expression*, the value of which is relative to the base of the current section. The addresses of the subsequent assembly statements begin at the new location counter value. The value of *expression* must be greater than the current value of the location counter.

The following code snippet is presented as an example.

```
.text
.org 0x1000
Foo:
...
blr
```

The label `Foo` reflects the value of `.text + 0x1000`. The runtime value of `Foo` depends upon where the section defined by `.text`, is placed by the linker. For example, if `Foo` is placed at `0x10000000`, its final value is `0x10000000`.

NOTE You must use the CodeWarrior IDE and Linker to place code at an absolute address.

option

`.option keyword setting`

Sets the assembler options as described in [Table 3.5](#). Specifying *reset* sets the option to its previous setting. Using *reset* a second time resets the option to the setting before the current setting.

Table 3.5 Option keywords

Keyword	Description
<code>alignment off on reset</code>	Controls whether data is aligned on natural boundary. This does not correspond to any option in the Assembler settings panel.
<code>branchsize 8 16 32</code>	Specifies the size of forward branch displacement. This keyword applies only to the x86 and 68K assemblers. This does not correspond any option in the Assembler settings panel.
<code>case off on reset</code>	Specifies whether identifiers are case sensitive. If this option is on, identifiers are case sensitive. If this option is off, identifiers are not case sensitive. This corresponds to the Case-sensitive identifiers checkbox of the Assembler settings panel, described in “Case-sensitive identifiers” .
<code>colon off on reset</code>	Specifies whether labels must end with a colon (:). If this option is on, you must specify each label with a colon at the end. If this option is off, you can omit the colon from the end of label names that start in the first column. (This option corresponds to the Labels must end with ':' checkbox of the Assembler settings panel, described in “Labels must end with ':'” .)
<code>no_at_macros off on</code>	If this option is on, the assembler does not allow macros that use \$AT. If this option is off, the assembler produces a warning if a macro uses \$AT.
<div> <div>NOTE</div> <div>This option keyword string applies only to the MIPS Assembler.</div> </div>	

Keyword	Description
period off on reset	Specifies whether the assembler requires a period (.) in directive names. If this option is on, each directive must start with a period. If this option is off, you can omit the period in front of a directive. This corresponds to the Directives begin with '.' checkbox of the Assembler settings panel, described in “Directives begin with '.'” .
reorder off on reset	Specifies whether the assembler inserts a NOP (no operation) instruction after jumps and branches. If this option is on, the assembler inserts a NOP instruction. If this option is off, the assembler does not insert a NOP instruction, and you can substitute an instruction of your choice after jumps and branches.
<hr/> NOTE This option keyword string applies only to the MIPS Assembler. <hr/>	
space off on reset	Specifies whether the assembler allows a space in an operand field. If this option is on, operand fields can contain spaces. If this option is off, a space in the operand field signals the start of a comment. (This option corresponds to the Allow space in operand field checkbox of the Assembler settings panel, described in “Allow space in operand field” .)

Debugging Directives

When you enable the debugger, the assembler automatically generates some debug information for your project. However, you can use the following directives in the debug section to provide the debugger with more detailed information:

- [file](#)
- [function](#)

- [line](#)
- [size](#)
- [type](#)

NOTE The preceding directives are allowed *only* in the `.debug` and `.text` sections of an assembly file.

For the debugging directives to work, you must enable debugging for the particular file that contains them (in the Project window).

file

```
.file "filename"
```

Specifies the name of the file containing the source code. This directive enables generated assembly code to be correlated with the source code.

You must supply the `.function` and `.line` statements as well as the `.file` directive if you plan on writing your own DWARF code. The following is an example of how to use the `.file` directive when writing your own DWARF code.

```
.file "MyFile.c"
.text
.function "MyFunction",start,end-start
start:
    .line 1
    lwz r3, 0(r3)
    .line 2
    blr
end:
```

NOTE The `.file` directive must precede the other debugging directives in the assembly language file.

function

```
.function "func", label, length
```

Specifies that the subroutine *func* begins at *label* and is *length* bytes long. This directive generates file debugging data.

line

`.line number`

Specifies the absolute line number in the current source file that generated the subsequent code or data. The first line in the file is numbered 1.

size

`.size symbol, expression`

Specifies that *symbol* is *expression* bytes long.

type

`.type symbol, type`

Specifies that *symbol* is of type *type*, where type can be either `@function` (a function) or `@object` (a variable).

Using Macros

This chapter describes how to define and use macros. You can use the same macro language regardless of your target processor.

This chapter includes the following topics:

- [Defining Macros](#)
- [Invoking Macros](#)

Defining Macros

This section, which describes how to define macros, includes the following topics:

- [Macro Definition Syntax](#)
- [Using Macro Arguments](#)
- [Using Local Labels in a Macro](#)
- [Creating Unique Labels and Equates](#)
- [Referring to the Number of Arguments](#)

Macro Definition Syntax

A *macro definition* is one or more assembly statements that define:

- the name of a macro
- the format of the macro call
- the assembly statements to process when you invoke the macro

You can use the following methods to define a macro:

- [Defining a macro with the `.macro` directive](#)
- [Defining a macro with the `#define` directive](#)

Defining a macro with the .macro directive

One way to define a macro is to use the .macro directive. [Listing 4.1](#) shows the syntax of a macro definition using the .macro directive.

Listing 4.1 Macro definition syntax using the .macro directive

```
name:      .macro  [ parameter ] [ ,parameter ] ...  
          macro_body  
          .endm
```

The .macro directive indicates the first line of a macro definition. Every macro definition must end with the .endm directive.

[Table 4.1](#) describes the syntax elements shown in [Listing 4.1](#).

Table 4.1 Macro syntax descriptions for .macro directive

Syntax Element	Description
<i>name</i>	A label used to invoke the macro.
<i>parameter</i>	Operands that are passed to the macro and used in the macro body.
<i>macro_body</i>	One or more assembly language statements that are substituted for a macro call when you invoke the macro.

You can specify a conditional assembly block within a macro. Based on the result of the tested condition, you can use the .mexit directive to stop macro execution before the assembler reaches the .endm directive.

[Listing 4.2](#) shows a macro that uses the .mexit directive.

Listing 4.2 Conditional macro using the .mexit directive

```
# define a macro  
addto      .macro dest,val  
           .if val==0  
             no-op
```

```
.mexit # execution goes to the statement
        # immediately after the .endm directive
.elseif val==1
    # use compact instruction
    inc dest
.mexit # execution goes to the statement
        # immediately after the .endm directive
.endif
# if val is not equal to either 0 or 1,
# add dest and val
    add dest,val
# end macro definition
.endm
```

[Listing 4.3](#) shows assembly language code that calls the `addto` macro shown in [Listing 4.2](#).

Listing 4.3 Assembly code that calls the `addto` macro

```
# specify an executable code section
.text
xor     eax,eax
# call the addto macro
    addTo eax,0
    addTo eax,1
    addTo eax,2
    addTo eax,3
```

[Listing 4.4](#) shows the expanded `addto` macro calls shown in [Listing 4.3](#).

Listing 4.4 Expanded `addto` macro calls

```
xor     eax,eax
nop
inc     eax
add     eax,2
add     eax,3
```

Defining a macro with the `#define` directive

Another way to define a macro is to use the `#define` directive. [Listing 4.5](#) shows the syntax of a macro definition using the `#define` directive.

Listing 4.5 Macro definition syntax using the #define directive

```
#define name [ (parms) ] assembly_statement [ ; ] [ \ ]  
assembly_statement [ ; ] [ \ ]  
assembly_statement
```

parms ::= parameter [,parameter]...

NOTE If you specify parameters for a macro, you must enclose the parameters in parentheses.

[Table 4.2](#) describes the syntax elements shown in [Listing 4.5](#).

Table 4.2 Macro syntax descriptions for #define directive

Syntax Element	Description
<i>name</i>	A label used to invoke the macro.
<i>parameter</i>	Operands that are passed to the macro.
<i>assembly_statement</i>	<p>An assembly language statement that is substituted for a macro call when you invoke the macro. You can extend the assembly language statement beyond the length of one physical line by typing a backslash (\) at the end of a line and continuing the statement on the subsequent line.</p> <p>You also can specify multiple assembly statements in the macro by typing a semicolon (;) followed by a backslash (\) and typing a new assembly statement on the next physical line.</p>

Using Macro Arguments

You can refer to parameters directly by name. [Listing 4.6](#) shows the `setup` macro, which moves an integer into a register and branches to the label `_final_setup`.

Listing 4.6 The setup macro

```
setup:      .macro name
            mov     eax, name
            call    _final_setup
            .endm
```

[Listing 4.7](#) shows one way to invoke the setup macro.

Listing 4.7 Calling setup

```
#define VECT 0
setup VECT
```

[Listing 4.8](#) shows how the assembler expands the setup macro after the preceding call.

Listing 4.8 Expanded setup

```
move     eax, VECT
call     _final_setup
```

When you refer to named macro parameters in the macro body, you can precede or follow the macro parameter with &&. This lets you embed the parameter in a string. For example, [Listing 4.9](#) shows the smallnum macro, which creates a small float by appending the string E-20 to the macro argument.

Listing 4.9 The smallnum macro

```
smallnum:  .macro    mantissa
            .float    mantissa&&E-20
            .endm
```

[Listing 4.10](#) shows one way to invoke the smallnum macro.

Listing 4.10 Invoking smallnum

```
smallnum 10
```

[Listing 4.11](#) shows how the assembler expands the smallnum macro after the preceding call.

Listing 4.11 Expanding smallnum

```
.float    10E-20
```

Using Local Labels in a Macro

When you use a local label (a label that begins with @) in a macro, the scope of the label is limited to the expansion of the macro. For more information, see [“Local labels”](#).

Creating Unique Labels and Equates

You can generate unique labels and equates within a macro with the following characters: \@. Each time you invoke the macro, the assembler generates a unique symbol of the form ??nnnn, such as ??0001 or ??0002.

You refer to unique labels and equates (those that use \@) in your code with the same methods used for regular labels and equates. The assembler replaces the \@ sequence with a unique numeric string and increments the value of the string each time you invoke the macro.

[Listing 4.12](#) shows a macro that uses unique labels and equates.

Listing 4.12 Unique label macro

```
my_macro: .macro
          foo\@ = my_count
my_count .set my_count + 1
          add ebx, foo\@
          jmp label\@
          add eax, ebx
label\@:
          nop
          .endm
```

[Listing 4.13](#) shows a call to the my_macro macro twice (with my_count initialized to 0).

Listing 4.13 Invoking my_macro

```
my_count .set 0
my_macro
my_macro
```

[Listing 4.14](#) shows the expanded my_macro code after the calls in [Listing 4.13](#).

Listing 4.14 Expanded my_macro calls

```
foo??0000    =      my_count
my_count     .set   my_count + 1
              add    ebx, foo??0000
              jmp     label??0000
              add    eax, ebx

label??0000
              nop

foo??0001    =      my_count
my_count     .set   my_count + 1
              add    ebx, foo??0001
              jmp     label??0001
              add    eax, ebx

label??0001
              nop
```

Referring to the Number of Arguments

To refer to the number of non-null arguments passed to a macro, use the special symbol `narg`. You can use it only during macro expansion.

Invoking Macros

To invoke a macro, use its name in your assembler listing.

When invoking a macro, you must separate parameters with commas. To pass a parameter that includes a comma, enclose the parameter in angle brackets.

For example, [Listing 4.15](#) shows a macro named `pattern`, which repeats a pattern of bytes passed to it the number of times specified in the macro call.

Listing 4.15 The pattern macro

```
pattern:     .macro times,bytes
              .rept times
              .byte bytes
              .endr
            .endm
```

[Listing 4.16](#) shows a statement that calls `pattern`, passing a parameter that includes a comma.

Listing 4.16 Calling a macro with an argument that contains commas

```
.data
```

```
halfgrey:    pattern 4,<0xAA,0x55>
```

The call in [Listing 4.16](#) generates the same data as the code shown in [Listing 4.17](#).

Listing 4.17 Alternate way to generate a repeating pattern of bytes

```
halfgrey:    .byte 0xAA,0x55,0xAA,0x55,0xAA,0x55,0xAA,0x55
```

Common Assembler Settings

This chapter describes the settings on the Assembler target settings panel that are common to all the assemblers.

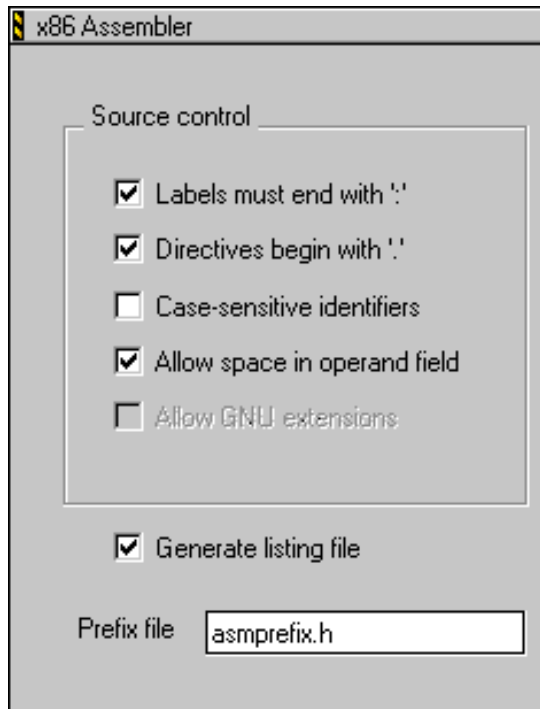
Displaying the Assembler Target Settings Panel

To modify the settings for an assembler:

1. Select **Edit > Project Settings**.
2. In the resulting dialog box, select the name of the assembler to see its settings panel.

[Figure 5.1](#) shows the settings on the Assembler target settings panel that are common to all the assemblers. For information on settings that may be specific to your assembler, see the processor-specific chapters of this manual.

Figure 5.1 Common assembler settings



Common Assembler Settings Descriptions

The following common assembler settings exist:

- [Labels must end with ':'](#)
- [Directives begin with ':'](#)
- [Case-sensitive identifiers](#)
- [Allow space in operand field](#)
- [Generate listing file](#)
- [Prefix file](#)

Labels must end with ':'

You can use the **Labels must end with ':'** checkbox to specify whether labels must end with a colon (:). If you select this checkbox, a label must end with a colon (:) and can begin in any column. If you

clear this checkbox, a symbol is a label if it starts in column 1 *or* if it ends with a colon (:).

By default, the **Labels must end with '.'** checkbox is selected. This checkbox corresponds to the `colon` parameter of the `.option` directive, described in [“option”](#).

NOTE The **Labels must end with '.'** checkbox is especially useful when porting existing code that has symbols that do not end with a colon (:).

For more information, see [“Labels”](#).

Directives begin with '.'

You can use the **Directives begin with '.'** checkbox to specify whether a period (.) must precede each directive name. If you select this checkbox, a period (.) must precede each directive. If you clear this checkbox, you can omit the period. For more information, see [“Using Directives”](#).

By default, the **Directives begin with '.'** checkbox is selected. This checkbox corresponds to the `period` parameter of the `.option` directive, described in [“option”](#).

Case-sensitive identifiers

You can use the **Case-sensitive identifiers** checkbox to specify whether symbols are case-sensitive. If you select this checkbox, symbols are case sensitive. For example, in this case, `SYM1`, `sym1`, and `Sym1` are three different symbols.

If you clear this checkbox, symbols are *not* case-sensitive. Therefore, in this case, `SYM1`, `sym1`, and `Sym1` are the same symbol. For more information, see [“Symbols.”](#)

NOTE Instruction, directive, and macro names are always case insensitive.

By default, the **Case-sensitive identifiers** checkbox is selected. This checkbox corresponds to the `case` parameter of the `.option` directive, described in [“option”](#).

Allow space in operand field

You can use the **Allow space in operand field** checkbox to specify whether a comment can start with a space in the operand field. If you select this checkbox, the assembler allows spaces in the operand field. If you clear this checkbox, the assembler ignores any text between a space character in the operand field and the end of the line (which makes that text a comment). For more information, see [“Comments”](#). By default, the **Allow space in operand field** checkbox is selected. This checkbox corresponds to the `space` parameter of the `.option` directive, described in [“option”](#).

Generate listing file

You can use the **Generate listing file** checkbox to create a text file that you can use to compare your source code with the machine code that the assembler produced. If you select this checkbox, the assembler creates a listing file using the source name and the following suffix:

```
.list
```

For example, for the file `test.asm`, the assembler assigns the following name to the listing file:

```
test.asm.list
```

If you clear the **Generate listing file** checkbox, the assembler does not create a listing file. By default, the **Generate listing file** checkbox is cleared.

Prefix file

You can use the **Prefix file** field to specify a file that the assembler processes before every assembly file in your project. The effect of using a prefix file is similar to putting the same `.include` directive at the beginning of every assembly file. By default, no prefix file is specified.

PowerPC-Specific Information

The CodeWarrior PowerPC assembler supports all instructions for the PowerPC processor.

This chapter provides information specific to the PowerPC processor. For example, this chapter discusses features and examples that differ from the information provided in the other chapters of this manual.

This chapter includes the following topics:

- [Related Documentation](#)
- [PowerPC-Specific Examples](#)

Related Documentation

PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors (published by Motorola, Inc.) is helpful for writing PowerPC assembly language code.

PowerPC-Specific Examples

This section contains examples shown in previous chapters that differ for the PowerPC assembler. [Table 6.1](#) lists the PowerPC-specific examples and the corresponding examples shown in previous chapters.

Table 6.1 Corresponding example table

Original Example	PowerPC-Specific Example
Listing 2.2	Listing 6.1
Listing 4.2	Listing 6.2
Listing 4.3	Listing 6.3
Listing 4.4	Listing 6.4
Listing 4.6	Listing 6.5
Listing 4.8	Listing 6.7
Listing 4.12	Listing 6.8
Listing 4.14	Listing 6.10

[Listing 6.1](#) shows the scope of local labels in macros.

Listing 6.1 PowerPC example: the scope of local labels in a macro

```
MAKEPOS:  .MACRO
    cmpdi   r3, 1
    bne     @SKIP
    neg     r3,r3
@SKIP:    ;Scope of this label is within the macro
    .ENDM
START:
    lis     r2, COUNT@h    ; COUNT is defined elsewhere
    ori     r2,r2,COUNT@l
    cmpdi   r3, 1
    bne     @SKIP
    MAKEPOS
@SKIP:    ;Scope of this label is START to END
           ;excluding lines arising from
           ;macro expansion
    add     r2,r2,1
END:
```

[Listing 6.2](#) shows a macro that uses the `.mexit` directive.

Listing 6.2 PowerPC example: conditional macro using the `.mexit` directive

```
; define a macro
addto:   .macro    val,dest
        .if val==0
```



```
        nop
        .mexit    ; execution goes to the statement
                  ; immediately after the .endm directive
        .elseif val==1
            addi dest,dest,1
        .mexit    ; execution goes to the statement
                  ; immediately after the .endm directive
        .endif
; if val is not equal to either 0 or 1,
; add dest and val
        addi dest,dest,val
; end macro definition
        .endm
```

[Listing 6.3](#) shows assembly language code that calls the `addto` macro shown in [Listing 6.2](#).

Listing 6.3 PowerPC example: assembly code that calls the `addto` macro

```
; specify an executable code section
        .text
        xor      r3,r3,r2
; call the addto macro
        addTo r3,0
        addTo r3,1
        addTo r3,2
        addTo r3,3
```

[Listing 6.4](#) shows the listing file for the macro calls shown in [Listing 6.3](#).

Listing 6.4 PowerPC example: Listing file contents for `addto` macro calls

```
.text
    nop
    addi    r3,r3,1
    addi    r3,r3,2
    addi    r3,r3,3
```

[Listing 6.5](#) shows the `setup` macro, which moves an integer into a register and branches to the label `_final_setup`.

Listing 6.5 PowerPC example: the setup macro

```
setup:      .macro name
            lis    r3, name@h
            ori    r3, name@l
            b      _final_setup
            .endm
```

[Listing 6.6](#) shows one way to invoke the setup macro. ([Listing 6.6](#) shows the same call as [Listing 4.7](#).)

Listing 6.6 PowerPC example: calling setup

```
#define VECT 0
setup VECT
```

[Listing 6.7](#) shows how the assembler expands the setup macro after a particular call.

Listing 6.7 PowerPC example: expanded setup

```
lis    r3,VECT
b      _final_setup
```

[Listing 6.8](#) shows a macro that uses unique labels.

Listing 6.8 PowerPC example: unique label macro

```
my_macro: .macro
          foo\@ = my_count
my_count .set my_count + 1
          addi r3,r3,foo\@
          b    label\@
          add  r4,r4,r3
label\:
          nop
          .endm
```

[Listing 6.9](#) shows a call to the my_macro macro twice (with my_count initialized to 0). ([Listing 6.9](#) shows the same calls as [Listing 4.13](#).)

Listing 6.9 PowerPC example: invoking my_macro

```
mycount .set 0
my_macro
my_macro
```

[Listing 6.10](#) shows the assembler output for the unique label macro.

Listing 6.10 PowerPC example: expanded my_macro calls

```
foo??0000    =      my_count
my_count     .set    my_count + 1
              addi    r3,r3,foo??0000
              b       label??f0000
              add     r4,r4,r3
label??0000
              nop
foo??0001    =      my_count
my_count     .set    my_count + 1
              addi    r3,r3,foo??0001
              b       label??0001
              add     r4,r4,r3
label??0001
              nop
```

PowerPC-Specific Information

PowerPC-Specific Examples

Index

Symbols

= (equal sign) symbol definition directive 41
@ (at-sign) 15
\@ (unique label symbol) 56

A

align assembler control directive 45
alignment keyword 47
Allow space in operand field 25
alternate operators 23
ascii data declaration directive 44
asciz data declaration directive 44
assembler control directives 44–47
 align 45
 endian 45
 error 45
 include 45
 option 46
 org 46
 pragma 46
at-sign (@) 15

B

binary operators 22
branchsize keyword 47
byte data declaration directive 42

C

case keyword 47
Case-sensitive identifiers checkbox 19, 47, 61
character constants 21
colon keyword 47
comments statement syntax 24
conditional directives 29–34
 else 33
 elseif 31
 endif 31
 if 30
 ifc 31
 ifdef 30
 ifeq 33
 ifge 34

ifgt 34
ifle 34
iflt 33
ifnc 31
ifndef 30
ifne 33

constants

character 21
floating-point 21
integer 20

D

data declaration directives 42–44
 ascii 44
 asciz 44
 byte 42
 double 44
 fill 43
 float 44
 long 42
 short 42
 space 43
data section control directive 35
debug section 25
debug section control directive 36
debugging directives 48–50
 file 49
 function 49
 line 50
 size 50
 type 50
#define macro directive 28
defining macros 51–57
Directives begin with '.' checkbox 27, 48, 61
double data declaration directive 44

E

ELF

 section flags 39
 section types 38
 syntax of section directive 37
else conditional directive 33
elseif conditional directive 31
endian assembler control directive 45

`endif` conditional directive 31
`endm` macro directive 28
`equ` symbol definition directive 41
`equates`
 creating unique 56-??
 definition 17
`error` assembler control directive 45
`extern` scope control directive 40

F

`file` debugging directive 49
`fill` data declaration directive 43
`float` data declaration directive 44
floating-point constants 21
forward `equates`, definition 18
function debugging directive 49

G

Generate listing file checkbox 62
`global` scope control directive 40

I

`if` conditional directive 30
`ifc` conditional directive 31
`ifdef` conditional directive 30
`ifeq` conditional directive 33
`ifge` conditional directive 34
`ifgt` conditional directive 34
`ifle` conditional directive 34
`iflt` conditional directive 33
`ifnc` conditional directive 31
`ifndef` conditional directive 30
`ifne` conditional directive 33
`include` assembler control directive 45
integer constants 20

L

labels
 creating unique 56-??
 definition 14
 Labels must end with ':' checkbox 14, 47, 60
`line` debugging directive 50
literal 9
local label 15
`long` data declaration directive 42

M

`macro` directive 28
macro directives 27-??
 `#define` directive 28
 `endm` directive 28
 `macro` directive 28
 `mexit` directive 28
macros
 arguments 54
 defining with the `#define` directive 53
 defining with the `.macro` directive 52
 invoking 57-58
 local labels in 56
 macro definition syntax 51
 number of arguments (`narg`) 57
 unique `equates` in 56
 unique labels in 56
`metasymbol` 9
`mexit` macro directive 28

N

`no_at_macros` keyword 47

O

`offset` section control directive 36
option 25
option assembler control directive 46
option keywords
 alignment 47
 branchsize 47
 case 47
 colon 47
 `no_at_macros` 47
 period 48
 reorder 48
 space 48
`org` assembler control directive 46

P

period keyword 48
PowerPC Assembler 10
`pragma` assembler control directive 46
Prefix file field 62
`previous` section control directive 36
`pss` section control directive 35
`public` scope control directive 40

R

release notes 7
reorder **keyword** 48
rodata **section control directive** 35

S

sbss **section control directive** 36
scope control directives 39–40
 extern 40
 global 40
 public 40
scope, symbol 14
sdata **section control directive** 35
sdata2 **section control directive** 35
section control directives 34–39
 data 35
 debug 36
 offset 36
 previous 36
 pss 35
 rodata 35
 sbss 36
 sdata 35
 sdata2 35
 section 37
 text 35
section **section control directive** 37
set **symbol definition directive** 41
short **data declaration directive** 42
size **debugging directive** 50
space **data declaration directive** 43
space **keyword** 48
symbol
 definition 13, 14
 scope 14
symbol definition directives 40–41
 = (equal sign) 41
 equ 41
 set 41
 textequ 41
syntax
 assembly language statement 12–13
 comments 24
 constants 19–22
 expression 22–24
 forward equate 18–19
 symbol 14–18

T

text **section control directive** 35
textequ **symbol definition directive** 41
type **debugging directive** 50

U

unary operators 23
unique label symbol (\@) 56

