CodeWarrior™ Development Studio

Assembler Reference for ColdFire® Processors

Revised: 19 October 2006

freescale semiconductor

How to Contact Us

Corporate Headquarters	Freescale Semiconductor, Inc. 7700 West Parmer Lane Austin, TX 78729 U.S.A.
World Wide Web	`http://www.freescale.com/codewarrior`
Technical Support	`http://www.freescale.com/support`

Table of Contents

1

Introduction

The CodeWarrior™ IDE includes assemblers that support several specific processors. This manual explains the corresponding assembly-language syntax and IDE settings for these assemblers.

Release Notes

Release notes contain important information about new features, bug fixes, and incompatibilities. Release notes reside in directory:

```
(CodeWarrior directory}\Release_Notes
```

In This Book

This manual explains the syntax for assembly-language statements that the CodeWarrior assemblers use. These explanations cover macros and directives, as well as simple statements.

NOTE For information on the *inline* assembler of the CodeWarrior C/C++ compiler, see the *Targeting* manual for your target processor or the *C Compilers Reference*.

All the assemblers share the same basic assembly-language syntax. but instruction mnemonics and register names are different for each target processor.

To get the most from this manual, you should be familiar with assembly language and with your target processor.

Unless otherwise stated, all the information in this manual applies to all the assemblers. Table 1.1 lists the *general* chapters of this manual — the chapters that pertain to all the assemblers. This manual also includes a chapter that is specific to your target processor.

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.

The code examples in the general chapters are for x86 processors. If the corresponding code is different for your target processor, the processor-specific chapter includes counterpart examples.

Where to Learn More

Each assembler uses the standard assembly-language mnemonics and register names that the processor manufacturer defines. The processor-specific chapter of this manual includes references to documents that provide additional information about your target processor.

2

Assembly Language Syntax

This chapter explains the syntax of assembly language statements. It consists of these topics:

Assembly Language Statements

The three types of assembly language statements are:

- Machine instructions
- Macro calls
- Assembler directives

Instructions, directives, and macro names are case insensitive: the assembler considers `MOV`, `Mov`, and `mov` to be the same instruction.

Remember these rules for assembly language statements:

1. A statement must reside on a single line; the maximum length of a statement is 512 characters.
2. You can concatenate two or more lines into one statement by typing a backslash (\) character at the end of lines. But such a concatenated statement must not exceed the 512-character limit.
3. There is no limit to macro expansion, but individual statements and concatenated statements must not exceed the 512-character limit.
4. Each line of the source file can contain only one statement unless the assembler is running in GNU mode. (This mode allows multiple statements on one line, with semicolon separators.)

The processor-specific chapter of this manual tells you where find machine instructions for your target processor. Other chapters of this manual provide more information about assembler directives and macros.

Statement Syntax

Listing 2.1 shows the syntax of an assembly language statement. Table 2.1 describes the elements of this syntax.

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 Syntax Elements

Element	Description
symbol	A combination of characters that represents a value.
machine_instruction	A machine instruction for your target processor.
assembler_directive	A special instruction that tells the assembler how to process other assembly language statements. For example, certain assembler directives specify the beginning and end of a macro.
macro_call	A statement that calls a previously defined macro.
constant	A defined value, such as a string of characters or a numeric value.
expression	A mathematical expression.
register_name	The name of a register; these names are processor-specific.
comment	Text that the assembler ignores, useful for documenting your code.

Symbols

A *symbol* is a group of characters that represents a value, such as an address, numeric constant, string constant, or character constant. There is no length limit to symbols.

The syntax of a symbol is:

```
symbol ::= label | equate
```

In general, symbols have file-wide scope. This means:

1. You can access the symbol from anywhere in the file that includes the symbol definition.
2. You cannot access the symbol from another file.

However, it is possible for symbols to have a different scope, as the Local Labels subsection explains.

Labels

A *label* is a symbol that represents an address. A label's scope depends on whether the label is local or non-local.

The syntax of a label is:

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

The default settings are that each label ends with a colon (:), a label can begin in any column. However, if you port existing code that does not follow this convention, you should clear the **Labels must end with ':'** checkbox of the Assembler settings panel. After you clear the checkbox, you may use labels that do not end with colons, but such labels must begin in column 1.

NOTE For more information, see the Common Assembler Settings chapter.

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 a:

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

Subsequent characters can be from the preceding list or a:

- numeral (0-9), or

- dollar sign ($).

Local Labels

A *local label* is a symbol that represents an address and has local scope: the range forward and backward within the file to the points where the assembler encounters non-local labels.

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

- letters (a-z or A-Z)
- numerals (0-9)
- underscores (_)
- question marks (?)
- dollar sign. ($)
- periods (.)

NOTE You cannot export local labels; 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: the `@SKIP` label defined in the macro does not conflict with the `@SKIP` label defined in the main body of code.

Listing 2.2 Local Label Scope 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
```

Relocatable Labels

The assembler assumes a flat 32-bit memory space. You can use the expressions of Table 2.2 to specify the relocation of a 32-bit label.

NOTE The assembler for your target processor may not allow all of these expressions.

Table 2.2 Relocatable Label Expressions

Expression	Represents
label	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.
label`@l`	The low 16-bits of the relocated address of the symbol.
label`@h`	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.
label`@ha`	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.
label`@sdax`	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.
label`@got`	For processors 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. To create an equate, use the `.equ` or `.set` directive.

The first character of an equate must be a:

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

Subsequent characters can be from the preceding list or a:

- numeral (0-9) or
- dollar sign ($)

The assembler allows *forward equates*. This means that a reference to an equate can be in a file before the equate's definition. When an assembler encounters such 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 any unresolved expressions. If necessary, the assembler repeatedly reevaluates expressions until it resolves them all or cannot resolve them any further. If the assembler cannot resolve an expression, it issues an error message.

NOTE The assembler must be able to resolve immediately any expression whose value affects the location counter.
If the assembler can make a reasonable assumption about the location counter, it allows the expression. For example, in a forward branch instruction for a 68K processor, you can specify a default assumption of 8, 16, or 32 bits.

The code of Listing 2.3 shows a valid forward equate.

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 of Listing 2.4 is not valid. The assembler cannot immediately resolve the expression in the `.space` directive, so 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 of the Assembler settings panel lets you control case-sensitivity for symbols:

- Check the checkbox to make symbols case sensitive — `SYM1`, `sym1`, and `Sym1` are three different symbols.
- Clear the checkbox to make symbols *not* case-sensitive — `SYM1`, `sym1`, and `Sym1` are the same symbol. (This is the default setting.)

Constants

The assembler recognizes three kinds of constants:

- Integer Constants
- Floating-Point Constants
- Character Constants

Integer Constants

Table 2.3 lists the notations for integer constants. Use the preferred notation for new code. The alternate notations are for porting existing code.

Table 2.3 Preferred Integer Constant Notation

Type	Preferred Notation	Alternate Notation
Hexadecimal	`0x` followed by a string of hexadecimal digits, such as `0xdeadbeef`.	`$` followed by string of hexadecimal digits, such as `$deadbeef`. (For certain processors, this is the preferred notation.)
		`0` followed by a string of hexadecimal digits, ending with `h`, such as `0deadbeefh`.
Decimal	String of decimal digits, such as `12345678`.	String of decimal digits followed by `d`, such as `12345678d`.
Binary	`%` followed by a string of binary digits, such as `%01010001`.	0b followed by a sting of binary digits, such as `0b01010001`.
		String of binary digits followed by `b`, such as `01010001b`.

NOTE The assembler uses 32-bit signed arithmetic to store and manipulate integer constants.

Floating-Point Constants

You can specify floating-point constants in either hexadecimal or decimal format. The decimal format must contain a decimal point or an exponent. Examples are `1E-10` and `1.0`.

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

Character Constants

Enclose a character constant in single quotes. However, if the character constant includes a single quote, use double quotes to enclose the character constant.

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. Examples are `'A'`, `'ABC'`, and `'TEXT'`.

A character constant can contain any of the escape sequences that Table 2.4 lists.

Table 2.4 Character Constant Escape Sequences

Sequence	Description
\b	Backspace
\n	Line feed (ASCII character 10)
\r	Return (ASCII character 13)
\t	Tab
\'	Single quote
\"	Double quote
\\	Backslash
\xnn	Hexadecimal value of nn
\nnn	Octal value of nn

During computation, the assembler zero-extends a character constant to 32 bits. You can use a character constant anywhere you can use an integer constant.

Expressions

The assembler uses 32-bit signed arithmetic to evaluates expressions; it does not check for arithmetic overflow.

As different processors use different operators, the assembler uses an expression syntax similar to that of the C language. Expressions use C operators and follow C rules for parentheses and associativity.

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

Table 2.5 lists the expression operators that the assembler supports.

Table 2.5 Expression Operators

Category	Operator	Description
Binary	+	add
	-	subtract
	*	multiply
	/	divide
	%	modulo
	||	logical OR
	&&	logical AND
	|	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
Unary	+	unary plus
	-	unary minus
	~	unary bitwise complement
Alternate	<>	not equal to

Operator precedence is:

1. ( )
2. @
3. unary + - ~ !
4. * / %
5. binary + -
6. << >>
7. < <= > >=
8. == !=
9. &
10. ^
11. |
12. &&
13. ||

(Gnu- or ADS-compatibility modes change some of these operator precedences.)

Comments

There are several ways to specify comments:

1. Use either type of C-style comment, which can start in any column:

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

2. Start the comment with an asterisk (*) in the first column of the line. Alternate comment specifiers, for compatibility with other assemblers, are #, .*, and --.

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

3. Use a processor-specific comment character anywhere on the line (the processor-specific chapter of this document explains whether such a character exists for your target processor). A 68K/Coldfire example is:

   ```
   move.l d0,d1          ;This is a comment
   ```

   A PowerPC example is;

   ```
   mr      r1,r0         #This is a comment
   ```

NOTE Gnu compatibility mode may involve a different comment character, and may involve a different meaning for the `;` character.

4. Clear the **Allow space in operand field** checkbox of the Assembler settings panel. Subsequently, if you type a space in an operand field, all the remaining text of the line is a comment.

Data Alignment

The assembler's default alignment is on a natural boundary for the data size and for the target processor family. To turn off this default alignment, use the `alignment` keyword argument with to the `.option` directive.

NOTE The assembler does not align data automatically in the `.debug` section.

3

Using Directives

This chapter explains available directives for the preprocessor and the main, or *native*, assembler. Remember these key points:

- Some directives may not be available for your target processor.
- The starting character for preprocessor directives is the hash or pound sign (#); the default starting character for native assembler directives is the period (.).
- Many preprocessor directives have native-assembler counterparts, but the directives of each set are not the same.

When you submit source files to the assembler, the code goes through the preprocessor. Then the preprocessor-output code goes through the native assembler. This leads to a general rule of not mixing preprocessor and native-assembler directives.

For example, consider the simple symbol-definition test of Listing 3.1:

Listing 3.1 Mixed-Directive Example

```
#define         ABC   MyVal
      .ifdef    ABC             ;Definition test
```

Before the native assembler sees this code, the C preprocessor converts the line `.ifdef ABC` to `.ifdef MyVal`. This means that the native assembler tests for a definition of `MyVal`, not `ABC`.

For a definition test of `ABC`, you should use either the preprocessor directives of Listing 3.2 or the native assembler syntax of Listing 3.3:

Listing 3.2 Preprocessor-Directive Example

```
#define         ABC   MyVal
#ifdef  ABC             ;Definition test
```

Listing 3.3 Native-Assembler-Directive Example

```
ABC   =   1
      .ifdef    ABC             ;Definition test
```

The sections of this chapter are:

Preprocessor Directives

Table 3.1 lists the preprocessor directives. Explanations follow the table.

Table 3.1 Preprocessor Directives

Directive	Description
#define	Defines a preprocessor macro.
#elif	Starts an alternative conditional assembly block, with another condition.
#else	Starts an alternative conditional assembly block.
#endif	Ends a conditional assembly block.
#error	Prints the specified error message.
#if	Starts a conditional-assembly block.
#ifdef	Starts a symbol-defined conditional assembly block.
#ifndef	Starts a symbol-not-defined conditional assembly block.
#include	Takes input from the specified file.
#line	Specifies absolute line number.
#pragma	Uses setting of specified pragma.
#undefine	Removes the definition of a preprocessor macro.

#define

Defines a preprocessor macro.

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

Parameters

`name`

Name of the macro.

`parms`

List of parameters, separated by commas. Parentheses must enclose the list.

`assembly_statement`

Any valid assembly statement.

Remarks

To extend an *assembly_statement*, type a backslash (\) and continue the statement on the next line. In GNU mode, multiple statements can be on one line of code — separate them with semicolon characters (;).

#elif

Starts an optional, alternative conditional-assembly block, adding another boolean-expression condition.

```
#elif bool-expr statement-group
```

Parameters

`bool-expr`

Any boolean expression.

`statement-group`

Any valid assembly statements.

Remarks

This directive must be part of an `#if ... #elif ... [#else] ... #endif` conditional structure (with each of these directives starting a new line). The preprocessor implements the assembly statements that `#elif` introduces only if (1) the bool-expr condition of the #if directive is *false*, and (2) the bool-expr condition of the #elif directive is *true*.

For a logical structure of multiple levels, you can use the `#elif` directive several times, as in this pattern:

```
#if bool-expr-1
    statement-group-1
#elif bool-expr-2
    statement-group-2
#elif bool-expr-3
    statement-group-3
#elif bool-expr-4
    statement-group-4
#else
    statement-group-5
#endif
```

- If this structure's `bool-expr-1` is true, the preprocessor executes the `statement-group-1` statements, then goes to the #endif directive.
- If `bool-expr-1` is false, the preprocessor skips `statement-group-1`, executing the first `#elif` directive. If `bool-expr-2` is true, the preprocessor executes `statement-group-2`, then goes to the `#endif` directive.
- If `bool-expr-2` also is false, the preprocessor skips `statement-group-2`, executing the *second* `#elif` directive.
- The preprocessor continues evaluating the boolean expressions of succeeding `#elif` directives until it comes to a boolean expression that is true.
- If none of the boolean expressions are true, the preprocessor processes `statement-group-5`, because this structure includes an `#else` directive.
- If none of the boolean values were true and there were no `#else` directive, the preprocessor would not process any of the statement groups.)

#else

Starts an optional, alternative conditional assembly block.

```
#else statement-group
```

Parameter

`statement-group`

Any valid assembly statements.

Remarks

This directive must be part of an `#if ... [#elif] ... #else ... #endif` conditional structure (with each of these directives starting a new line). The preprocessor implements the assembly statements that `#else` introduces *only if* the bool-expr condition of the `#if` directive is *false*.

If this directive is part of a conditional structure that includes several `#elif` directives, the preprocessor implements the assembly statements that `#else` introduces only if *all* the bool-expr conditions are *false*.

#endif

Ends a conditional assembly block; mandatory for each `#if`, `#ifdef`, and `#ifndef` directive.

```
.endif
```

#error

Prints the specified error message to the IDE Errors and Warnings window.

```
#error "message"
```

Parameter

`message`

Error message, in double quotes.

#if

Starts a conditional assembly block, making assembly conditional on the truth of a boolean expression.

```
#if bool-expr statement-group
```

Parameters

`bool-expr`

Any boolean expression.

`statement-group`

Any valid assembly statements.

Remarks

This directive starts an `#if ... [#elif] ... [#else] ... #endif` conditional structure (with each of these directives starting a new line). There must be a corresponding `#endif` directive for each `#if` directive. An `#else` directive is optional; one or more `#elif` directives are optional.

The simplest such conditional structure follows the pattern `#if ... assembly statements ... #endif`. The preprocessor implements the assembly statements only if the `#if` directive's bool-expr condition is *true*.

The next simplest conditional structure follows the pattern `#if ... assembly statements 1 ... #else ... assembly statements 2 ... #endif`. The preprocessor implements the assembly statements 1 if the `#if` directive's bool-expr condition is *true*; the preprocessor implements assembly statements 2 if the condition is *false*.

You can use `#elif` directives to create increasingly complex conditional structures.

#ifdef

Starts a conditional assembly block, making assembly conditional on the definition of a symbol.

```
#ifdef symbol statement-group
```

Parameters

`symbol`

Any valid symbol.

`statement-group`

Any valid assembly statements.

Remarks

If previous code includes a definition for `symbol`, the preprocessor implements the statements of the block. If `symbol` is not defined, the preprocessor skips the statements of the block.

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

#ifndef

Starts a conditional assembly block, making assembly conditional on a symbol *not* being defined.

```
#ifndef symbol statement-group
```

Parameter

`symbol`

Any valid symbol.

`statement-group`

Any valid assembly statements.

Remarks

If previous code does *not* include a definition for `symbol`, the preprocessor implements the statements of the block. If there *is* a definition for `symbol`, the preprocessor skips the statements of the block.

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

#include

Tells the preprocessor to take input from the specified file.

```
#include filename
```

Parameter

`filename`

Name of an input file.

Remarks

When the preprocessor reaches the end of the specified file, it takes input from the assembly statement line that follows the `#include` directive. The specified file itself can contain an `#include` directive that specifies yet another input file.

#line

Specifies the absolute line number (of the current source file) for which the preprocessor generates subsequent code or data.

```
#line number
```

Parameter

`number`

Line number of the file; the file's first line is number 1.

#pragma

Tells the assembler to use a particular pragma setting as it assembles code.

```
#pragma pragma-type setting
```

Parameters

`pragma-type`

Type of pragma.

`setting`

Setting value.

#undefine

Removes the definition of a preprocessor macro.

```
#undefine name
```

Parameters

`name`

Name of the macro.

Native Assembler Directives

The default starting character for native assembler directives is the period (.). But you can omit this starting period if you clear the **Directives begin with '.'** checkbox of the Assembler settings panel.

Table 3.2 lists these directives by type. Explanations of the directives follow the table, in alphabetic order.

Table 3.2 Assembler Directives

Type	Directive	Description
Macro	.endm	Ends a macro definition.
	.macro	Starts a macro definition.
	.mexit	Ends macro execution early.
Conditional	.else	Starts an alternative conditional assembly block.
	.elseif	Starts an alternative conditional assembly block, adding another condition.
	.endif	Ends a conditional assembly block.
	.if	Starts a conditional assembly block.
	.ifc	Starts a 2-strings-equal conditional assembly block.
	.ifdef	Starts a symbol-defined conditional assembly block
	.ifnc	Starts a 2-strings-not-equal conditional assembly block.
	.ifndef	Starts a symbol-not-defined conditional assembly block.

Table 3.2 Assembler Directives (*continued*)

Type	Directive	Description
Compatibility Conditional	.ifeq	Starts a string-equals-0 conditional assembly block.
	.ifge	Starts a string->=-0 conditional assembly block.
	.ifgt	Starts a string->-0 conditional assembly block.
	.ifle	Starts a string-<=-0 conditional assembly block.
	.iflt	Starts a string-<-0 conditional assembly block.
	.ifne	Starts a string-not-equals-0 conditional assembly block.
Section Control	.bss	Specifies an unititialized, read-only data section.
	.data	Specifies an initialized, read-write data section.
	.debug	Specifies a debug section.
	.offset	Starts a record definition.
	.previous	Reverts to the previous section.
	.rodata	Specifies an initialized, read-only data section.
	.sbss	Specifies an uninitialized, read-write small data section.
	.sbss2	Specifies an uninitialized, read-write small data section.
	.sdata	Specifies an initialized, read-write small data section.
	.sdata0	Specifies an initialized, read-write small data section.
	.sdata2	Specifies an initialized, read-only small data section.
	.section	Defines an ELF object-file section.
	.text	Specifies an executable code section.

Table 3.2 Assembler Directives (*continued*)

Type	Directive	Description
Scope Control	.extern	Imports specified labels.
	.global	Exports specified labels.
	.public	Declares specified labels public.
Symbol Definition	.equ	Defines an equate; assigns a permanent value.
	equal sign (=)	Defines an equate; assigns an initial value.
	.set	Defines an equate.
	.textequ	Defines an equate; assigns a string value.
Data Declaration	.ascii	Declares a storage block for a string.
	.asciz	Declares a 0-terminated storage block for a string.
	.byte	Declares an initialized block of bytes.
	.double	Declares an initialized block of 64-bit, floating-point numbers.
	.float	Declares an initialized block of 32-bit, floating-point numbers.
	.long	Declares an initialized block of 32-bit short integers.
	.short	Declares an initialized block of 16-bit short integers.
	.space	Declares a 0-initialized block of bytes.
Assembler Control	.align	Aligns location counter to specified power of 2.
	.endian	Specifies target-processor byte ordering.
	.error	Prints specified error message.
	.include	Takes input from specified file.
	.option	Sets an option.
	.org	Changes location-counter value.
	.pragma	Uses setting of specified pragma.

Table 3.2 Assembler Directives (*continued*)

Type	Directive	Description
Debugging	.file	Specifies source-code file.
	.function	Generates debugging data.
	.line	Specifies absolute line number.
	.size	Specifies symbol length.
	.type	Specifies symbol type.

.align

Aligns the location counter on the specified value.

```
.align expression
```

Parameter

`expression`

Alignment value.

Remarks

The `expression` value is the *actual* alignment value, so `.align 2` specifies 2-byte alignment. (For certain other assemblers, `expression` is an *exponent* for 2, so `.align 2` would specify 4-byte alignment.)

.ascii

Declares a block of storage for a string; the assembler allocates a byte for each character.

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

Parameters

`label`

Name of the storage block.

`string`

String value to be stored, in double quotes. This string can contain any of the escape sequences that Table 3.3 lists.

Table 3.3 Escape Sequences

Sequence	Description
\b	Backspace
\n	Line feed (ASCII character 10)
\r	Return (ASCII character 13)
\t	Tab
\'	Single quote
\"	Double quote
\\	Backslash
\nnn	Octal value of \nnn
\xnn	Hexadecimal value of nn

.asciz

Declares a zero-terminated block of storage for a string.

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

Parameters

`label`

Name of the storage block.

`string`

String value to be stored, in double quotes. This string can contain any of the escape sequences that Table 3.4 lists.

Table 3.4 Escape Sequences

Sequence	Description
\b	Backspace
\n	Line feed (ASCII character 10)
\r	Return (ASCII character 13)
\t	Tab
\'	Single quote
\"	Double quote
\\	Backslash
\nnn	Octal value of \nnn
\xnn	Hexadecimal value of nn

Remarks

The assembler allocates a byte for each `string` character. The assembler then allocates an extra byte at the end, initializing this extra byte to zero.

.bss

Specifies an uninitialized read-write data section.

```
.bss
```

.byte

Declares an initialized block of bytes.

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

Parameters

`label`

Name of the block of bytes.

`expression`

Value for one byte of the block; must fit into one byte.

.data

Specifies an initialized read-write data section.

```
.data
```

.debug

Specifies a debug section.

```
.debug
```

Remarks

This directive is appropriate if you must provide certain debugging information explicitly, in a debug section. But this directive turns *off* automatic generation of debugging information (which the assembler does if you enable the debugger). Furthermore, this directive tells the assembler to ignore the debugging directives `.file`, `.function`, `.line`, `.size`, and `.type`.

As Providing Debugging Information explains, using the .debug directive may be the least common method of providing debugging information to the assembler.

.double

Declares an initialized block of 64-bit, floating-point numbers; the assembler allocates 64 bits for each value.

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

Parameters

`label`

Name of the storage block.

`value`

Floating-point value; must fit into 64 bits.

.else

Starts an optional, alternative conditional assembly block.

```
.else statement-group
```

Parameter

`statement-group`

Any valid assembly statements.

Remarks

This directive must be part of an `.if ... [.elseif] ... .else ... .endif` conditional structure (with each of these directives starting a new line). The assembler processes the assembly statements that `.else` introduces *only if* the bool-expr condition of the `.if` directive is *false*.

If this directive is part of a conditional structure that includes several `.elseif` directives, the assembler processes the assembly statements that `.else` introduces only if *all* the bool-expr conditions are *false*.

.elseif

Starts an optional, alternative conditional assembly block, adding another boolean-expression condition.

```
.elseif bool-expr statement-group
```

Parameters

`bool-expr`

Any boolean expression.

`statement-group`

Any valid assembly statements.

Remarks

This directive must be part of an `.if ... .elseif ... [.else] ... .endif` conditional structure (with each of these directives starting a new line). The assembler processes the assembly statements that `.elseif` introduces *only if* (1) the bool-expr condition of the `.if` directive is *false*, and (2) the bool-expr condition of the `.elseif` directive is *true*.

For a logical structure of multiple levels, you can use the `.elseif` directive several times, as in this pattern:

```
.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
.endif
```

- If this structure's `bool-expr-1` is true, the assembler executes the `statement-group-1` statements, then goes to the `.endif` directive.
- If `bool-expr-1` is false, the assembler skips `statement-group-1`, executing the first `.elseif` directive. If `bool-expr-2` is true, the assembler executes `statement-group-2`, then goes to the `.endif` directive.
- If `bool-expr-2` also is false, the assembler skips `statement-group-2`, executing the *second* `.elseif` directive.
- The assembler continues evaluating the boolean expressions of succeeding `.elseif` directives until it comes to a boolean expression that is true.
- If none of the boolean expressions are true, the assembler processes `statement-group-5`, because this structure includes an `.else` directive.
- If none of the boolean values were true and there were no `.else` directive, the assembler would not process any of the statement groups.)

.endian

Specifies byte ordering for the target processor; valid only for processors that permit change of endianness.

```
.endian big | little
```

Parameters

`big`

Big-endian specifier.

`little`

Little-endian specifier.

.endif

Ends a conditional assembly block. A matching `.endif` directive is mandatory for each type of `.if` directive.

```
.endif
```

.endm

Ends the definition of a macro.

```
.endm
```

.equ

Defines an equate, assigning a permanent value. You cannot change this value at a later time.

```
equate .equ expression
```

Parameters

`equate`

Name of the equate.

`expression`

Permanent value for the equate.

equal sign (=)

Defines an equate, assigning an initial value. You can change this value at a later time.

```
equate = expression
```

Parameters

`equate`

Name of the equate.

`expression`

Temporary initial value for the equate.

Remarks

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

.error

Prints the specified error message to the IDE Errors and Warnings window.

```
.error "error"
```

Parameter

`error`

Error message, in double quotes.

.extern

Tells the assembler to *import* the specified labels, that is, find the definitions in another file.

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

Parameter

`label`

Any valid label.

Remarks

You cannot import equates or local labels.

An alternative syntax for this directive is `.extern section:label`, as in `.extern .sdata:current_line`. Some processor architectures require this alternative syntax to distinguish text from data.

.file

Specifies the source-code file; enables correlation of generated assembly code and source code.

```
.file "filename"
```

Parameter

`filename`

Name of source-code file, in double quotes.

Remarks

This directive is appropriate if you must explicitly provide a filename to the assembler *as debugging information*. Providing Debugging Information explains additional information about debugging.

Example

Listing 3.4 shows how to use the `.file` directive for your own DWARF code.

Listing 3.4 DWARF Code Example

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

.float

Declares an initialized block of 32-bit, floating-point numbers; the assembler allocates 32 bits for each value.

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

Parameters

`label`

Name of the storage block.

`value`

Floating-point value; must fit into 32 bits.

.function

Tells the assembler to generate debugging data for the specified subroutine.

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

Parameters

`func`

Subroutine name, in double quotes.

`label`

Starting label of the subroutine.

`length`

Number of bytes in the subroutine.

Remarks

This directive is appropriate if you must explicitly provide debugging information to the assembler. Providing Debugging Information explains additional information about debugging.

.global

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

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

Parameter

`label`

Any valid label.

Remarks

You cannot export equates or local labels.

.if

Starts a conditional assembly block, making assembly conditional on the truth of a boolean expression.

```
.if bool-expr statement-group
```

Parameters

`bool-expr`

Any boolean expression.

`statement-group`

Any valid assembly statements.

Remarks

This directive starts an `.if ... [.elseif] ... [.else] ... .endif` conditional structure (with each of these directives starting a new line). There must be a corresponding `.endif` directive for each `.if` directive. An `.else` directive is optional; one or more `.elseif` directives are optional.

The simplest such conditional structure follows the pattern `.if ... assembly statements ... .endif`. The preprocessor implements the assembly statements only if the `.if` directive's bool-expr condition is *true*.

The next simplest conditional structure follows the pattern `.if ... assembly statements 1 ... .else ... assembly statements 2 ... .endif`. The preprocessor implements the assembly statements 1 if the `.if`

directive's bool-expr condition is *true*; the preprocessor implements assembly statements 2 if the condition is *false*.

You can use `.elseif` directives to create increasingly complex conditional structures.

.ifc

Starts a conditional assembly block, making assembly conditional on the equality of two strings.

```
.ifc string1, string2 statement-group
```

Parameters

`string1`

Any valid string.

`string2`

Any valid string.

`statement-group`

Any valid assembly statements.

Remarks

If `string1` and `string2` are equal, the assembler processes the statements of the block. (The equality comparison is case-sensitive.) If the strings are *not* equal, the assembler skips the statements of the block.

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

.ifdef

Starts a conditional assembly block, making assembly conditional on the definition of a symbol.

```
.ifdef symbol statement-group
```

Parameters

`symbol`

Any valid symbol.

`statement-group`

Any valid assembly statements.

Remarks

If previous code includes a definition for `symbol`, the assembler processes the statements of the block. If `symbol` is not defined, the assembler skips the statements of the block.

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

.ifeq

Starts a conditional assembly block, making assembly conditional on an expression value being equal to zero.

```
.ifeq expression statement-group
```

Parameters

`expression`

Any valid expression.

`statement-group`

Any valid assembly statements

Remarks

If the `expression` value equals 0, the assembler processes the statements of the block. If the `expression` value does *not* equal 0, the assembler skips the statements of the block.

.ifge

Starts a conditional assembly block, making assembly conditional on an expression value being greater than or equal to zero.

```
.ifge expression statement-group
```

Parameters

`expression`

Any valid expression.

`statement-group`

Any valid assembly statements.

Remarks

If the `expression` value is greater than or equal to 0, the assembler processes the statements of the block. If the `expression` value is less than 0, the assembler skips the statements of the block.

.ifgt

Starts a conditional assembly block, making assembly conditional on an expression value being greater than zero.

```
.ifgt expression statement-group
```

Parameters

`expression`

Any valid expression.

`statement-group`

Any valid assembly statements.

Remarks

If the `expression` value is greater than 0, the assembler processes the statements of the block. If the `expression` value is less than or equal to 0, the assembler skips the statements of the block.

.ifle

Starts a conditional assembly block, making assembly conditional on an expression value being less than or equal to zero.

```
.ifle expression statement-group
```

Parameters

`expression`

Any valid expression.

`statement-group`

Any valid assembly statements.

Remarks

If the `expression` value is less than or equal to 0, the assembler processes the statements of the block. If the `expression` value is *greater* than 0, the assembler skips the statements of the block.

.iflt

Starts a conditional assembly block, making assembly conditional on an expression value being less than zero.

```
.iflt expression statement-group
```

Parameters

`expression`

Any valid expression.

`statement-group`

Any valid assembly statements.

Remarks

If the `expression` value is less than 0, the assembler processes the statements of the block. If the `expression` value equals or exceeds 0, the assembler skips the statements of the block.

.ifnc

Starts a conditional assembly block, making assembly conditional on the *inequality* of two strings.

```
.ifnc string1, string2 statement-group
```

Parameters

`string1`

Any valid string.

`string2`

Any valid string.

`statement-group`

Any valid assembly statements.

Remarks

If `string1` and `string2` are *not* equal, the assembler processes the statements of the block. (The inequality comparison is case-sensitive.) If the strings *are* equal, the assembler skips the statements of the block.

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

.ifndef

Starts a conditional assembly block, making assembly conditional on a symbol *not* being defined.

```
.ifndef symbol statement-group
```

Parameters

`symbol`

Any valid symbol.

`statement-group`

Any valid assembly statements.

Remarks

If previous code does *not* include a definition for `symbol`, the assembler processes the statements of the block. If there *is* a definition for `symbol`, the assembler skips the statements of the block.

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

.ifne

Starts a conditional assembly block, making assembly conditional on an expression value *not* being equal to zero.

```
.ifne expression statement-group
```

Parameters

`expression`

Any valid expression.

`Statement-group`

Any valid assembly statements.

Remarks

If the `expression` value is *not* equal to 0, the assembler processes the statements of the block. If the `expression` value *does* equal 0, the assembler skips the statements of the block.

.include

Tells the assembler to take input from the specified file.

```
.include filename
```

Parameter

`filename`

Name of an input file.

Remarks

When the assembler reaches the end of the specified file, it takes input from the assembly statement line that follows the `.include` directive. The specified file can itself contain an `.include` directive that specifies yet another input file.

.line

Specifies the absolute line number (of the current source file) for which the assembler generates subsequent code or data.

```
.line number
```

Parameter

`number`

Line number of the file; the file's first line is number 1.

Remarks

This directive is appropriate if you must explicitly provide a line number to the assembler *as debugging information*. But this directive turns *off* automatic generation of debugging information (which the assembler does if you enable the debugger). Providing Debugging Information explains additional information about debugging.

.long

Declares an initialized block of 32-bit short integers.

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

Parameters

`label`

Name of the block of integers.

`expression`

Value for 32 bits of the block; must fit into 32 bits.

.macro

Starts the definition of a macro.

```
label .macro [ parameter ] [ ,parameter ] ...
```

Parameters

`label`

Name you give the macro.

`parameter`

Optional parameter for the macro.

.mexit

Stops macro execution before it reaches the `.endm` directive. Program execution continues with the statement that follows the macro call.

```
.mexit
```

.offset

Starts a record definition, which extends to the start of the next section.

```
.offset [expression]
```

Parameter

`expression`

Optional initial location-counter value.

Remarks

Table 3.5 lists the only directives you can use inside a record.

Table 3.5 Directives Allowed in a Record

.align	.double	.org	.textequ
.ascii	.equ	.set	
.asciz	.float	.short	
.byte	.long	.space	

Data declaration directives such as `.byte` and `.short` update the location counter, but do not allocate any storage.

Example

Listing 3.5 shows a sample record definition.

Listing 3.5 Record Definition with Offset Directive

```
            .offset
top:        .short   0
left:       .short   0
```

```
bottom:     .short    0
right:      .short    0
rectSize    .equ      *
```

.option

Sets an assembler control option as Table 3.6 describes.

```
.option keyword setting
```

Parameters

`keyword`

Control option.

`setting`

Setting value appropriate for the option: OFF, ON, RESET, or a particular number value. RESET returns the option to its previous setting.

Table 3.6 Option Keywords

Keyword	Description
alignment off | on | reset	Controls data alignment on a natural boundary. Does not correspond to any option of the Assembler settings panel.
branchsize 8 | 16 | 32	Specifies the size of forward branch displacement. Applies only to x86 and 68K assemblers. Does not correspond to any option of the Assembler settings panel.
case off | on | reset	Specifies case sensitivity for identifiers. Corresponds to the **Case-sensitive identifiers** checkbox of the Assembler settings panel.
colon off | on | reset	Specifies whether labels must end with a colon (:). The OFF setting means that you can omit the ending colon from label names that start in the first column. Corresponds to the **Labels must end with ':'** checkbox of the Assembler settings panel.

Table 3.6 Option Keywords (*continued*)

Keyword	Description
no_at_macros off | on	Controls $AT use in macros. The OFF setting means that the assembler issues a warning if a macro uses $AT. Applies only to the MIPS Assembler.
no_section_resume on | off | reset	Specifies whether section directives such as `.text` resume the last such section or creates a new section.
period off | on | reset	Controls period usage for directives. The ON setting means that each directive must start with a period. Corresponds to the **Directives begin with '.'** checkbox of the Assembler settings panel.
processor procname | reset	Specifies the target processors for the assembly code; tells the assembler to confirm that all instructions are valid for those processors. Separate names of multiple processors with vertical bars (|).
reorder off | on | reset	Controls NOP instructions after jumps and branches. The ON setting means that the assembler inserts a NOP instruction, possibly preventing pipeline problems. The OFF setting means that the assembler does not insert a NOP instruction, so that you can specify a different instruction after jumps and branches. Applies only to the MIPS Assembler.
space off | on | reset	Controls spaces in operand fields. The OFF setting means that a space in an operand field starts a comment. Corresponds to the **Allow space in operand field** checkbox of the Assembler settings panel.

.org

Changes the location-counter value, relative to the base of the current section.

```
.org expression
```

Parameter

`expression`

New value for the location counter; must be greater than the current location-counter value.

Remarks

Addresses of subsequent assembly statements begin at the new expression value for the location counter, but *this value is relative to the base of the current section*.

Example

In Listing 3.6, the label `Alpha` reflects the value of `.text + 0x1000`. If the linker places the `.text` section at `0x10000000`, the runtime `Alpha` value is `0x10001000`.

Listing 3.6 Address-Change Example

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

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

.pragma

Tells the assembler to use a particular pragma setting as it assembles code.

```
.pragma pragma-type setting
```

Parameters

`pragma-type`

Type of pragma.

`setting`

Setting value.

.previous

Reverts to the previous section; toggles between the current section and the previous section.

```
.previous
```

.public

Declares specified labels to be public.

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

Parameter

`label`

Any valid label.

Remarks

If the labels already are defined in the same file, the assembler exports them (makes them available to other files). If the labels are *not* already defined, the assembler imports them (finds their definitions in another file).

.rodata

Specifies an initialized read-only data section.

```
.rodata
```

.sbss

Specifies a small data section as uninitialized and read-write. (Some architectures do not support this directive.)

```
.sbss
```

.sbss2

Specifies a small data section as uninitialized and read-write. (Some architectures do not support this directive.)

```
.sbss2
```

.sdata

Specifies a small data section as initialized and read-write. (Some architectures do not support this directive.)

```
.sdata
```

.sdata0

Specifies a small data section as read/write. (Some architectures do not support this directive.)

```
.sdata2
```

.sdata2

Specifies a small data section as initialized and read-only. (Some architectures do not support this directive.)

```
.sdata2
```

.section

Defines a section of an object file.

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

Parameters

`name`

Name of the section.

`alignment`

Alignment boundary.

`type`

Numeric value for the ELF section type, per Table 3.7. The default `type` value is `1`: (SHT_PROGBITS).

`flags`

Numeric value for the ELF section flags, per Table 3.8. The default `flags` value is `0x00000002, 0x00000001`: (SHF_ALLOC+SHF_WRITE).

Table 3.7 ELF Section Header Types (SHT)

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

Table 3.7 ELF Section Header Types (SHT) (*continued*)

Type	Name	Meaning
10	SHLIB	Section has unspecified semantics, so does not conform to the Application Binary Interface (ABI) standard.
11	DYNSYM	Section contains a minimal set of symbols for dynamic linking.

Table 3.8 ELF Section Header Flags (SHF)

Flag	Name	Meaning
0x00000001	WRITE	Section contains data that is writable during execution.
0x00000002	ALLOC	Section occupies memory during execution.
0x00000004	EXECINSTR	Section contains executable machine instructions.
0xF0000000	MASKPROC	Bits this mask specifies are reserved for processor-specific purposes.

Remarks

You can use this directive to create arbitrary relocatable sections, including sections to be loaded at an absolute address.

Most assemblers generate ELF (Executable and Linkable Format) object files, but a few assemblers generate COFF (Common Object File Format) object files.

The assembler supports this alternative syntax, which you may find convenient:

```
.section    name,typestring
```

(The `name` parameter has the same role as in the full syntax. The `typestring` value can be `text`, `data`, `rodata`, `bss`, `sdata`, or so forth.)

Normally, repeating a `.text` directive would resume the previous `.text` section. But to have each `.text` directive create a separate section, include in this relocatable section the statement `.option no_section_resume_on`.

Example

This example specifies a section named `vector`, with an alignment of 4 bytes, and default type and flag values:

```
.section vector,4
```

.set

Defines an equate, assigning an initial value. You can change this value at a later time.

```
equate .set expression
```

Parameters

`equate`

Name of the equate.

`expression`

Temporary initial value for the equate.

.short

Declares an initialized block of 16-bit short integers.

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

Parameters

`label`

Name of the block of integers.

`expression`

Value for 16 bits of the block; must fit into 16 bits.

.size

Specifies a length for a symbol.

```
.size symbol, expression
```

Parameters

`symbol`

Symbol name.

`expression`

Number of bytes.

Remarks

This directive is appropriate if you must explicitly provide a symbol size to the assembler *as debugging information*. Providing Debugging Information explains additional information about debugging.

.space

Declares a block of bytes, initializing each byte to zero or to a specified fill value.

```
[label] .space expression [, fill_value]
```

Parameters

`label`

Name of the block of bytes.

`expression`

Number of bytes in the block.

`fill_value`

Initialization value for each bytes in the block; the default value is zero.

.text

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

```
.text
```

Remarks

Normally, repeating a `.text` directive would resume the previous `.text` section. But to have each `.text` directive create a separate section, include the statement `.option no_section_resume_on` in a relocatable section. (Use the `.section` directive to create such a section.)

.textequ

Defines a text equate, assigning a string value.

```
equate .textequ "string"
```

Parameters

`equate`

Name of the equate.

`string`

String value for the equate, in double quotes.

Remarks

This directive helps port existing code. You can use it to give new names to machine instructions, directives, and operands.

Upon finding a text equate, the assembler replaces it with the string value before performing any other processing on that source line.

Examples

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

.type

Specifies the type of a symbol.

```
.type symbol, @function | @object
```

Parameters

`symbol`

Symbol name.

`@function`

Function type specifier.

`@object`

Variable specifier

Remarks

This directive is appropriate if you must explicitly provide a type to the assembler *as debugging information*. Providing Debugging Information explains additional information about debugging.

Providing Debugging Information

Perhaps the most common way to provide project debugging information to the assembler is to let the assembler itself automatically generate the information. This level of debugging information means that the debugger source window can display the assembly source file. It also means that you can step through the assembly code and set breakpoints.

For this *automatic* generation of debugging information, important points are:

1. Avoid directives `.debug` and `.line`; using either directive turns *off* automatic generation.
2. For some implementations, the linker requires instructions to be in the `.text` section, in order for automatic generation to happen.
3. In automatic-debug mode, the assembler puts everything into a single function (the assembler does not know how source code may be divided into functions). Accordingly, you may see names such as `@DummyFn1` in the debugger stack window. But if you wish, you can use the `.function` directive to divide the code into sections.
4. When you debug the assembly-language code, the code may seem *spaghetti-like* and it may not create valid call frames on the stack. This is normal for the assembler. Because of this, however, the debugger cannot provide stack-crawl information.

An alternative method is providing debugging information to the assembler explicitly, via the debugging directives `.file`, `.function`, `.line`, `.size`, and `.type`. This would be particularly appropriate if you were developing a new compiler that output assembly source code: these directives would relate the assembler code back to the original source-code input to the new compiler. But you must avoid the `.debug` directive, which tells the assembler to ignore the debugging directives.

A final method of providing debugging information, rare in normal use, is using the `.debug` directive to create an explicit debug section. Such a section might begin:

```
.debug
.long  1
.asciz  "MyDebugInfo"
```

But remember that the `.debug` directive deactivates any of the debugging directives.

4

Using Macros

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

This chapter includes these topics:

Defining Macros

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 of the macro

To define a macro, use the `.macro` directive.

NOTE If you use a local label in a macro, the scope of the label is limited to the expansion of the macro. (Local labels begin with the @ character.)

The `.macro` directive is part of the first line of a macro definition. Every macro definition ends with the `.endm` directive .Listing 4.1 shows the full syntax, and Table 4.1 explains the syntax elements.

Listing 4.1 Macro Definition Syntax: .macro Directive

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

Table 4.1 Syntax Elements: .macro Directive

Element	Description
`name`	Label that invokes the macro.
`parameter`	Operand the assembler passes to the macro for us in the macro body.
`macro_body`	One or more assembly language statements. Invoking the macro tell the assembler to substitutes these statements.

The body of a simple macro consists of just one or two statements for the assembler to execute. Then, in response to the `.endm` directive, the assembler resumes program execution at the statement immediately after the macro call.

But not all macros are so simple. For example, a macro can contain a conditional assembly block, The conditional test could lead to the `.mexit` directive stopping execution early, before it reaches the `.endm` directive.

Listing 4.2 is the definition of macro `addto`, which includes an `.mexit` directive. Listing 4.3 shows the assembly-language code that calls the `addto` macro. Listing 4.4 shows the expanded `addto` macro calls.

Listing 4.2 Conditional Macro Definition

```
//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 Assembly Code that Calls 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 Expanded addto Macro Calls

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

Using Macro Arguments

You can refer to parameters directly by name. Listing 4.5 shows the `setup` macro, which moves an integer into a register and branches to the label `_final_setup`. Listing 4.6 shows a way to invoke the `setup` macro., and Listing 4.7 shows how the assembler expands the `setup` macro.

Listing 4.5 Setup Macro Definition

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

Listing 4.6 Calling Setup Macro

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

Listing 4.7 Expanding Setup Macro

```
           move    eax, VECT
```

```
            call    _final_setup
```

If 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 4.8 shows the `smallnum` macro, which creates a small float by appending the string `E-20` to the macro argument. Listing 4.9 shows a way to invoke the `smallnum` macro, and Listing 4.10 shows how the assembler expands the `smallnum` macro.

Listing 4.8 Smallnum Macro Definition

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

Listing 4.9 Invoking Smallnum Macro

```
smallnum  10
```

Listing 4.10 Expanding Smallnum Macro

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

Macro syntax includes positional parameter references (this feature can provide compatibility with other assemblers). For example, Listing 4.11 shows a macro with positional references `\1` and `\2`. Listing 4.12 shows an invocation of this macro, with parameter values `10` and `print`. Listing 4.13 shows the macro expansion.

Listing 4.11 Doit Macro Definition

```
doit:     .macro
          mov     eax,\1
          call    \2
          .endm
```

Listing 4.12 Invoking Doit Macro

```
doit    10,print
```

Listing 4.13 Expanding Doit Macro

```
    move    eax,10
```

```
call   print
```

Macro Repeat Directives

The assembler macro language includes the repeat directives `.rept`, `.irp`, and `.irpc`, along with the `.endr` directive, which must end any of the other three.

.rept

Repeats the statements of the block the specified number of times; the `.endr` directive must follow the statements.

```
.rept  expression
statement-group
.endr
```

Parameters

`expression`

Any valid expression that evaluates to a positive integer.

`statement-group`

Any statements valid in assembly macros.

.irp

Repeats the statements of the block, each time substituting the next parameter value. The `.endr` directive must follow the statements.

```
.irp  name exp1[,exp2[,exp3]...]
statement-group
.endr
```

Parameters

`name`

Placeholder name for expression parameter values.

`exp1, exp2, exp3`

Expression parameter values; the number of these expressions determines the number of repetitions of the block statements.

`statement-group`

Any statements valid in assembly macros.

Example

Listing 4.14 specifies three repetitions of `.byte`, with successive `name` values 1, 2, and 3. Listing 4.15 shows this expansion.

Listing 4.14 .irp Directive Example

```
.irp    databyte  1,2,3
.byte   databyte
.endr
```

Listing 4.15 .irp Example Expansion

```
.byte  1
.byte  2
.byte  3
```

.irpc

Repeats the statements of the block as many times as there are characters in the string parameter value. For each repetition, the next character of the string replaces the name parameter.

`.irpc  name,string`

`statement-group`

`.endr`

Parameters

`name`

Placeholder name for string characters.

`string`

Any valid character string.

`statement-group`

Any statements valid in assembly macros.

Creating Unique Labels and Equates

Use the backslash and at characters (\@) to have the assembler generate unique labels and equates within a macro. Each time you invoke the macro, the assembler generates a unique symbol of the form `??nnnn`, such as `??0001` or `??0002`.

In your code, you refer to such unique labels and equates just as you do for regular labels and equates. But each time you invoke the macro, the assembler replaces the \@ sequence with a unique numeric string and increments the string value.

Listing 4.16 shows a macro that uses unique labels and equates. Listing 4.17 shows two calls to the `my_macro` macro, with `my_count` initialized to 0. Listing 4.18 shows the expanded `my_macro` code after the two calls.

Listing 4.16 Unique Label Macro Definition

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

Listing 4.17 Invoking my_macro Macro

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

Listing 4.18 Expanding my_macro Calls

```
alpha??0000 =     my_count
my_count    .set  my_count + 1
            add   ebx, alpha??0000
            jmp   label??0000
            add   eax, ebx
label??0000
            nop
alpha??0001 =     my_count
```

```
my_count    .set  my_count + 1
            add   ebx, alpha??0001
            jmp   label??0001
            add   eax, ebx
label??0001
            nop
```

Number of Arguments

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

Invoking Macros

To invoke a macro, use its name in your assembler listing, separating parameters with commas. To pass a parameter that includes a comma, enclose the parameter in angle brackets.

For example, Listing 4.19 shows macro `pattern`, which repeats a pattern of bytes passed to it the number of times specified in the macro call. Listing 4.20 shows a statement that calls `pattern`, passing a parameter that includes a comma. Listing 4.21 is another example calling statement; the assembler generates the same code in response to the calling statement of either Listing 4.20 or Listing 4.21.

Listing 4.19 Pattern Macro Definition

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

Listing 4.20 Macro Argument with Commas

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

Listing 4.21 Alternate Byte-Pattern Method

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

5

Common Assembler Settings

The Assembler target settings panel includes settings common to all the assemblers. This chapter explains these settings.

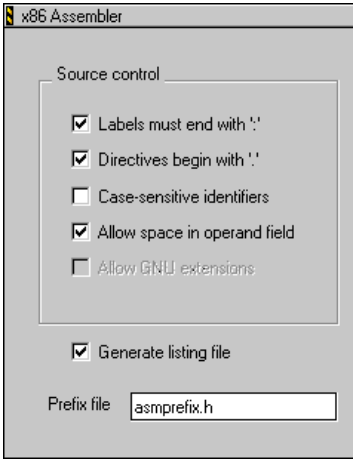
Displaying Assembler Target Settings Panel

To modify the settings for an assembler:

1. From the main menu bar, select **Edit > Project Settings**. A dialog box appears.
2. Select the name of the assembler. Its settings panel appears.

Figure 5.1 shows the settings common to all the assemblers. For information on other settings that pertain to your assembler, see the processor-specific chapter of this manual.

Figure 5.1 Common Assembler Settings



Common Assembler Settings Descriptions

Table 5.1 explains the common assembler settings.

Table 5.1 Common Assembler Settings

Element	Purpose	Comments
Labels must end with : checkbox	Clear — Symbols that start in column 1 or end with colons are labels. Checked — All labels must end with colons, but can start in any column.	Default: Checked. Corresponds to the `colon` keyword of the `.option` directive. Clearing this checkbox makes sense if you import existing code that has symbols without colons.
Directives begin with . checkbox	Clear — You may omit starting period from directives. Checked — Directives must start with periods.	Default: Checked. Corresponds to the `period` keyword of the `.option` directive.
Case-sensitive identifiers checkbox	Clear — Symbols are *not* case sensitive. Checked — Symbols *are* case sensitive.	Default: Checked. Corresponds to the `case` keyword of the `.option` directive. Instructions, directives, and macro names *never* are case sensitive.
Allow space in operand field checkbox	Clear — Space character in an operand field starts a comment. Checked — Spaces are allowed in operand fields.	Default: Checked. Corresponds to the `space` parameter of the `.option` directive.
Generate listing file checkbox	Clear — Assembler does not create a listing file. Checked — Assembler creates a listing file. The filename is that of the source file, but with extension `.list`.	Default: Clear. A listing file is a text file useful for comparing source and machine code.
Prefix file text box	Tells the assembler to process the specified file before processing each file of your project.	Default: None. Specifying a file is like putting the same `.include` directive at the start of each assembly file.

6

ColdFire-Specific Information

Almost all the information of earlier chapters pertains to ColdFire® target processors. The few differences are:

1. **Comments** — Assembly Language Syntax explains these common ways to specify comments:
   - Characters `//`, starting in any column.
   - Characters `/*` ... `*/`, starting in any column.
   - An asterisk (`*`), starting in the first column of the line.
   - A space in an operand field, provided that you clear the **Allow space in operand field** checkbox of the Assembler settings panel.

   A ColdFire target processor gives you these additional ways to specify comments:
   - In GNU mode: starting the comment with a vertical stroke (`|`) character.
   - Not in GNU mode: starting the comment with a semicolon (`;`).

   Such comments may begin in any column of a line.
2. **Hexadecimal Notation** — For ColdFire processors, the preferred hexadecimal notation is `$`, as in `$deadbeef`. This contrasts with Chapter 2, which explains that the preferred notation for most processors is `0x`.
3. **Sections** — As Using Directives explains, not all target architectures support the small-data assembler directives `.sbss`, `.sbss2`, `.sdat`, `.sdata0`, or `.sdata2`. For the ColdFire architecture, the linker can be more restrictive than the assembler. You may need to experiment to find out which of these directives are supported by both your assembler and linker.

   As with most assemblers, the ColdFire assembler generates ELF, not COFF, object files.
4. **Automatic Debugging** — For automatic generation of debugging information, your linker may require that instructions be in the `.text` section.

Index

Symbols

A

C

U