

CodeWarrior Development Studio for Microcontrollers V10.x Kinetis Assembler Reference Manual

Document Number: CWMCUKINASMREF
Rev 10.6, 02/2014

Contents

Section number	Title	Page
Chapter 1		
Introduction		
1.1	Release Notes.....	9
1.2	In This Book.....	9
1.3	Where to Learn More.....	10
1.4	Accompanying Documentation.....	10
Chapter 2		
Assembly Language Syntax		
2.1	Assembly Language Statements.....	13
2.2	Statement Syntax.....	14
2.3	Symbols.....	14
2.3.1	Labels.....	15
2.3.1.1	Non-Local Labels.....	16
2.3.1.2	Local Labels.....	16
2.3.1.3	Referencing Labels.....	17
2.3.1.4	Relocatable Labels.....	18
2.3.2	Equates.....	19
2.3.3	Case-Sensitive Identifiers.....	20
2.4	Constants.....	21
2.4.1	Integer Constants.....	21
2.4.2	Floating-Point Constants.....	21
2.4.3	Character Constants.....	22
2.5	Expressions.....	22
2.6	Comments.....	24
2.7	Data Alignment.....	25
2.8	GNU Compatibility.....	25
2.8.1	GNU Compatible Syntax Option.....	25
2.8.2	Supported Extensions.....	26

Section number	Title	Page
2.8.3	Unsupported Extensions.....	28
Chapter 3		
Using Directives		
3.1	Preprocessor Directives.....	32
3.1.1	#define.....	32
3.1.2	#elif.....	33
3.1.3	#else.....	34
3.1.4	#endif.....	35
3.1.5	#error.....	35
3.1.6	#if	35
3.1.7	#ifdef	36
3.1.8	#ifndef.....	37
3.1.9	#include.....	37
3.1.10	#line.....	38
3.1.11	#pragma.....	38
3.1.12	#undefine.....	38
3.2	Native Assembler Directives.....	39
3.2.1	.align.....	41
3.2.2	.ascii.....	42
3.2.3	.asciz.....	42
3.2.4	.bss.....	43
3.2.5	.byte.....	43
3.2.6	.data.....	44
3.2.7	.debug.....	44
3.2.8	.double.....	44
3.2.9	.else.....	45
3.2.10	.elseif.....	45
3.2.11	.endian.....	46
3.2.12	.endif.....	47

Section number	Title	Page
3.2.13	<code>.endm</code>	47
3.2.14	<code>.equ</code>	47
3.2.15	equal sign (=).....	48
3.2.16	<code>.error</code>	48
3.2.17	<code>.extern</code>	48
3.2.18	<code>.file</code>	49
3.2.19	<code>.float</code>	50
3.2.20	<code>.function</code>	50
3.2.21	<code>.global</code>	51
3.2.22	<code>.if</code>	51
3.2.23	<code>.ifc</code>	52
3.2.24	<code>.ifdef</code>	53
3.2.25	<code>.ifeq</code>	53
3.2.26	<code>.ifge</code>	54
3.2.27	<code>.ifgt</code>	54
3.2.28	<code>.ifle</code>	55
3.2.29	<code>.iflt</code>	55
3.2.30	<code>.ifnc</code>	56
3.2.31	<code>.ifndef</code>	56
3.2.32	<code>.ifne</code>	57
3.2.33	<code>.include</code>	58
3.2.34	<code>.line</code>	58
3.2.35	<code>.long</code>	59
3.2.36	<code>.macro</code>	59
3.2.37	<code>.mexit</code>	59
3.2.38	<code>.offset</code>	60
3.2.39	<code>.option</code>	60
3.2.40	<code>.org</code>	61
3.2.41	<code>.pragma</code>	62

Section number	Title	Page
3.2.42	.previous.....	63
3.2.43	.public.....	63
3.2.44	.rodata.....	63
3.2.45	.sbss.....	64
3.2.46	.sbss2.....	64
3.2.47	.sdata.....	64
3.2.48	.sdata0.....	64
3.2.49	.sdata2.....	65
3.2.50	.section.....	65
3.2.51	.set.....	67
3.2.52	.short.....	67
3.2.53	.size.....	68
3.2.54	.space.....	68
3.2.55	.text.....	68
3.2.56	.textequ.....	69
3.2.57	.type.....	69
3.3	Debugging Directives.....	70
3.3.1	.file.....	71
3.3.2	.function.....	71
3.3.3	.line.....	72
3.3.4	.size.....	73
3.3.5	.type.....	73

Chapter 4 Using Macros

4.1	Defining Macros.....	75
4.1.1	Using Macro Arguments.....	77
4.1.2	Macro Repeat Directives.....	78
4.1.2.1	.rept.....	78
4.1.2.2	.irp.....	79

Section number	Title	Page
4.1.2.3	.irpc.....	80
4.1.3	Creating Unique Labels and Equates.....	80
4.1.4	Number of Arguments.....	81
4.2	Invoking Macros.....	81

Chapter 5 Kinetic Assembler General Settings

5.1	Displaying Kinetic Assembler General Settings.....	83
-----	--	----

Chapter 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. In this chapter:

- [Release Notes](#)
- [In This Book](#)
- [Where to Learn More](#)
- [Accompanying Documentation](#)

1.1 Release Notes

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

```
<CWInstallDir>\MCU\Release_Notes
```

`CWInstallDir` is the directory where the CodeWarrior software is installed.

1.2 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 about the inline assembler of the CodeWarrior C/C++ compiler, refer to the *Targeting Manual* for your target processor or the *Target Assembler/Compiler Manual*.

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. The following table 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	Describes an overview about 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.
Kinetis Assembler General 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.

1.3 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.

1.4 Accompanying Documentation

The **Documentation** page describes the documentation included in the *CodeWarrior Development Studio for Microcontrollers v10.x*. You can access the **Documentation** by:

- opening the `START_HERE.html` in `<CWInstallDir>\MCU\Help` folder,
- selecting **Help > Documentation** from the IDE's menu bar, or selecting the **Start > Programs > Freescale CodeWarrior > CW for MCU v10.x > Documentation** from the Windows taskbar.

NOTE

To view the online help for the CodeWarrior tools, first select **Help > Help Contents** from the IDE's menu bar. Next, select required manual from the **Contents** list. For general information about the CodeWarrior IDE and debugger, refer to the *CodeWarrior Common Features Guide* in this folder: <CWInstallDir>\MCU\Help\PDF



Chapter 2

Assembly Language Syntax

This chapter explains the syntax of the assembly language statements. In this chapter:

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

2.1 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.

2.2 Statement Syntax

The following listing shows the syntax of an assembly language statement. The table after the listing describes the elements of this syntax.

Listing: 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.

2.3 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 the symbols.

The syntax of a symbol is:

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

In general, the 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. In this topic:

- [Labels](#)
- [Equates](#)
- [Case-Sensitive Identifiers](#)

2.3.1 Labels

A *label* is a symbol that represents an address. The scope of a label 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, refer to the [Kinetis Assembler General Settings](#) chapter.

In this topic:

- [Non-Local Labels](#)
- [Local Labels](#)
- [Referencing Labels](#)
- [Relocatable Labels](#)

2.3.1.1 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 (\$).

2.3.1.2 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 a 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.

The following list shows the scope of a 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: Local Label Scope in a Macro

```
MAKEPOS      .MACRO
    cmp #1,  d0
    bne      @SKIP
    neg      d0
@SKIP:      ;Scope of this label is within the macro
    .ENDM
START:
    move COUNT,  d0
    cmp #1,      d0
    bne @SKIP
    MAKEPOS
@SKIP:      ;Scope of this label is START to END
            ;excluding lines arising from
            ;macro expansion
    addq #1,  d0
END: rts
```

2.3.1.3 Referencing Labels

You can give reference to the contents of a label and the address of a label in *ARM Assemblers*.

The equal sign (=) distinguishes between the address of a label reference and the contents of a label reference.

```
ldr r1, label // Returns the contents of the label.
```

```
ldr r1, =label // Returns the actual address of the label.
```

The following listing shows the code using `label` and `=label`.

Listing: Referencing Labels Using `label` and `=label`

```
main.c
extern void asmfunc(void;
```

Symbols

```

int main()
{
    asmfunc();
}

asmfunc.asm
    .public asmfunc
    .public __SP_INIT
SVC_Stack_Size .equ 0x100
    .text
asmfunc:
    ldr r1,=@SVC_Stack
    ldr r0,=@USR_Stack
    ldr r1,@SVC_Stack
    ldr r0,@USR_Stack
    ldr r2,[r1]
    ldr r3,[r0]
    mov pc,lr
@SVC_Stack
    .long __SP_INIT
@USR_Stack
    .long __SP_INIT-SVC_Stack_Size

```

2.3.1.4 Relocatable Labels

The assembler assumes a flat 32-bit memory space. The following table lists the expressions 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. Also, it is the PC-relative target of a branch or call. It is a 32-bit address.

Table continues on the next page...

Table 2-2. Relocatable Label Expressions (continued)

Expression	Represents
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 <i>label@l</i> 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 <i>label@l</i> 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.

2.3.2 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.

NOTE

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 Kinetis processor, you can specify a default assumption of 8, 16, or 32 bits.

The following listing is a valid forward equate.

Listing: 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:
```

The following listing is an invalid forward equate. The assembler cannot immediately resolve the expression in the `.space` directive, so the effect on the location counter is unknown.

Listing: Invalid Forward Equate

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

2.3.3 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.)

2.4 Constants

The assembler recognizes three kinds of constants:

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

2.4.1 Integer Constants

The following table 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 string 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 an integer constants.

2.4.2 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.

2.4.3 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 the following table lists.

Table 2-4. Character Constant Escape Sequences

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

2.5 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 (*).

The following table 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	
Binary	<	less than
	>	greater than
Unary	+	unary plus
	-	unary minus
	~	unary bitwise complement
Alternate	<>	not equal to

The operator precedence is:

Comments

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.

2.6 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 Kinetis example is:

```
mov r0,r1      ;This is a comment
```

A PowerPC example is;

```
mov 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.

2.7 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.

2.8 GNU Compatibility

The CodeWarrior Assembler supports several GNU-format assembly language extensions. In this topic:

- [GNU Compatible Syntax Option](#)
- [Supported Extensions](#)
- [Unsupported Extensions](#)

2.8.1 GNU Compatible Syntax Option

You can enable GNU compatible assembler syntax either through the **ARM Assembler** target settings panel in the CodeWarrior IDE, or by specifying `-gnu` on the `mwarmasm.exe` command line.

The GNU Compatible Syntax options are effective only when GNU assembler format conflicts with the CodeWarrior assembler format. The following table lists the specific cases.

Table 2-6. GNU Compatible Syntax Option

Case	Comment
Defining Equates	Whether defined using <code>.equ</code> or <code>.set</code> , all equates can be re-defined.
Ignored directives	The <code>.type</code> directive is ignored.
Undefined Symbols	Undefined symbols are automatically treated as imported
Arithmetic Operators	<ul style="list-style-type: none"> • <code>< and ></code> mean left-shift and right-shift instead of less than and greater than. • <code>< ! ></code> means bitwise-or-not instead of logical not.
Precedence Rules	Precedence rules for operators are changed to be compatible with GNU rather than with C.
Local Labels	Local labels with multi-number characters are supported (example: "1000:"). There is no limit on the number of digits in the label name. Multiple instances of the label are allowed. When referenced, you get the nearest one - forwards or backwards depending on whether you append 'f' or 'b' to the number.
Numeric Constants	Numeric constants beginning with 0 are treated as octal.
Semicolon Use	Semicolons can be used as a statement separator.
Unbalanced Quotes	A single unbalanced quote can be used for character constants. For example, <code>.byte 'a</code>

2.8.2 Supported Extensions

The supported GNU extensions, regardless whether you enable GNU assembler compatible syntax are as follows:

- Lines beginning with `#`, `*` or `;` are always treated as comment, even if the comment symbol for that assembler is something different.
- Escape characters in strings extended to include `\xNN` for hex digits and `\NNN` for octal.
- Binary constants can begin with `ob`.
- The supported GNU macro language, with macros defined by:

```
.macro name, arg1 [=default1], arg2...s1
...
.endm
```

Arguments can have default values as shown, and when called can be specified by value or position. (refer to the GNU documentation for details.)

- New or enhanced directives (refer to the GNU documentation for details.)

The following table lists the supported GNU assembler directives.

Table 2-7. Supported GNU Assembler Directives

Directive	Description	Comment
<code>.abort</code>	End assembly	Supported
<code>.align N, [pad]</code>	Align	Now accepts optional padding byte
<code>.app-file name</code>	Source name	Synonym for <code>.file</code>
<code>.balign[w1] N, [pad]</code>	Align	Align to N (with optional padding value)
<code>.common name, length</code>	Common data	Reserve space in BSS for global symbol
<code>.def</code>	Debugging	Accepted but ignored
<code>.desc</code>	Debugging	Accepted but ignored
<code>.dim</code>	Debugging	Accepted but ignored
<code>.eject</code>	Eject page	Accepted but ignored
<code>.endr</code>	End repeat	refer <code>.irp, .irpc</code>
<code>.endif</code>	Debugging	Accepted but ignored
<code>.fill N, [size], [val]</code>	Repeat data	Emit N copies of width `size', value `val'
<code>.hword val..</code>	Half-word	Synonym for <code>.short</code>
<code>.ident</code>	Tags	Accepted but ignored
<code>.ifndef name</code>	Conditional	Synonym for <code>.ifndef</code>
<code>.include name</code>	Include file	Now accepts single, double or no quotes
<code>.init val..</code>	Word	Synonym for <code>.long</code>
<code>.irp name, values</code>	Repeat	Repeat up to <code>.endr</code> substituting values for name
<code>.irpc name, chars</code>	Repeat	Repeat up to <code>.endr</code> substituting chars for name
<code>.lcomm name, length</code>	Local common	Reserve length bytes in bss
<code>.Iflags</code>	Ignored	Accepted but ignored
<code>.ln lineno</code>	Line number	Synonym for <code>.line</code>
<code>.list</code>	Listing on	Switch on listing
<code>.local name</code>	Local Macro var	Declare name as local to macro
<code>.macro name, args..</code>	Macros	Supports GNU syntax, default values, etc
<code>.nolist</code>	Listing off	Disable listing
<code>.org pos, fill</code>	Origin	Now allows fill value to be specified
<code>.p2align[w1] N[, pad]</code>	Align	Align to $2^{*}N$, using pad value `pad'
<code>.psize</code>	Page size	Accepted but ignored
<code>.rept N</code>	Repeat	Repeat block up to <code>.endr</code> N times
<code>.sbttl</code>	Subtitle	Accepted but ignored

Table continues on the next page...

Table 2-7. Supported GNU Assembler Directives (continued)

Directive	Description	Comment
<code>.scl</code>	Debugging	Accepted but ignored
<code>.size name,N</code>	Set size	Set size of name to N
<code>.skip N[,pad]</code>	Space	Skip N bytes, pad with `pad`
<code>.space N[,pad]</code>	Space	Skip N bytes, pad with `pad`
<code>.stabd</code>	Debugging	Accepted but ignored
<code>.stabn</code>	Debugging	Accepted but ignored
<code>.stabs</code>	Debugging	Accepted but ignored
<code>.str "string"</code>	Constant string	Synonym for <code>.asciz</code>
<code>.string "string"</code>	Constant String	Synonym for <code>.asciz</code>
<code>.tag</code>	Debugging	Accepted but ignored
<code>.title</code>	Title	Accepted but ignored
<code>.type</code>	Debugging	Ignored in GNU mode
<code>.val</code>	Debugging	Accepted but ignored
<code>.word</code>	Word	Synonym for <code>.long</code>

2.8.3 Unsupported Extensions

The CodeWarrior Assembler does not support all GNU extensions. The following table lists the unsupported GNU extensions.

Table 2-8. Unsupported GNU Extensions

Extension	Comment
Sub-sections (such as <code>.text 2</code>)	The sub-section number is ignored. As a workaround, you can create your own sections with the <code>.section <name></code> directive. You may have an arbitrary number of text subsections with the names <code>.text1</code> , <code>.text2</code> , etc.
Assignment to location counter (such as <code>. = .+4</code>)	As a workaround, you can advance the location counter with <code>.space <expr></code>
Empty expressions defaulting to 0. For example, <code>.byte ,</code> equivalent to <code>(.byte 0,0)</code>	There is no workaround for this. You must always supply the arguments.
<code>.linkonce</code> directive	The linker automatically detects logically-identical sections, and uses the following factors to determine whether to keep only one or both in the final image: <ul style="list-style-type: none"> the binding of the symbols associated with each section. the location of these two sections. For example, are the sections in the same overlay or overlay group? Is one in main, and the other in an overlay group?

Table continues on the next page...

Table 2-8. Unsupported GNU Extensions (continued)

Extension	Comment
.octa	16-byte numbers are not supported directly. As a workaround, you can use consecutive <code>.long</code> directives to build a large number in memory.
.quad	8-byte numbers are not supported directly. As a workaround, you can use consecutive <code>.long</code> directives to build a large number in memory.



Chapter 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 as in the following listing.

Listing: 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 or the native assembler syntax.

Listing: Preprocessor-Directive Example

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

Listing: Native-Assembler-Directive Example

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

In this chapter:

- [Preprocessor Directives](#)
- [Native Assembler Directives](#)
- [Debugging Directives](#)

3.1 Preprocessor Directives

The following listed are the preprocessor directives.

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.

3.1.1 #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 (;).

3.1.2 #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
```

Preprocessor Directives

```
#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.)

3.1.3 #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*.

3.1.4 #endif

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

```
.endif
```

3.1.5 #error

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

```
#error "message"
```

Parameter

message

Error message, in double quotes.

3.1.6 #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.

3.1.7 #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.

3.1.8 #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.

3.1.9 #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.

3.1.10 `#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.

3.1.11 `#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.

NOTE

This pragma is not supported for Kinetis processor.

3.1.12 #undefine

Removes the definition of a preprocessor macro.

```
#undefine name
```

Parameters

name

Name of the macro.

3.2 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.

The following listed are the native assembler directives.

Table 3-2. Assembler Directives

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

Table continues on the next page...

Table 3-2. Assembler Directives (continued)

Type	Directive	Description
Compatibility Conditional	<code>.ifeq</code>	Starts a string-equals-0 conditional assembly block.
	<code>.ifge</code>	Starts a string->=0 conditional assembly block.
	<code>.ifgt</code>	Starts a string->0 conditional assembly block.
Compatibility Conditional	<code>.ifle</code>	Starts a string-<=0 conditional assembly block.
	<code>.iflt</code>	Starts a string-<0 conditional assembly block.
	<code>.ifne</code>	Starts a string-not-equals-0 conditional assembly block.
Section Control	<code>.bss</code>	Specifies an uninitialized, read-only data section.
	<code>.data</code>	Specifies an initialized, read-write data section.
	<code>.debug</code>	Specifies a debug section.
	<code>.offset</code>	Starts a record definition.
	<code>.previous</code>	Reverts to the previous section.
	<code>.rodata</code>	Specifies an initialized, read-only data section.
	<code>.sbss</code>	Specifies an uninitialized, read-write small data section.
	<code>.sbss2</code>	Specifies an uninitialized, read-write small data section.
	<code>.sdata</code>	Specifies an initialized, read-write small data section.
	<code>.sdata0</code>	Specifies an initialized, read-write small data section.
	<code>.sdata2</code>	Specifies an initialized, read-only small data section.
	<code>.section</code>	Defines an ELF object-file section.
<code>.text</code>	Specifies an executable code section.	
Scope Control	<code>.extern</code>	Imports specified labels.
	<code>.global</code>	Exports specified labels.
	<code>.public</code>	Declares specified labels public.
Symbol Definition	<code>.equ</code>	Defines an equate; assigns a permanent value.
	equal sign (=)	Defines an equate; assigns an initial value.
	<code>.set</code>	Defines an equate.
	<code>.textequ</code>	Defines an equate; assigns a string value.
Data Declaration	<code>.ascii</code>	Declares a storage block for a string.

Table continues on the next page...

Table 3-2. Assembler Directives (continued)

Type	Directive	Description
	<code>.asciz</code>	Declares a 0-terminated storage block for a string.
	<code>.byte</code>	Declares an initialized block of bytes.
	<code>.double</code>	Declares an initialized block of 64-bit, floating-point numbers.
	<code>.float</code>	Declares an initialized block of 32-bit, floating-point numbers.
	<code>.long</code>	Declares an initialized block of 32-bit short integers.
	<code>.short</code>	Declares an initialized block of 16-bit short integers.
	<code>.space</code>	Declares a 0-initialized block of bytes.
Assembler Control	<code>.align</code>	Aligns location counter to specified power of 2.
	<code>.endian</code>	Specifies target-processor byte ordering.
	<code>.error</code>	Prints specified error message.
	<code>.include</code>	Takes input from specified file.
	<code>.option</code>	Sets an option.
	<code>.org</code>	Changes location-counter value.
	<code>.pragma</code>	Uses setting of specified pragma.
Debugging	<code>.file</code>	Specifies source-code file.
	<code>.function</code>	Generates debugging data.
	<code>.line</code>	Specifies absolute line number.
	<code>.size</code>	Specifies symbol length.
	<code>.type</code>	Specifies symbol type.

3.2.1 `.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.)

3.2.2 .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. The following table lists the escape sequences that this string can contain.

Table 3-3. Escape Sequences

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

3.2.3 .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. The following table lists the escape sequences that this string can contain.

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.

3.2.4 .bss

Specifies an uninitialized read-write data section.

```
.bss
```

3.2.5 .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.

3.2.6 .data

Specifies an initialized read-write data section.

.data

3.2.7 .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 [Debugging Directives](#) explains, using the `.debug` directive may be the least common method of providing debugging information to the assembler.

3.2.8 .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.

3.2.9 .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]elseendif` 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*.

3.2.10 .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 `.ifelseif ... [.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.)

3.2.11 .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.

3.2.12 .endif

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

```
.endif
```

3.2.13 .endm

Ends the definition of a macro.

```
.endm
```

3.2.14 .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.

3.2.15 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.

3.2.16 `.error`

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

```
.error "error"
```

Parameter

error

Error message, in double quotes.

3.2.17 `.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.

3.2.18 `.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*. For more information about debugging, refer to the [Debugging Directives](#).

Example

The following list shows how to use the `.file` directive for your own DWARF code.

Listing: DWARF Code Example

Native Assembler Directives

```
.file "MyFile.c"

.text

.function "MyFunction",start,end-start
start:

.line 1
lwz r3, 0(r3)

.line 2
blr

end:
```

3.2.19 .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.

3.2.20 .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. For more information about debugging, refer to the [Debugging Directives](#).

3.2.21 .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.

3.2.22 .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 statementsendif`. 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 1else ... assembly statements 2endif`. 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.

3.2.23 `.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.

3.2.24 `.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.

3.2.25 `.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.

3.2.26 `.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.

3.2.27 `.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.

3.2.28 `.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.

3.2.29 `.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.

3.2.30 `.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.

3.2.31 `.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.

3.2.32 `.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.

3.2.33 .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.

3.2.34 .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). For more information about debugging, refer to the [Debugging Directives](#).

3.2.35 .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.

3.2.36 .macro

Starts the definition of a macro.

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

Parameters

label

Name you give the macro.

parameter

Optional parameter for the macro.

3.2.37 .mexit

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

`.mexit`

3.2.38 `.offset`

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

`.offset [expression]`

Parameter

`expression`

Optional initial location-counter value.

Remarks

The following table lists the only directives you can use inside a record.

Table 3-5. Directives Allowed in a Record

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

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

Example

The following list shows a sample record definition.

Listing: Record Definition with Offset Directive

```

top:      .offset
         .short  0

left:     .short  0

bottom:  .short  0

right:   .short  0

rectSize .equ   *
```

3.2.39 .option

Sets an assembler control option. The following table lists and describes the *option* keywords.

```
.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.
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.
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 ().
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.

3.2.40 .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

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: Address-Change Example

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

NOTE

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

3.2.41 .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.

3.2.42 .previous

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

```
.previous
```

3.2.43 .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).

3.2.44 .rodata

Specifies an initialized read-only data section.

```
.rodata
```

3.2.45 **.sbss**

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

```
.sbss
```

3.2.46 **.sbss2**

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

```
.sbss2
```

3.2.47 **.sdata**

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

```
.sdata
```

3.2.48 **.sdata0**

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

```
.sdata2
```


3.2.49 .sdata2

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

```
.sdata2
```

3.2.50 .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. The default `type` value is 1: (SHT_PROGBITS).

flags

Numeric value for the ELF section flags. 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.

Table continues on the next page...

Table 3-7. ELF Section Header Types (SHT) (continued)

Type	Name	Meaning
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.
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
```

3.2.51 `.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.

3.2.52 `.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.

3.2.53 .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*. For more information about debugging, refer to the [Debugging Directives](#).

3.2.54 .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.

3.2.55 `.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.)

3.2.56 `.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"
```

3.2.57 .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*. For more information about debugging, refer to the [Debugging Directives](#).

3.3 Debugging Directives

When you enable the debugger, the assembler automatically generates some debug information for your project. However, you can use these directives in the debug section to provide additional information to the debugger.

NOTE

These debugging directives are valid only in the `.debug` and `.text` sections of an assembly file. Additionally, you must enable debugging for the file that contains the debugging directives.

In this topic:

- [.file](#)
- [.function](#)
- [.line](#)
- [.size](#)
- [.type](#)

3.3.1 .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

If you do not have `.line` directives in your code, using a `.file` directive will prevent the assembler from generating DWARF `.debug_line` information, and lines in the source file will not appear during debugging.

If you do have `.line` directives in your code, it is not necessary to use the `.file` directive, as the `.function` directive will provide source-level debugging.

Example

The following listing shows how to use the `.file` directive for your own DWARF code.

Listing: File directive for DWARF code

```
.file "MyFile.c"
.text
.function "MyFunction",start,end-start
start:
.line 5
ldr r3,[r2]
.line 7
blx
end:
```

3.3.2 .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

To ensure that the debugger can display your assembly source code while debugging, you must use the `.function` directive to identify the *NitroMain* function.

The following listing shows the identification of *NitroMain* using the `.function` directive.

Listing: Identifying NitroMain Using .function Directive

```
.function "main", main,main_end-main
main:
    and r1, r2, #1
    orr r1, r2, #1
    tst r1, r2
    mov r0, r2
lp:
    b lp
main_end:
```

3.3.3 .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

By default, the assembler generates all the necessary source line information for debugging purposes. If you want to exercise line-by-line control over the source line information, you can use the `.line` directive. If there is at least one `.line` directive in your code, the assembler generates source line information only for the code located immediately after the directive. All other source line information will be suppressed.

3.3.4 `.size`

Specifies a length for a symbol.

```
.size symbol, expression
```

Parameters

symbol

Symbol name.

expression

Number of bytes.

3.3.5 `.type`

Specifies the type of a symbol.

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

Parameters

symbol

Debugging Directives

Symbol name.

@function

Function type specifier.

@object

Variable specifier.

Chapter 4

Using Macros

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

In this chapter:

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

4.1 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. The following listing shows the full syntax.

Listing: Macro Definition Syntax: `.macro` Directive

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

The following table explains the syntax elements.

Table 4-1. Syntax Elements: .macro Directive

Element	Description
name	Label that invokes the macro.
parameter	Operand the assembler passes to the macro for use in the macro body.
macro_body	One or more assembly language statements. Invoking the macro tells the assembler to substitute 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.

The following listing shows the definition of macro `addto`, which includes a `.mexit` directive.

Listing: 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
add #1, 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 val, dest
// end macro definition
.endm
```

The following listing shows the assembly-language code that calls the `addto` macro.

Listing: Assembly Code that Calls addto Macro

```
// specify an executable code section
.text
xor d0,d0
// call the addto macro
addto d0,0
addto d0,1
addto d0,2
addto d0,3
```

The following listing shows the expanded `addto` macro calls.

Listing: Expanded addto Macro Calls

```
xor d0,d0
nop
add d0
add d0,2
add d0,3
```

4.1.1 Using Macro Arguments

You can refer to the parameters directly by name. The following listing shows the `setup` macro, which moves an integer into a register and branches to the label `_final_setup`.

Listing: Setup Macro Definition

```
setup:      .macro name
            mov     name,d0

            jsr    _final_setup

            .endm
```

The following listing shows a way to invoke the `setup` macro.

Listing: Calling Setup Macro

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

The following listing shows how the assembler expands the `setup` macro.

Listing: Expanding Setup Macro

```
move VECT, d0
jsr _final_setup
```

If you refer to the 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, The following listing shows the `smallnum` macro, which creates a small float by appending the string `E-20` to the macro argument.

Listing: Smallnum Macro Definition

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

            .endm
```

The following listing shows a way to invoke the `smallnum` macro.

Listing: Invoking Smallnum Macro

```
smallnum 10
```

The following listing shows how the assembler expands the `smallnum` macro.

Listing: Expanding Smallnum Macro

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

Macro syntax includes positional parameter references (this feature can provide compatibility with other assemblers). For example, The following listing shows a macro with positional references `\1` and `\2`.

Listing: Doit Macro Definition

```
doit:      .macro
           move    \1,d0

           jsr    \2

           .endm
```

The following listing shows an invocation of this macro, with parameter values `10` and `print`.

Listing: Invoking Doit Macro

```
doit      10,print
```

The following listing shows the macro expansion

Listing: Expanding Doit Macro

```
move     10,d0
jsr      print
```

4.1.2 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.

4.1.2.1 .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.

4.1.2.2 .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

The following listing specifies three repetitions of `.byte`, with successive `name` values 1, 2, and 3.

Listing: .irp Directive Example

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

The following listing shows this expansion.

Listing: .irp Example Expansion

Defining Macros

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

4.1.2.3 .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.

4.1.3 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 ??nxxx, 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.

The following listing shows a macro that uses unique labels and equates.

Listing: Unique Label Macro Definition

```
my_macro: .macro
    alpha\@ = my_count
my_count .set my_count + 1
    add alpha\@,d0
    jmp label\@
```



```

add d1,d0
label\@:
nop
.endm

```

The following listing shows two calls to the `my_macro` macro, with `my_count` initialized to 0.

Listing: Invoking `my_macro` Macro

```

my_count .set 0
my_macro
my_macro

```

The following listing shows the expanded `my_macro` code after the two calls.

Listing: Expanding `my_macro` Calls

```

alpha??0000 = my_count
my_count .set my_count + 1
add alpha??0000,d0
jmp label??0000
add d1,d0
label??0000
nop
alpha??0001 = my_count
my_count .set my_count + 1
add alpha??0001,d0
jmp label??0001
add d1,d0
label??0001
nop

```

4.1.4 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.

4.2 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, the following listing shows macro `pattern`, which repeats a pattern of bytes passed to it the number of times specified in the macro call.

Listing: Pattern Macro Definition

```

pattern:      .macro times,bytes                .rept times
               .byte bytes

```

Invoking Macros

```
.endr  
.endm
```

The following listing shows a statement that calls `pattern`, passing a parameter that includes a comma.

Listing: Macro Argument with Commas

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

The following listing is another example calling statement; the assembler generates the same code in response to the calling statement of either of the above listings.

Listing: Alternate Byte-Pattern Method

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

Chapter 5

Kinetis Assembler General Settings

When you create a Kinetis project, the IDE creates a set of Kinetis assembler properties for the project. This chapter explains the Kinetis assembler general settings.

5.1 Displaying Kinetis Assembler General Settings

To view and modify the general settings for the Kinetis assembler:

1. Right-click the Kinetis project, for which you want to modify the properties, in the **CodeWarrior Projects** view.
2. Select **Properties** from the context-menu.
3. The **Properties for <project>** dialog box appears.
4. Expand the **C/C++ Build** tree control and select **Settings**.
5. The **Settings** page appears in the right panel.
6. From the **Configuration** drop-down list, select the launch configuration for which you want to modify the build properties.
7. Click the **Tool Settings** tab.
8. Expand the **ARM Assembler** tree control and select **General**. The Kinetis assembler general properties appears in the right panel of the **Tool Settings** tab.
9. Modify the properties as per the requirements.

The following table lists and describes the general assembler options for Kinetis.

Table 5-1. Tool settings - Kinetis Assembler > General Options

Option	Description
Labels Must End With `:`	Clear if system does not require labels to end with colons. By default, the option is checked.
Directives Must Begin With `.`	Clear if the system does not require directives to start with periods. By default, the option is checked.

Table continues on the next page...

**Table 5-1. Tool settings - Kinetis Assembler > General Options
(continued)**

Option	Description
Allow Space In Operand Field	Clear to restrict the assembler from adding spaces in operand fields. By default, the option is checked.
Case-Sensitive Identifiers	Clear to instruct the assembler to ignore case in identifiers. By default, the option is checked.
Enable GNU Assembler Compatible Syntax	Equivalent to <code>-gnu</code> option, provides a level of assembly compatibility with GNU assembler
Enable ARM ADS Compatible Syntax	Equivalent to <code>-ads</code> option, this option allows assembly of ARM, Ltd. RVDS assembly syntax.
Other Flags	Specify additional command line options for the assembler; type in custom flags that are not otherwise available in the UI.

NOTE

For more information about the Kinetis assembler options, such as settings in the **ARM Assembler** panel and **ARM Assembler > Input** panel, refer to the *Microcontrollers V10.x Targeting Manual*. You can access the document from the following location: `<CWInstallDir>\MCU\Help\PDF;` `CWInstallDir` is the directory where CodeWarrior software is installed.

10. Click **Apply** to save the changes.
11. Click **OK**.

The **Properties for <project>** dialog box closes.

Index

[.align 41](#)
[.ascii 42](#)
[.asciz 42](#)
[.bss 43](#)
[.byte 43](#)
[.data 44](#)
[.debug 44](#)
[.double 44](#)
[.else 45](#)
[.elseif 45](#)
[.endian 46](#)
[.endif 47](#)
[.endm 47](#)
[.equ 47](#)
[.error 48](#)
[.extern 49](#)
[.file 49, 71](#)
[.float 50](#)
[.function 50, 72](#)
[.global 51](#)
[.if 51](#)
[.ifc 52](#)
[.ifdef 53](#)
[.ifeq 53](#)
[.ifge 54](#)
[.ifgt 54](#)
[.ifle 55](#)
[.iflt 55](#)
[.ifnc 56](#)
[.ifndef 56](#)
[.ifne 57](#)
[.include 58](#)
[.irp 79](#)
[.irpc 80](#)
[.line 58, 72](#)
[.long 59](#)
[.macro 59](#)
[.mexit 59](#)
[.offset 60](#)
[.option 61](#)
[.org 62](#)
[.pragma 62](#)
[.previous 63](#)
[.public 63](#)
[.rept 78](#)
[.rodata 63](#)
[.sbss 64](#)
[.sbss2 64](#)
[.sdata 64](#)
[.sdata0 64](#)
[.sdata2 65](#)
[.section 65](#)
[.set 67](#)

[.short 67](#)
[.size 68, 73](#)
[.space 68](#)
[.text 69](#)
[.textequ 69](#)
[.type 70, 73](#)
[#define 32](#)
[#elif 33](#)
[#else 34](#)
[#endif 35](#)
[#error 35](#)
[#if 35](#)
[#ifdef 36](#)
[#ifndef 37](#)
[#include 37](#)
[#line 38](#)
[#pragma 38](#)
[#undefine 39](#)

A

[Alignment 25](#)
[Arguments 77, 81](#)
[Assembler 39, 83](#)
[Assembly 13](#)

C

[Case-Sensitive 20](#)
[Character 22](#)
[Comments 24](#)
[Compatibility 25](#)
[Compatible 25](#)
[Constants 21, 22](#)
[Creating 80](#)

D

[Data 25](#)
[Debugging 70](#)
[Defining 75](#)
[Directives 31, 32, 39, 70, 78](#)
[Displaying 83](#)

E

[equal sign 48](#)
[Equates 19, 80](#)
[Expressions 23](#)
[Extensions 26, 28](#)

F

Floating-Point [22](#)

G

General [83](#)

GNU [25](#)

I

Identifiers [20](#)

Integer [21](#)

Invoking [81](#)

K

Kinetis [83](#)

L

Labels [15–18](#), [80](#)

Language [13](#)

Local [16](#)

M

Macro [77](#), [78](#)

Macros [75](#), [81](#)

N

Native [39](#)

Non-Local [16](#)

Number [81](#)

O

Option [25](#)

P

Preprocessor [32](#)

R

Referencing [17](#)

Relocatable [18](#)

Repeat [78](#)

S

Settings [83](#)

Statement [14](#)

Statements [13](#)

Supported [26](#)

Symbols [14](#)

Syntax [13](#), [14](#), [25](#)

U

Unique [80](#)

Unsupported [28](#)



How to Reach Us:

Home Page:

freescale.com

Web Support:

freescale.com/support

Information in this document is provided solely to enable system and software implementers to use Freescale products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. Freescale reserves the right to make changes without further notice to any products herein.

Freescale makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in Freescale data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including "typicals," must be validated for each customer application by customer's technical experts. Freescale does not convey any license under its patent rights nor the rights of others. Freescale sells products pursuant to standard terms and conditions of sale, which can be found at the following address: freescale.com/SalesTermsandConditions.

Freescale, the Freescale logo, and Kinetis are trademarks of Freescale Semiconductor, Inc., Reg. U.S. Pat. & Tm. Off. All other product or service names are the property of their respective owners. ARM is the registered trademark of ARM Limited. The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org.

© 2010–2014 Freescale Semiconductor, Inc.