**Freescale Semiconductor, Inc.**

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

Copyright © Metrowerks Corporation. 2003. ALL RIGHTS RESERVED.

The reproduction and use of this document and related materials are governed by a license agreement media, it may be printed for non-commercial personal use only, in accordance with the license agreement related to the product associated with the documentation. Consult that license agreement before use or reproduction of any portion of this document. If you do not have a copy of the license agreement, contact your Metrowerks representative or call 800-377-5416 (if outside the US call +1-512-996-5300). Subject to the foregoing non-commercial personal use, 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.

Metrowerks reserves the right to make changes to any product described or referred to in this document without further notice. Metrowerks makes no warranty, representation or guarantee regarding the merchantability or fitness of its products for any particular purpose, nor does Metrowerks assume any liability arising out of the application or use of any product described herein and specifically disclaims any and all liability. **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 navigation or communication, emergency systems, or other applications with a similar degree of potential hazard.**

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

**How to Contact Metrowerks**

<table>
<thead>
<tr>
<th>Corporate Headquarters</th>
<th>Metrowerks Corporation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7700 West Parmer Lane</td>
</tr>
<tr>
<td></td>
<td>Austin, TX 78729</td>
</tr>
<tr>
<td></td>
<td>U.S.A.</td>
</tr>
</tbody>
</table>

| World Wide Web                        | http://www.metrowerks.com |

| Sales                                 | Voice: 800-377-5416       |
|                                      | Fax: 512-996-4910         |
|                                      | Email: sales@metrowerks.com |

| Technical Support                     | Voice: 800-377-5416       |
|                                      | Email: support@metrowerks.com |

For more information: [www.freescale.com](http://www.freescale.com)
# Contents

## 1 Introduction

1. Read the Release Notes! ........................................ 5
2. What’s in This Book ........................................ 5
3. Chapter Descriptions ........................................ 6
4. Code Examples. ........................................ 6
5. Conventions Used in This Manual ......................... 7
6. Where to Learn More ....................................... 7

## 2 Assembly Language Syntax

1. Assembly Language Statements Description ............ 9
2. Assembly Language Statement Syntax .................... 10
3. Symbols ..................................................... 11
4. Labels ....................................................... 12
5. Equates ..................................................... 14
6. Case-sensitive identifiers .................................. 16
7. Constants ................................................... 16
   - Integer Constants ....................................... 17
   - Floating-Point Constants ............................... 18
   - Character Constants .................................... 18
8. Expressions ................................................ 19
9. Comments ................................................... 21
10. Data Alignment ............................................ 22

## 3 Using Directives

1. Macro Directives .......................................... 23
2. Conditional Preprocessor Directives .................... 25
   - Other conditional preprocessor directives .......... 29
3. Section Control Directives ................................ 30
4. Scope Control Directives .................................. 35
5. Symbol Definition Directives ............................. 36
6. Data Declaration Directives .............................. 37

---

For more information: [www.freescale.com](http://www.freescale.com)
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer Directives</td>
<td>37</td>
</tr>
<tr>
<td>String Directives</td>
<td>38</td>
</tr>
<tr>
<td>Floating-Point Directives</td>
<td>39</td>
</tr>
<tr>
<td>Assembler Control Directives</td>
<td>40</td>
</tr>
<tr>
<td>Debugging Directives</td>
<td>43</td>
</tr>
<tr>
<td>Using Macros</td>
<td>47</td>
</tr>
<tr>
<td>Defining Macros</td>
<td>47</td>
</tr>
<tr>
<td>Macro Definition Syntax</td>
<td>47</td>
</tr>
<tr>
<td>Using Macro Arguments</td>
<td>50</td>
</tr>
<tr>
<td>Using Local Labels in a Macro</td>
<td>52</td>
</tr>
<tr>
<td>Creating Unique Labels and Equates</td>
<td>52</td>
</tr>
<tr>
<td>Referring to the Number of Arguments</td>
<td>53</td>
</tr>
<tr>
<td>Invoking Macros</td>
<td>53</td>
</tr>
<tr>
<td>Common Assembler Settings</td>
<td>55</td>
</tr>
<tr>
<td>Displaying Assembler Target Settings</td>
<td>55</td>
</tr>
<tr>
<td>Common Assembler Settings Descriptions</td>
<td>56</td>
</tr>
<tr>
<td>Labels must end with :</td>
<td>56</td>
</tr>
<tr>
<td>Directives begin with</td>
<td>57</td>
</tr>
<tr>
<td>Case-sensitive identifiers</td>
<td>57</td>
</tr>
<tr>
<td>Allow space in operand field</td>
<td>57</td>
</tr>
<tr>
<td>Generate listing file</td>
<td>58</td>
</tr>
<tr>
<td>Prefix file</td>
<td>58</td>
</tr>
<tr>
<td>PowerPC-Specific Information</td>
<td>59</td>
</tr>
<tr>
<td>Related Documentation</td>
<td>59</td>
</tr>
<tr>
<td>PowerPC-Specific Examples</td>
<td>59</td>
</tr>
<tr>
<td>Index</td>
<td>65</td>
</tr>
</tbody>
</table>
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.

For more information: www.freescale.com
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>
<thead>
<tr>
<th>Chapter Title</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>This chapter, which describes this manual.</td>
</tr>
<tr>
<td>Assembly Language Syntax</td>
<td>Describes the main syntax of assembly language statements.</td>
</tr>
<tr>
<td>Using Directives</td>
<td>Describes the assembler directives.</td>
</tr>
<tr>
<td>Using Macros</td>
<td>Describes how to define and invoke macros.</td>
</tr>
<tr>
<td>Common Assembler Settings</td>
<td>Describes the assembler settings that are common among the assemblers</td>
</tr>
</tbody>
</table>

### Code Examples

The code examples in the general chapters of this manual (Table 1.1 on page 6) 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

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>literal</td>
<td>Include the item in your statement as shown.</td>
</tr>
<tr>
<td>metasymbol</td>
<td>Replace the symbol with an appropriate value.</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>[a]</td>
<td>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. 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 ).</td>
</tr>
</tbody>
</table>

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.


The *PowerPC Assembler* supports all instructions for the Freescale MPC8xx, Freescale MPC505, and the IBM little-endian code. It generates only 32-bit code.

For more information: www.freescale.com
Freescale Semiconductor, Inc.

For more information: www.freescale.com
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:

For more information: www.freescale.com
• 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

<table>
<thead>
<tr>
<th>Syntax Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>symbol</td>
<td>A symbol is a combination of characters that represents a value. For more information, see “Symbols” on page 11.</td>
</tr>
<tr>
<td>machine_instruction</td>
<td>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.</td>
</tr>
<tr>
<td>assembler_directive</td>
<td>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”.</td>
</tr>
</tbody>
</table>

For more information: www.freescale.com
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 on page 10.

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” on page 13.

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
```
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>
<thead>
<tr>
<th>This...</th>
<th>Represents this</th>
</tr>
</thead>
<tbody>
<tr>
<td>label</td>
<td>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.</td>
</tr>
<tr>
<td>label@l</td>
<td>The low 16-bits of the relocated address of the symbol.</td>
</tr>
<tr>
<td>label@h</td>
<td>The high 16-bits of the relocated address of the symbol. You can OR this with label@l to produce the full 32-bit relocated address.</td>
</tr>
<tr>
<td>label@ha</td>
<td>The adjusted high 16-bits of the relocated address of the symbol. You can add this to label@l to produce the full 32-bit relocated address.</td>
</tr>
<tr>
<td>label@sdax</td>
<td>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.</td>
</tr>
<tr>
<td>label@got</td>
<td>For chips with a global offset table, the offset from the base of the global offset table to the 32-bit entry for label.</td>
</tr>
</tbody>
</table>

**Equates**

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

For more information: www.freescale.com
This section contains the following topics:

- Equate names
- Forward Equates

Equate names

The first character of an equate 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

```assembly
.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

```assembly
;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

<table>
<thead>
<tr>
<th>For numbers of this type...</th>
<th>Use...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal</td>
<td>A string of decimal digits, such as 12345678.</td>
</tr>
<tr>
<td>Hexadecimal</td>
<td>A dollar sign ($) followed by a string of hexadecimal digits, such as $deadbeef.</td>
</tr>
<tr>
<td>Binary</td>
<td>A percent sign (%) followed by a string of binary digits, such as %01010001.</td>
</tr>
</tbody>
</table>

To help you port existing code, the assembler also supports the notation in Table 2.4.

Table 2.4 Alternate integer constant notation

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

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>
<thead>
<tr>
<th>Sequence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>\b</code></td>
<td>Backspace</td>
</tr>
<tr>
<td><code>\n</code></td>
<td>Line feed (ASCII character 10)</td>
</tr>
<tr>
<td><code>\r</code></td>
<td>Return (ASCII character 13)</td>
</tr>
<tr>
<td><code>\t</code></td>
<td>Tab</td>
</tr>
<tr>
<td><code>\&quot;</code></td>
<td>Double quote</td>
</tr>
<tr>
<td><code>\</code></td>
<td>Backslash</td>
</tr>
<tr>
<td><code>\nnn</code></td>
<td>Octal value of <code>\nnn</code></td>
</tr>
</tbody>
</table>

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

For more information: www.freescale.com
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>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>add</td>
</tr>
<tr>
<td>-</td>
<td>subtract</td>
</tr>
<tr>
<td>*</td>
<td>multiply</td>
</tr>
<tr>
<td>/</td>
<td>divide</td>
</tr>
<tr>
<td>%</td>
<td>modulo</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>logical AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;</td>
<td>bitwise AND</td>
</tr>
<tr>
<td>^</td>
<td>bitwise XOR</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>shift left</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>shift right (zeros are shifted into high order bits)</td>
</tr>
<tr>
<td>==</td>
<td>equal to</td>
</tr>
<tr>
<td>!=</td>
<td>not equal to</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
</tr>
<tr>
<td>=&gt;</td>
<td>greater than or equal to</td>
</tr>
</tbody>
</table>
The assembler supports the unary operators listed in Table 2.7.

Table 2.7 Unary operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>unary plus</td>
</tr>
<tr>
<td>-</td>
<td>unary minus</td>
</tr>
<tr>
<td>~</td>
<td>unary bitwise complement</td>
</tr>
</tbody>
</table>

The assembler also supports the operations listed in Table 2.8.

Table 2.8 Alternate operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&gt;</td>
<td>not equal to</td>
</tr>
<tr>
<td>%</td>
<td>modulo</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

1. unary + - ~
2. * / %
3. binary + -
4. << >>
5. < <= > >=
6. == !=
7. &
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 colon 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: www.freescale.com
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”.

For more information: www.freescale.com
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
#elif
#else
#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 .endif directive.

This section discusses the following topics:

- `if`
- `ifdef`
- `ifndef`
- `ifc`
- `ifnc`
- `endif`
- `elseif`
- `else`
- Other conditional preprocessor directives

```
.if
  .if bool-exp
```

Specifies the beginning of a conditional assembly block, where `bool-exp` is a Boolean expression. If `bool-exp` 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

```c
#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

```c
#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

```c
.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.
Using Directives
Conditional Preprocessor Directives

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-exp

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-exp statement-group
[ .elseif bool-exp statement-group ]...
[ .else statement-group ]
.endif

In the preceding syntax statement, bool-exp is any Boolean expression and statement-group is any group of assembly language statements.
Using Directives
Conditional Preprocessor Directives

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.

For more information: www.freescale.com
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.

`.ifle (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.

For more information: www.freescale.com
If string1 is not less than or equal to string2, the assembler skips over all associated statements.

.ift (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

For more information: www.freescale.com
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.

ddebug
  .debug

For more information: www.freescale.com
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” on page 43.

**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:

**Table 3.1 Directives within a record**

<table>
<thead>
<tr>
<th>.equ</th>
<th>.set</th>
<th>.texteque</th>
</tr>
</thead>
<tbody>
<tr>
<td>.align</td>
<td>.org</td>
<td>.space</td>
</tr>
<tr>
<td>.byte</td>
<td>.short</td>
<td>.long</td>
</tr>
<tr>
<td>.space</td>
<td>.ascii</td>
<td>.asciz</td>
</tr>
<tr>
<td>.float</td>
<td>.double</td>
<td></td>
</tr>
</tbody>
</table>

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
```

For more information: www.freescale.com
rectSize .equ *

section

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

```
.sectio 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 3.2 describes the syntax elements for the ELF .section directive.

<table>
<thead>
<tr>
<th>Syntax Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The name of the section.</td>
</tr>
<tr>
<td>alignment</td>
<td>Specifies the alignment boundary of the section.</td>
</tr>
<tr>
<td>type</td>
<td>Numeric value for the ELF section type, per Table 3.3 on page 33. The default type value is SHT_PROGBITS.</td>
</tr>
<tr>
<td>flags</td>
<td>Numeric value for the ELF section flags, per Table 3.4 on page 35. The default flags value is SHF_ALLOC+SHF_WRITE.</td>
</tr>
</tbody>
</table>

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

```
.sectio vector,4
```

Table 3.3 defines the ELF section types.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>Indicates that the section header is inactive.</td>
</tr>
<tr>
<td>1</td>
<td>PROGBITS</td>
<td>Indicates that the section contains information defined by the program.</td>
</tr>
</tbody>
</table>

For more information: www.freescale.com
Table 3.3 ELF section types

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>SYMTAB</td>
<td>Indicates that the section contains a symbol table.</td>
</tr>
<tr>
<td>3</td>
<td>STRTAB</td>
<td>Indicates that the section contains a string table.</td>
</tr>
<tr>
<td>4</td>
<td>RELA</td>
<td>Indicates that the section contains relocation entries with explicit addends.</td>
</tr>
<tr>
<td>5</td>
<td>HASH</td>
<td>Indicates that the section contains a symbol hash table.</td>
</tr>
<tr>
<td>6</td>
<td>DYNAMIC</td>
<td>Indicates that the section contains information used for dynamic linking.</td>
</tr>
<tr>
<td>7</td>
<td>NOTE</td>
<td>Indicates that the section contains information that marks the file, often for compatibility purposes between programs.</td>
</tr>
<tr>
<td>8</td>
<td>NOBITS</td>
<td>Indicates that the section occupies no space in the object file.</td>
</tr>
<tr>
<td>9</td>
<td>REL</td>
<td>Indicates that the section contains relocation entries without explicit addends.</td>
</tr>
<tr>
<td>10</td>
<td>SHLIB</td>
<td>Indicates that the section has unspecified semantics and, therefore, does not conform to the Application Binary Interface (ABI) standard.</td>
</tr>
<tr>
<td>11</td>
<td>DYNDSYM</td>
<td>Indicates that the section contains a minimal set of symbols used for dynamic linking.</td>
</tr>
</tbody>
</table>

Table 3.4 defines the ELF section flags.

For more information: www.freescale.com
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`

```
.globl 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 ] ...
```

---

*For more information: [www.freescale.com](http://www.freescale.com)*
Using Directives

Symbol Definition Directives

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.

For more information: www.freescale.com
equ

\textit{equate} \texttt{.equ expression}

Permanently assigns the value \textit{expression} to \textit{equate}. You cannot change the value of \textit{equate} after defining it.

\texttt{textequ}

\textit{equate} \texttt{.textequ "string"}

Substitutes \textit{equate} with the text you specify in \textit{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 \textit{equate}, the assembler replaces it with \textit{string} before performing any other processing on that source line. Listing 3.2 shows examples of \texttt{.textequ} statements.

\begin{verbatim}
Listing 3.2  textequ examples
   dc.b    .textequ    ".byte"
   endc    .textequ    ".endif"
\end{verbatim}

\section*{Data Declaration Directives}

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

- Integer Directives
- String Directives
- Floating-Point Directives

\section*{Integer Directives}

The following directives initialize blocks of integer data:

- \texttt{byte}
- \texttt{short}
- \texttt{long}
- \texttt{space}
- \texttt{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 3.5.

For more information: www.freescale.com
Using Directives

Data Declaration Directives

**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.

---

For more information: www.freescale.com
**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` (big | little)
- `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`

For more information: www.freescale.com
Sets the assembler options as described in Table 3.6. 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 3.6 Option keywords

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alignment off</td>
<td>on</td>
</tr>
<tr>
<td>branchsize 8</td>
<td>16</td>
</tr>
<tr>
<td>case off</td>
<td>on</td>
</tr>
<tr>
<td>colon off</td>
<td>on</td>
</tr>
<tr>
<td>no_at_macros off</td>
<td>on</td>
</tr>
</tbody>
</table>
| 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 .”.

For more information: www.freescale.com
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).

### Table 3.6 Option keywords

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reorder off</td>
<td>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. This option keyword string applies only to the MIPS Assembler.</td>
</tr>
<tr>
<td>space off</td>
<td>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”.)</td>
</tr>
</tbody>
</table>

### 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.

**For more information:** www.freescale.com
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.

For more information: www.freescale.com
**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

For more information: www.freescale.com
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

<table>
<thead>
<tr>
<th>name: .macro [ parameter ] [ ,parameter ] ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>macro_body</td>
</tr>
<tr>
<td>.endm</td>
</tr>
</tbody>
</table>

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

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

```assembly
#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
      .mexit # use compact instruction
```
Using Macros
Defining Macros

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 on page 49.

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 [ ; ] [ \ ]
Using Macros

Defining Macros

assembly_statement [ ; ] [ \ ]

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

<table>
<thead>
<tr>
<th>Syntax Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>A label used to invoke the macro.</td>
</tr>
<tr>
<td>parameter</td>
<td>Operands that are passed to the macro.</td>
</tr>
<tr>
<td>assembly_statement</td>
<td>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. 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.</td>
</tr>
</tbody>
</table>

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  _final_setup
Listing 4.7 shows one way to invoke the `setup` macro.

**Listing 4.7  Calling setup**

```assembly
#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**

```assembly
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**

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

Listing 4.10 shows one way to invoke the `smallnum` macro.

**Listing 4.10  Invoking smallnum**

```assembly
smallnum 10
```

Listing 4.11 shows how the assembler expands the `smallnum` macro after the preceding call.
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 myMacro macro twice (with my_count initialized to 0).

Listing 4.13 Invoking my_MACRO

```
my_count  .set 0
my_MACRO
my_MACRO
```

For more information: www.freescale.com
Listing 4.14 shows the expanded my_macro code after the calls in Listing 4.13 on page 52.

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

For more information: www.freescale.com
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**

```assembly
.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 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.
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.
Common Assembler Settings
Common Assembler Settings Descriptions

(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.

---

For more information: www.freescale.com
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 Freescale, 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>
<thead>
<tr>
<th>Original Example</th>
<th>PowerPC-Specific Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Listing 2.2</td>
<td>Listing 6.1 on page 60</td>
</tr>
<tr>
<td>Listing 4.2</td>
<td>Listing 6.2 on page 60</td>
</tr>
</tbody>
</table>

For more information: www.freescale.com
Table 6.1  Corresponding example table

<table>
<thead>
<tr>
<th>Original Example</th>
<th>PowerPC-Specific Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Listing 4.3</td>
<td>Listing 6.3 on page 61</td>
</tr>
<tr>
<td>Listing 4.4</td>
<td>Listing 6.4 on page 61</td>
</tr>
<tr>
<td>Listing 4.6</td>
<td>Listing 6.5 on page 61</td>
</tr>
<tr>
<td>Listing 4.8</td>
<td>Listing 6.7 on page 62</td>
</tr>
<tr>
<td>Listing 4.12</td>
<td>Listing 6.8 on page 62</td>
</tr>
<tr>
<td>Listing 4.14</td>
<td>Listing 6.10 on page 63</td>
</tr>
</tbody>
</table>

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

For more information: www.freescale.com
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**

```assembly
; specify an executable code section
.text
xor r3,r3,r2
; call the addto macro
add to r3,0
add to r3,1
add to r3,2
add to 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**

```assembly
.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**

```assembly
setup: .macro name
lis r3, name@h
```

For more information: www.freescale.com
PowerPC-Specific Information

62

Assembler Reference

Freescale Semiconductor, Inc.

for more information: www.freescale.com
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
```
Symbols
= (equal sign) symbol definition directive 36
@ (at-sign) 13
\@ (unique label symbol) 52

A
align assembler control directive 40
alignment keyword 42
Allow space in operand field 21
alternate operators 20
ascii data declaration directive 39
asciz data declaration directive 39
assembler control directives 40–42
align 40
derror 40
error 40
include 41
option 41
org 41
pragma 41
at-sign (@) 13

B
binary operators 19
branchsize keyword 42
byte data declaration directive 37

C
case keyword 42
Case-sensitive identifiers checkbox 16, 42, 57
center constants 18
colon keyword 42
comments statement syntax 21
conditional directives 25–30
everse 28
elseif 27
endif 27
if 25
ifc 26
ifdef 26
ifeq 29
ifge 30
ifgt 30
ifle 29
iflt 29
ifnc 27
ifndef 26
ifne 29
constants
character 18
floating-point 18
integer 17

d
data declaration directives 37–40
ascii 39
asciz 39
byte 37
double 39
fill 38
float 39
long 38
short 38
space 38
data section control directive 31
debug section 22
debug section control directive 31
debugging directives 43–45
file 43
function 44
line 44
size 44
type 45
#define macro directive 24
defining macros 47–53
Directives begin with `.' checkbox 23, 42, 57
double data declaration directive 39

E
ELF
section flags 34
section types 33
syntax of section directive 33
everse conditional directive 28
elseif conditional directive 27
endian assembler control directive 40

For more information: www.freescale.com
endif conditional directive 27
endm macro directive 24
equ symbol definition directive 36
equates
   creating unique 52–??
   definition 14
eerror assembler control directive 40
eextern scope control directive 35

F
file debugging directive 43
fill data declaration directive 38
float data declaration directive 39
floating-point constants 18
forward equates, definition 15
function debugging directive 44

G
Generate listing file checkbox 58
global scope control directive 35

I
if conditional directive 25
ifc conditional directive 26
ifdef conditional directive 26
ifeq conditional directive 29
ifge conditional directive 30
ifgt conditional directive 30
ifle conditional directive 29
iflt conditional directive 29
ifnc conditional directive 27
ifndef conditional directive 26
ifne conditional directive 29
include assembler control directive 41
integer constants 17

L
labels
   creating unique 52–??
   definition 12
   Labels must end with ':' checkbox 12, 42, 56
line debugging directive 44
literal 7
local label 13
long data declaration directive 38

M
macro directive 24
macro directives 23–??
   #define directive 24
   endm directive 24
   macro directive 24
   mexit directive 24
macros
   arguments 50
   defining with the #define directive 49
   defining with the .macro directive 48
   invoking 53–54
   local labels in 52
   macro definition syntax 47
   number of arguments (narg) 53
   unique equates in 52
   unique labels in 52
metasymbol 7
mexit macro directive 24

N
no_at_macros keyword 42

O
offset section control directive 32
option 22
option assembler control directive 41
option keywords
   alignment 42
   branchsize 42
   case 42
   colon 42
   no_at_macros 42
   period 42
   reorder 43
   space 43
org assembler control directive 41

P
period keyword 42
PowerPC Assembler 7
pragma assembler control directive 41
Prefix file field 58
previous section control directive 32
pss section control directive 31
public scope control directive 36

For more information: www.freescale.com
<table>
<thead>
<tr>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>release notes 5</td>
</tr>
<tr>
<td>reorder keyword 43</td>
</tr>
<tr>
<td>rodata section control directive 31</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>sbss section control directive 31</td>
</tr>
<tr>
<td>scope control directives 35–36</td>
</tr>
<tr>
<td>extern 35</td>
</tr>
<tr>
<td>global 35</td>
</tr>
<tr>
<td>public 36</td>
</tr>
<tr>
<td>scope, symbol 11</td>
</tr>
<tr>
<td>sdata section control directive 31</td>
</tr>
<tr>
<td>sdata2 section control directive 31</td>
</tr>
<tr>
<td>section control directives 30–35</td>
</tr>
<tr>
<td>data 31</td>
</tr>
<tr>
<td>debug 31</td>
</tr>
<tr>
<td>offset 32</td>
</tr>
<tr>
<td>previous 32</td>
</tr>
<tr>
<td>pss 31</td>
</tr>
<tr>
<td>rodata 31</td>
</tr>
<tr>
<td>sbss 31</td>
</tr>
<tr>
<td>sdata 31</td>
</tr>
<tr>
<td>sdata2 31</td>
</tr>
<tr>
<td>section 33</td>
</tr>
<tr>
<td>text 31</td>
</tr>
<tr>
<td>section control directive 33</td>
</tr>
<tr>
<td>set symbol definition directive 36</td>
</tr>
<tr>
<td>short data declaration directive 38</td>
</tr>
<tr>
<td>size debugging directive 44</td>
</tr>
<tr>
<td>space data declaration directive 38</td>
</tr>
<tr>
<td>space keyword 43</td>
</tr>
<tr>
<td>symbol</td>
</tr>
<tr>
<td>definition 10, 11</td>
</tr>
<tr>
<td>scope 11</td>
</tr>
<tr>
<td>symbol definition directives 36–37</td>
</tr>
<tr>
<td>= (equal sign) 36</td>
</tr>
<tr>
<td>equ 36</td>
</tr>
<tr>
<td>set 36</td>
</tr>
<tr>
<td>textequ 37</td>
</tr>
<tr>
<td>syntax</td>
</tr>
<tr>
<td>assembly language statement 10–11</td>
</tr>
<tr>
<td>comments 21</td>
</tr>
<tr>
<td>constants 16–18</td>
</tr>
<tr>
<td>expression 19–21</td>
</tr>
<tr>
<td>forward equate 15–16</td>
</tr>
<tr>
<td>symbol 11–15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>text section control directive 31</td>
</tr>
<tr>
<td>textequ symbol definition directive 37</td>
</tr>
<tr>
<td>type debugging directive 45</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>unary operators 20</td>
</tr>
<tr>
<td>unique label symbol (@) 52</td>
</tr>
</tbody>
</table>