# Contents

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


Freescale Semiconductor, Inc.
### Chapter 3
#### Using Directives

<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.8.3</td>
<td>Unsupported Extensions</td>
<td>28</td>
</tr>
</tbody>
</table>

#### 3.1 Preprocessor Directives

<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1.1</td>
<td>#define</td>
<td>32</td>
</tr>
<tr>
<td>3.1.2</td>
<td>#elif</td>
<td>33</td>
</tr>
<tr>
<td>3.1.3</td>
<td>#else</td>
<td>34</td>
</tr>
<tr>
<td>3.1.4</td>
<td>#endif</td>
<td>35</td>
</tr>
<tr>
<td>3.1.5</td>
<td>#error</td>
<td>35</td>
</tr>
<tr>
<td>3.1.6</td>
<td>#if</td>
<td>35</td>
</tr>
<tr>
<td>3.1.7</td>
<td>#ifdef</td>
<td>36</td>
</tr>
<tr>
<td>3.1.8</td>
<td>#ifndef</td>
<td>37</td>
</tr>
<tr>
<td>3.1.9</td>
<td>#include</td>
<td>37</td>
</tr>
<tr>
<td>3.1.10</td>
<td>#line</td>
<td>38</td>
</tr>
<tr>
<td>3.1.11</td>
<td>#pragma</td>
<td>38</td>
</tr>
<tr>
<td>3.1.12</td>
<td>#undefine</td>
<td>38</td>
</tr>
</tbody>
</table>

#### 3.2 Native Assembler Directives

<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.1</td>
<td>.align</td>
<td>41</td>
</tr>
<tr>
<td>3.2.2</td>
<td>.ascii</td>
<td>42</td>
</tr>
<tr>
<td>3.2.3</td>
<td>.asciz</td>
<td>42</td>
</tr>
<tr>
<td>3.2.4</td>
<td>.bss</td>
<td>43</td>
</tr>
<tr>
<td>3.2.5</td>
<td>.byte</td>
<td>43</td>
</tr>
<tr>
<td>3.2.6</td>
<td>.data</td>
<td>44</td>
</tr>
<tr>
<td>3.2.7</td>
<td>.debug</td>
<td>44</td>
</tr>
<tr>
<td>3.2.8</td>
<td>.double</td>
<td>44</td>
</tr>
<tr>
<td>3.2.9</td>
<td>.else</td>
<td>45</td>
</tr>
<tr>
<td>3.2.10</td>
<td>.elseif</td>
<td>45</td>
</tr>
<tr>
<td>3.2.11</td>
<td>.endian</td>
<td>46</td>
</tr>
<tr>
<td>3.2.12</td>
<td>.endif</td>
<td>47</td>
</tr>
<tr>
<td>Section number</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>----------------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>3.2.13</td>
<td>.endm</td>
<td>47</td>
</tr>
<tr>
<td>3.2.14</td>
<td>.equ</td>
<td>47</td>
</tr>
<tr>
<td>3.2.15</td>
<td>equal sign (=)</td>
<td>48</td>
</tr>
<tr>
<td>3.2.16</td>
<td>.error</td>
<td>48</td>
</tr>
<tr>
<td>3.2.17</td>
<td>.extern</td>
<td>48</td>
</tr>
<tr>
<td>3.2.18</td>
<td>.file</td>
<td>49</td>
</tr>
<tr>
<td>3.2.19</td>
<td>.float</td>
<td>50</td>
</tr>
<tr>
<td>3.2.20</td>
<td>.function</td>
<td>50</td>
</tr>
<tr>
<td>3.2.21</td>
<td>.global</td>
<td>51</td>
</tr>
<tr>
<td>3.2.22</td>
<td>.if</td>
<td>51</td>
</tr>
<tr>
<td>3.2.23</td>
<td>.ifc</td>
<td>52</td>
</tr>
<tr>
<td>3.2.24</td>
<td>.ifdef</td>
<td>53</td>
</tr>
<tr>
<td>3.2.25</td>
<td>.ifeq</td>
<td>53</td>
</tr>
<tr>
<td>3.2.26</td>
<td>.ifge</td>
<td>54</td>
</tr>
<tr>
<td>3.2.27</td>
<td>.ifgt</td>
<td>54</td>
</tr>
<tr>
<td>3.2.28</td>
<td>.ifle</td>
<td>55</td>
</tr>
<tr>
<td>3.2.29</td>
<td>.iflt</td>
<td>55</td>
</tr>
<tr>
<td>3.2.30</td>
<td>.ifnc</td>
<td>56</td>
</tr>
<tr>
<td>3.2.31</td>
<td>.ifndef</td>
<td>56</td>
</tr>
<tr>
<td>3.2.32</td>
<td>.ifne</td>
<td>57</td>
</tr>
<tr>
<td>3.2.33</td>
<td>.include</td>
<td>58</td>
</tr>
<tr>
<td>3.2.34</td>
<td>.line</td>
<td>58</td>
</tr>
<tr>
<td>3.2.35</td>
<td>.long</td>
<td>59</td>
</tr>
<tr>
<td>3.2.36</td>
<td>.macro</td>
<td>59</td>
</tr>
<tr>
<td>3.2.37</td>
<td>.mexit</td>
<td>59</td>
</tr>
<tr>
<td>3.2.38</td>
<td>.offset</td>
<td>60</td>
</tr>
<tr>
<td>3.2.39</td>
<td>.option</td>
<td>60</td>
</tr>
<tr>
<td>3.2.40</td>
<td>.org</td>
<td>61</td>
</tr>
<tr>
<td>3.2.41</td>
<td>.pragma</td>
<td>62</td>
</tr>
<tr>
<td>Section number</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>----------------</td>
<td>------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>3.2.42</td>
<td>.previous</td>
<td>63</td>
</tr>
<tr>
<td>3.2.43</td>
<td>.public</td>
<td>63</td>
</tr>
<tr>
<td>3.2.44</td>
<td>.rodata</td>
<td>63</td>
</tr>
<tr>
<td>3.2.45</td>
<td>.sbss</td>
<td>64</td>
</tr>
<tr>
<td>3.2.46</td>
<td>.sbss2</td>
<td>64</td>
</tr>
<tr>
<td>3.2.47</td>
<td>.sdata</td>
<td>64</td>
</tr>
<tr>
<td>3.2.48</td>
<td>.sdata0</td>
<td>64</td>
</tr>
<tr>
<td>3.2.49</td>
<td>.sdata2</td>
<td>65</td>
</tr>
<tr>
<td>3.2.50</td>
<td>.section</td>
<td>65</td>
</tr>
<tr>
<td>3.2.51</td>
<td>.set</td>
<td>67</td>
</tr>
<tr>
<td>3.2.52</td>
<td>.short</td>
<td>67</td>
</tr>
<tr>
<td>3.2.53</td>
<td>.size</td>
<td>68</td>
</tr>
<tr>
<td>3.2.54</td>
<td>.space</td>
<td>68</td>
</tr>
<tr>
<td>3.2.55</td>
<td>.text</td>
<td>68</td>
</tr>
<tr>
<td>3.2.56</td>
<td>.textequ</td>
<td>69</td>
</tr>
<tr>
<td>3.2.57</td>
<td>.type</td>
<td>69</td>
</tr>
<tr>
<td>3.3</td>
<td>Debugging Directives</td>
<td>70</td>
</tr>
<tr>
<td>3.3.1</td>
<td>.file</td>
<td>71</td>
</tr>
<tr>
<td>3.3.2</td>
<td>.function</td>
<td>71</td>
</tr>
<tr>
<td>3.3.3</td>
<td>.line</td>
<td>72</td>
</tr>
<tr>
<td>3.3.4</td>
<td>.size</td>
<td>73</td>
</tr>
<tr>
<td>3.3.5</td>
<td>.type</td>
<td>73</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.1</td>
<td>Defining Macros</td>
<td>75</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Using Macro Arguments</td>
<td>77</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Macro Repeat Directives</td>
<td>78</td>
</tr>
<tr>
<td>4.1.2.1</td>
<td>.rept</td>
<td>78</td>
</tr>
<tr>
<td>4.1.2.2</td>
<td>.irp</td>
<td>79</td>
</tr>
</tbody>
</table>

Chapter 4
Using Macros
<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1.2.3</td>
<td>.irpc.............................................................................</td>
<td>80</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Creating Unique Labels and Equates................................</td>
<td>80</td>
</tr>
<tr>
<td>4.1.4</td>
<td>Number of Arguments..................................................</td>
<td>81</td>
</tr>
<tr>
<td>4.2</td>
<td>Invoking Macros..........................................................</td>
<td>81</td>
</tr>
</tbody>
</table>

### Chapter 5

**Kinetis Assembler General Settings**

<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Displaying Kinetis Assembler General Settings................</td>
<td>83</td>
</tr>
</tbody>
</table>
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 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. 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>
<thead>
<tr>
<th>Chapter Title</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>Describes an overview about this manual.</td>
</tr>
<tr>
<td>Assembly Language Syntax</td>
<td>Describes the main syntax of assembly language statements.</td>
</tr>
<tr>
<td>Using Directives</td>
<td>Describes the assembler directives.</td>
</tr>
<tr>
<td>Using Macros</td>
<td>Describes how to define and invoke macros.</td>
</tr>
<tr>
<td>Kinetis Assembler General Settings</td>
<td>Describes the assembler settings that are common among the assemblers.</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>symbol</td>
<td>A combination of characters that represents a value.</td>
</tr>
<tr>
<td>machine_instruction</td>
<td>A machine instruction for your target processor.</td>
</tr>
<tr>
<td>assembler_directive</td>
<td>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.</td>
</tr>
<tr>
<td>macro_call</td>
<td>A statement that calls a previously defined macro.</td>
</tr>
<tr>
<td>constant</td>
<td>A defined value, such as a string of characters or a numeric value.</td>
</tr>
<tr>
<td>expression</td>
<td>A mathematical expression.</td>
</tr>
<tr>
<td>register_name</td>
<td>The name of a register; these names are processor-specific.</td>
</tr>
<tr>
<td>comment</td>
<td>Text that the assembler ignores, useful for documenting your code.</td>
</tr>
</tbody>
</table>

### 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:

\[
symbol ::= label \mid equate
\]

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:

\[
label ::= local_label [ : ] \mid non-local_label[ : ]
\]

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**

```assembly
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**

```c
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

<table>
<thead>
<tr>
<th>Expression</th>
<th>Represents</th>
</tr>
</thead>
<tbody>
<tr>
<td>label</td>
<td>The offset from the address of the label to the base of its section, relocated by the section base address. Also, it is the PC-relative target of a branch or call. It is a 32-bit address.</td>
</tr>
</tbody>
</table>

Table continues on the next page...
Table 2-2. Relocatable Label Expressions (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Represents</th>
</tr>
</thead>
<tbody>
<tr>
<td>label@l</td>
<td>The low 16-bits of the relocated address of the symbol.</td>
</tr>
<tr>
<td>label@h</td>
<td>The high 16-bits of the relocated address of the symbol. You can OR this with label@l to produce the full 32-bit relocated address.</td>
</tr>
<tr>
<td>label@ha</td>
<td>The adjusted high 16-bits of the relocated address of the symbol. You can add this to label@l to produce the full 32-bit relocated address.</td>
</tr>
<tr>
<td>label@sdax</td>
<td>For labels in a small data section, the offset from the base of the small data section to the label. This syntax is not allowed for labels in other sections.</td>
</tr>
<tr>
<td>label@got</td>
<td>For processors with a global offset table, the offset from the base of the global offset table to the 32-bit entry for label.</td>
</tr>
</tbody>
</table>

### 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.
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**

```assembly
.data
.long alloc_size
alloc_size .set rec_size + 4
; a valid forward equate on next line
rec_size .set table_start-table_end
.text;
..table_start:
; ...
table_end:
```

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**

```assembly
;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>
<thead>
<tr>
<th>Type</th>
<th>Preferred Notation</th>
<th>Alternate Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hexadecimal</td>
<td>0x followed by a string of hexadecimal digits, such as 0xdeadbeef.</td>
<td>$ 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.</td>
</tr>
<tr>
<td>Decimal</td>
<td>String of decimal digits, such as 12345678.</td>
<td>String of decimal digits followed by d, such as 12345678d.</td>
</tr>
<tr>
<td>Binary</td>
<td>% followed by a string of binary digits, such as %01010001.</td>
<td>0b followed by a string of binary digits, such as 0b01010001.</td>
</tr>
</tbody>
</table>

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 \textit{1E-10} and \textit{1.0}.

You can use floating-point constants only in data generation directives such as \texttt{.float} and \texttt{.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.

\textbf{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 \texttt{‘A’}, \texttt{‘ABC’}, and \texttt{‘TEXT’}.

A character constant can contain any of the escape sequences that the following table lists.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
Sequence & Description \\
\hline
\texttt{\ \b} & Backspace \\
\texttt{\n} & Line feed (ASCII character 10) \\
\texttt{\r} & Return (ASCII character 13) \\
\texttt{\t} & Tab \\
\texttt{\'} & Single quote \\
\texttt{\"} & Double quote \\
\texttt{\\} & Backslash \\
\texttt{\xnn} & Hexadecimal value of \texttt{nn} \\
\texttt{\nnn} & Octal value of \texttt{nn} \\
\hline
\end{tabular}
\end{table}

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 evaluate 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>
<thead>
<tr>
<th>Table 2-5. Expression Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Category</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>Binary</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Binary</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Unary</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Alternate</td>
</tr>
</tbody>
</table>

The operator precedence is:
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>
<thead>
<tr>
<th>Case</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defining Equates</td>
<td>Whether defined using <code>.equ</code> or <code>.set</code>, all equates can be re-defined.</td>
</tr>
<tr>
<td>Ignored directives</td>
<td>The <code>.type</code> directive is ignored.</td>
</tr>
<tr>
<td>Undefined Symbols</td>
<td>Undefined symbols are automatically treated as imported</td>
</tr>
</tbody>
</table>
| Arithmetic Operators| • `< and >` mean left-shift and right-shift instead of less than and greater than.  
|                     | • `< ! >` 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, `.byte 'a` |

### 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 `0b`.
- 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>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>.abort</td>
<td>End assembly</td>
<td>Supported</td>
</tr>
<tr>
<td>.align N,[pad]</td>
<td>Align</td>
<td>Now accepts optional padding byte</td>
</tr>
<tr>
<td>.app-file name</td>
<td>Source name</td>
<td>Synonym for .file</td>
</tr>
<tr>
<td>.balign[wl] N,[pad]</td>
<td>Align</td>
<td>Align to N (with optional padding value)</td>
</tr>
<tr>
<td>.common name,length</td>
<td>Common data</td>
<td>Reserve space in BSS for global symbol</td>
</tr>
<tr>
<td>.def</td>
<td>Debugging</td>
<td>Accepted but ignored</td>
</tr>
<tr>
<td>.desc</td>
<td>Debugging</td>
<td>Accepted but ignored</td>
</tr>
<tr>
<td>.dim</td>
<td>Debugging</td>
<td>Accepted but ignored</td>
</tr>
<tr>
<td>.eject</td>
<td>Eject page</td>
<td>Accepted but ignored</td>
</tr>
<tr>
<td>.endr</td>
<td>End repeat</td>
<td>refer .irp, .irpc</td>
</tr>
<tr>
<td>.endef</td>
<td>Debugging</td>
<td>Accepted but ignored</td>
</tr>
<tr>
<td>.fill N,[size],[val]</td>
<td>Repeat data</td>
<td>Emit N copies of width <code>size', value </code>val'</td>
</tr>
<tr>
<td>.hword val..</td>
<td>Half-word</td>
<td>Synonym for .short</td>
</tr>
<tr>
<td>.ident</td>
<td>Tags</td>
<td>Accepted but ignored</td>
</tr>
<tr>
<td>.ifnotdef name</td>
<td>Conditional</td>
<td>Synonym for .ifndef</td>
</tr>
<tr>
<td>.include name</td>
<td>Include file</td>
<td>Now accepts single, double or no quotes</td>
</tr>
<tr>
<td>.init val..</td>
<td>Word</td>
<td>Synonym for .long</td>
</tr>
<tr>
<td>.irp name,values</td>
<td>Repeat</td>
<td>Repeat up to .endr substituting values for name</td>
</tr>
<tr>
<td>.irpc name,chars</td>
<td>Repeat</td>
<td>Repeat up to .endr substituting chars for name</td>
</tr>
<tr>
<td>.lcomm name,length</td>
<td>Local common</td>
<td>Reserve length bytes in bss</td>
</tr>
<tr>
<td>.iflags</td>
<td>Ignored</td>
<td>Accepted but ignored</td>
</tr>
<tr>
<td>.ln lineno</td>
<td>Line number</td>
<td>Synonym for .line</td>
</tr>
<tr>
<td>.list</td>
<td>Listing on</td>
<td>Switch on listing</td>
</tr>
<tr>
<td>.local name</td>
<td>Local Macro var</td>
<td>Declare name as local to macro</td>
</tr>
<tr>
<td>.macro name, args..</td>
<td>Macros</td>
<td>Supports GNU syntax, default values, etc</td>
</tr>
<tr>
<td>.nolist</td>
<td>Listing off</td>
<td>Disable listing</td>
</tr>
<tr>
<td>.org pos,fill</td>
<td>Origin</td>
<td>Now allows fill value to be specified</td>
</tr>
<tr>
<td>.palign[wl] N[,pad]</td>
<td>Align</td>
<td>Align to 2**N, using pad value `pad'</td>
</tr>
<tr>
<td>.psize</td>
<td>Page size</td>
<td>Accepted but ignored</td>
</tr>
<tr>
<td>.rept N</td>
<td>Repeat</td>
<td>Repeat block up to .endr N times</td>
</tr>
<tr>
<td>.sbttl</td>
<td>Subtitle</td>
<td>Accepted but ignored</td>
</tr>
</tbody>
</table>

Table continues on the next page...
<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>.scl</td>
<td>Debugging</td>
<td>Accepted but ignored</td>
</tr>
<tr>
<td>.size name,N</td>
<td>Set size</td>
<td>Set size of name to N</td>
</tr>
<tr>
<td>.skip N[,pad]</td>
<td>Space</td>
<td>Skip N bytes, pad with 'pad'</td>
</tr>
<tr>
<td>.space N[,pad]</td>
<td>Space</td>
<td>Skip N bytes, pad with 'pad'</td>
</tr>
<tr>
<td>.stabd</td>
<td>Debugging</td>
<td>Accepted but ignored</td>
</tr>
<tr>
<td>.stabs</td>
<td>Debugging</td>
<td>Accepted but ignored</td>
</tr>
<tr>
<td>.stabs</td>
<td>Debugging</td>
<td>Accepted but ignored</td>
</tr>
<tr>
<td>.str &quot;string&quot;</td>
<td>Constant string</td>
<td>Synonym for .asciz</td>
</tr>
<tr>
<td>.string &quot;string&quot;</td>
<td>Constant String</td>
<td>Synonym for .asciz</td>
</tr>
<tr>
<td>.tag</td>
<td>Debugging</td>
<td>Accepted but ignored</td>
</tr>
<tr>
<td>.title</td>
<td>Title</td>
<td>Accepted but ignored</td>
</tr>
<tr>
<td>.type</td>
<td>Debugging</td>
<td>Ignored in GNU mode</td>
</tr>
<tr>
<td>.val</td>
<td>Debugging</td>
<td>Accepted but ignored</td>
</tr>
<tr>
<td>.word</td>
<td>Word</td>
<td>Synonym for .long</td>
</tr>
</tbody>
</table>

### 2.8.3 Unsupported Extensions

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

<table>
<thead>
<tr>
<th>Extension</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sub-sections (such as .text 2)</td>
<td>The sub-section number is ignored. As a workaround, you can create your own sections with the .section &lt;name&gt; directive. You may have an arbitrary number of text subsections with the names .text1, .text2, etc.</td>
</tr>
<tr>
<td>Assignment to location counter (such as . = .+4)</td>
<td>As a workaround, you can advance the location counter with .space &lt;expr&gt;</td>
</tr>
<tr>
<td>Empty expressions defaulting to 0. For example, .byte , equivalent to (.byte 0,0)</td>
<td>There is no workaround for this. You must always supply the arguments.</td>
</tr>
</tbody>
</table>
| .linkonce 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:  
  - 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)

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

```c
#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**

```c
#define        ABC  MyVal
#ifndef ABC       ;Definition test
```

**Listing: Native-Assembler-Directive Example**

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

Freescale Semiconductor, Inc.
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

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

#### 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
• 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.
Preprocessor Directives

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**
Preprocessor Directives

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

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

*Table continues on the next page...*
Table 3-2. Assembler Directives (continued)

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

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.

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

### 3.2.3 .asciz

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

```assembly
[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>
<thead>
<tr>
<th>Sequence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\b</td>
<td>Backspace</td>
</tr>
<tr>
<td>\n</td>
<td>Line feed (ASCII character 10)</td>
</tr>
<tr>
<td>\r</td>
<td>Return (ASCII character 13)</td>
</tr>
<tr>
<td>\t</td>
<td>Tab</td>
</tr>
<tr>
<td>'</td>
<td>Single quote</td>
</tr>
<tr>
<td>&quot;</td>
<td>Double quote</td>
</tr>
<tr>
<td>\</td>
<td>Backslash</td>
</tr>
<tr>
<td>\nnn</td>
<td>Octal value of \nnn</td>
</tr>
<tr>
<td>\xnn</td>
<td>Hexadecimal value of nn</td>
</tr>
</tbody>
</table>

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.

eexpression

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] ... .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*.

### 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 .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.)

3.2.11 .endiian
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.

```assembly
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.

```assembly
.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.

```
.globallabel [, 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.
Native Assembler Directives

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.

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

Table 3-5. Directives Allowed in a Record

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 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**

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alignment off</td>
<td>on</td>
</tr>
<tr>
<td>case off</td>
<td>on</td>
</tr>
<tr>
<td>colon off</td>
<td>on</td>
</tr>
<tr>
<td>no_section_resume on</td>
<td>off</td>
</tr>
<tr>
<td>period off</td>
<td>on</td>
</tr>
<tr>
<td>processor procname</td>
<td>reset</td>
</tr>
<tr>
<td>space off</td>
<td>on</td>
</tr>
</tbody>
</table>
3.2.40 .org

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

.org expression

Parameter

description

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
Native Assembler Directives

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 value is 1: (SHT_PROGBITS).

- **flags**
  
  Numeric value for the ELF section flags. The default value is 0x00000002, 0x00000001: (SHF_ALLOC+SHF_WRITE).

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>Section header is inactive.</td>
</tr>
<tr>
<td>1</td>
<td>PROGBITS</td>
<td>Section contains information that the program defines.</td>
</tr>
<tr>
<td>2</td>
<td>SYMTAB</td>
<td>Section contains a symbol table.</td>
</tr>
<tr>
<td>3</td>
<td>STRTAB</td>
<td>Section contains a string table.</td>
</tr>
<tr>
<td>4</td>
<td>RELA</td>
<td>Section contains relocation entries with explicit addends.</td>
</tr>
</tbody>
</table>

*Table continues on the next page.*
<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>HASH</td>
<td>Section contains a symbol hash table.</td>
</tr>
<tr>
<td>6</td>
<td>DYNAMIC</td>
<td>Section contains information used for dynamic linking.</td>
</tr>
<tr>
<td>7</td>
<td>NOTE</td>
<td>Section contains information that marks the file, often for compatibility purposes between programs.</td>
</tr>
<tr>
<td>8</td>
<td>NOBITS</td>
<td>Section occupies no space in the object file.</td>
</tr>
<tr>
<td>9</td>
<td>REL</td>
<td>Section contains relocation entries without explicit addends.</td>
</tr>
<tr>
<td>10</td>
<td>SHLIB</td>
<td>Section has unspecified semantics, so does not conform to the Application Binary Interface (ABI) standard.</td>
</tr>
<tr>
<td>11</td>
<td>DYNSYM</td>
<td>Section contains a minimal set of symbols for dynamic linking.</td>
</tr>
</tbody>
</table>

Table 3-8. ELF Section Header Flags (SHF)

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

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
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

```assembly
name: .macro [ parameter ] [ ,parameter ] ...
macro_body
.endm
```
The following table explains the syntax elements.

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Label that invokes the macro.</td>
</tr>
<tr>
<td>parameter</td>
<td>Operand the assembler passes to the macro for use in the macro body.</td>
</tr>
<tr>
<td>macro_body</td>
<td>One or more assembly language statements. Invoking the macro tell the assembler to substitutes these statements.</td>
</tr>
</tbody>
</table>

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

```assembly
//define a macro
addto .macro dest,val
  .if val==0
    no-op  // execution goes to the statement after .mexit
  .elseif val==1
    add #1, dest  // use compact instruction
  .endif
  add val, dest  // if val is not equal to either 0 or 1, add dest and val
// end macro definition
.endm
```

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

### Listing: Assembly Code that Calls addto Macro

```assembly
// 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
```
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
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 ??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.

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

**Listing: Unique Label Macro Definition**

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

---


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

**Listing: Invoking my_macro Macro**

```assembly
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**

```assembly
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**

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

Freescale Semiconductor, Inc.
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

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Labels Must End With <code>:</code></td>
<td>Clear if system does not require labels to end with colons. By default, the option is checked.</td>
</tr>
<tr>
<td>Directives Must Begin With <code>.</code></td>
<td>Clear if the system does not require directives to start with periods. By default, the option is checked.</td>
</tr>
</tbody>
</table>

Table continues on the next page...
Table 5-1. Tool settings - Kinetis Assembler > General Options  
(continued)

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

**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
.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
.sbs 64
.sbs2 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
(=) 48
#define 32
#elif 33
#else 34
#endif 35
#error 35
#if 35
#ifdef 36
#ifndef 37
#include 37
#line 38
#pragma 38
#undef 39

A
Accompanying 10
Alignment 25
Arguments 77, 81
Assembler 39, 83
Assembly 13

B
Book 9

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
Documentation 10
Index

E
equal 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
Learn 10
Local 16

M
Macro 77, 78
Macros 75, 81
More 10

N
Native 39
Non-Local 16
Notes 9
Number 81

O
Option 25

P
Preprocessor 32

R
Referencing 17
Release 9
Relocatable 18
Repeat 78

S
Settings 83
sign 48
Statement 14
Statements 13
Supported 26
Symbols 14
Syntax 13, 14, 25

T
This 9

U
Unique 80
Unsupported 28

W
Where 10