
Document Number: CWPABTR
Rev. 10.x, 06/2015
## Contents

<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Compiler Architecture</td>
<td>35</td>
</tr>
<tr>
<td>1.2</td>
<td>Linker Architecture</td>
<td>36</td>
</tr>
<tr>
<td>2.1</td>
<td>Configuring Command-Line Tools</td>
<td>39</td>
</tr>
<tr>
<td>2.1.1</td>
<td>Setting CodeWarrior Environment Variables</td>
<td>39</td>
</tr>
<tr>
<td>2.1.2</td>
<td>Setting the PATH Environment Variable</td>
<td>40</td>
</tr>
<tr>
<td>2.2</td>
<td>Invoking Command-Line Tools</td>
<td>41</td>
</tr>
<tr>
<td>2.3</td>
<td>Getting Help</td>
<td>42</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Parameter Formats</td>
<td>43</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Option Formats</td>
<td>43</td>
</tr>
<tr>
<td>2.3.3</td>
<td>Common Terms</td>
<td>43</td>
</tr>
<tr>
<td>2.4</td>
<td>File Name Extensions</td>
<td>44</td>
</tr>
<tr>
<td>3.1</td>
<td>-ansi</td>
<td>45</td>
</tr>
<tr>
<td>3.2</td>
<td>-stdkeywords</td>
<td>46</td>
</tr>
<tr>
<td>3.3</td>
<td>-strict</td>
<td>46</td>
</tr>
<tr>
<td>4.1</td>
<td>-ARM</td>
<td>47</td>
</tr>
<tr>
<td>4.2</td>
<td>-bool</td>
<td>47</td>
</tr>
<tr>
<td>4.3</td>
<td>-Cpp_exceptions</td>
<td>48</td>
</tr>
<tr>
<td>4.4</td>
<td>-dialect</td>
<td>48</td>
</tr>
<tr>
<td>4.5</td>
<td>-for_scoping</td>
<td>49</td>
</tr>
<tr>
<td>4.6</td>
<td>-instmgr</td>
<td>49</td>
</tr>
<tr>
<td>Section number</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>----------------</td>
<td>--------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>4.7</td>
<td>iso_templates</td>
<td>50</td>
</tr>
<tr>
<td>4.8</td>
<td>RTTI</td>
<td>50</td>
</tr>
<tr>
<td>4.9</td>
<td>som</td>
<td>50</td>
</tr>
<tr>
<td>4.10</td>
<td>som_env_check</td>
<td>50</td>
</tr>
<tr>
<td>4.11</td>
<td>wchar_t</td>
<td>51</td>
</tr>
<tr>
<td>5.1</td>
<td>char</td>
<td>53</td>
</tr>
<tr>
<td>5.2</td>
<td>defaults</td>
<td>54</td>
</tr>
<tr>
<td>5.3</td>
<td>encoding</td>
<td>54</td>
</tr>
<tr>
<td>5.4</td>
<td>flag</td>
<td>55</td>
</tr>
<tr>
<td>5.5</td>
<td>gccext</td>
<td>56</td>
</tr>
<tr>
<td>5.6</td>
<td>gcc_extensions</td>
<td>56</td>
</tr>
<tr>
<td>5.7</td>
<td>M</td>
<td>56</td>
</tr>
<tr>
<td>5.8</td>
<td>make</td>
<td>57</td>
</tr>
<tr>
<td>5.9</td>
<td>mapcr</td>
<td>57</td>
</tr>
<tr>
<td>5.10</td>
<td>MM</td>
<td>57</td>
</tr>
<tr>
<td>5.11</td>
<td>MD</td>
<td>58</td>
</tr>
<tr>
<td>5.12</td>
<td>MMD</td>
<td>58</td>
</tr>
<tr>
<td>5.13</td>
<td>msext</td>
<td>58</td>
</tr>
<tr>
<td>5.14</td>
<td>once</td>
<td>59</td>
</tr>
<tr>
<td>5.15</td>
<td>pragma</td>
<td>59</td>
</tr>
<tr>
<td>5.16</td>
<td>relax_pointers</td>
<td>60</td>
</tr>
<tr>
<td>5.17</td>
<td>requireprotos</td>
<td>60</td>
</tr>
<tr>
<td>5.18</td>
<td>search</td>
<td>60</td>
</tr>
<tr>
<td>5.19</td>
<td>trigraphs</td>
<td>60</td>
</tr>
</tbody>
</table>

**Chapter 5**

Command-Line Options for Language Translation

<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>char</td>
<td>53</td>
</tr>
<tr>
<td>5.2</td>
<td>defaults</td>
<td>54</td>
</tr>
<tr>
<td>5.3</td>
<td>encoding</td>
<td>54</td>
</tr>
<tr>
<td>5.4</td>
<td>flag</td>
<td>55</td>
</tr>
<tr>
<td>5.5</td>
<td>gccext</td>
<td>56</td>
</tr>
<tr>
<td>5.6</td>
<td>gcc_extensions</td>
<td>56</td>
</tr>
<tr>
<td>5.7</td>
<td>M</td>
<td>56</td>
</tr>
<tr>
<td>5.8</td>
<td>make</td>
<td>57</td>
</tr>
<tr>
<td>5.9</td>
<td>mapcr</td>
<td>57</td>
</tr>
<tr>
<td>5.10</td>
<td>MM</td>
<td>57</td>
</tr>
<tr>
<td>5.11</td>
<td>MD</td>
<td>58</td>
</tr>
<tr>
<td>5.12</td>
<td>MMD</td>
<td>58</td>
</tr>
<tr>
<td>5.13</td>
<td>msext</td>
<td>58</td>
</tr>
<tr>
<td>5.14</td>
<td>once</td>
<td>59</td>
</tr>
<tr>
<td>5.15</td>
<td>pragma</td>
<td>59</td>
</tr>
<tr>
<td>5.16</td>
<td>relax_pointers</td>
<td>60</td>
</tr>
<tr>
<td>5.17</td>
<td>requireprotos</td>
<td>60</td>
</tr>
<tr>
<td>5.18</td>
<td>search</td>
<td>60</td>
</tr>
<tr>
<td>5.19</td>
<td>trigraphs</td>
<td>60</td>
</tr>
</tbody>
</table>

**Chapter 6**

Command-Line Options for Diagnostic Messages

<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>disassemble</td>
<td>63</td>
</tr>
<tr>
<td>Section number</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>6.2</td>
<td>-warningerror</td>
<td>64</td>
</tr>
<tr>
<td>6.3</td>
<td>-help</td>
<td>64</td>
</tr>
<tr>
<td>6.4</td>
<td>-maxerrors</td>
<td>65</td>
</tr>
<tr>
<td>6.5</td>
<td>-maxwarnings</td>
<td>66</td>
</tr>
<tr>
<td>6.6</td>
<td>-msgstyle</td>
<td>66</td>
</tr>
<tr>
<td>6.7</td>
<td>-ide_err</td>
<td>67</td>
</tr>
<tr>
<td>6.8</td>
<td>-nofail</td>
<td>67</td>
</tr>
<tr>
<td>6.9</td>
<td>-progress</td>
<td>67</td>
</tr>
<tr>
<td>6.10</td>
<td>-S</td>
<td>68</td>
</tr>
<tr>
<td>6.11</td>
<td>-stderr</td>
<td>68</td>
</tr>
<tr>
<td>6.12</td>
<td>-verbose</td>
<td>68</td>
</tr>
<tr>
<td>6.13</td>
<td>-version</td>
<td>69</td>
</tr>
<tr>
<td>6.14</td>
<td>-timing</td>
<td>69</td>
</tr>
<tr>
<td>6.15</td>
<td>-warnings</td>
<td>69</td>
</tr>
<tr>
<td>6.16</td>
<td>-wraplines</td>
<td>75</td>
</tr>
</tbody>
</table>

## Chapter 7
**Command-Line Options for Preprocessing**

<p>| 7.1           | -convertpaths                           | 77   |
| 7.2           | -cwd                                    | 78   |
| 7.3           | -D+                                     | 79   |
| 7.4           | -define                                 | 79   |
| 7.5           | -E                                      | 80   |
| 7.6           | -EP                                     | 80   |
| 7.7           | -gccincludes                            | 80   |
| 7.8           | -I-                                    | 81   |
| 7.9           | -I+                                    | 81   |
| 7.10          | -include                                | 82   |
| 7.11          | -ir                                    | 82   |
| 7.12          | -P                                      | 83   |</p>
<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.13</td>
<td>-precompile</td>
<td>83</td>
</tr>
<tr>
<td>7.14</td>
<td>-preprocess</td>
<td>84</td>
</tr>
<tr>
<td>7.15</td>
<td>-ppopt</td>
<td>84</td>
</tr>
<tr>
<td>7.16</td>
<td>-prefix</td>
<td>85</td>
</tr>
<tr>
<td>7.17</td>
<td>-noprecompile</td>
<td>85</td>
</tr>
<tr>
<td>7.18</td>
<td>-nosyspath</td>
<td>85</td>
</tr>
<tr>
<td>7.19</td>
<td>-stdinc</td>
<td>86</td>
</tr>
<tr>
<td>7.20</td>
<td>-U+</td>
<td>86</td>
</tr>
<tr>
<td>7.21</td>
<td>-undefine</td>
<td>86</td>
</tr>
</tbody>
</table>

**Chapter 8**

**Command-Line Options for Library and Linking**

<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1</td>
<td>-keepobjects</td>
<td>89</td>
</tr>
<tr>
<td>8.2</td>
<td>-nolink</td>
<td>89</td>
</tr>
<tr>
<td>8.3</td>
<td>-o</td>
<td>90</td>
</tr>
</tbody>
</table>

**Chapter 9**

**Command-Line Options for Object Code**

<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1</td>
<td>-c</td>
<td>91</td>
</tr>
<tr>
<td>9.2</td>
<td>-codegen</td>
<td>91</td>
</tr>
<tr>
<td>9.3</td>
<td>-enum</td>
<td>92</td>
</tr>
<tr>
<td>9.4</td>
<td>-min_enum_size</td>
<td>92</td>
</tr>
<tr>
<td>9.5</td>
<td>-ext</td>
<td>92</td>
</tr>
<tr>
<td>9.6</td>
<td>-strings</td>
<td>93</td>
</tr>
</tbody>
</table>

**Chapter 10**

**Command-Line Options for Optimization**

<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.1</td>
<td>-inline</td>
<td>95</td>
</tr>
<tr>
<td>10.2</td>
<td>-ipa</td>
<td>96</td>
</tr>
<tr>
<td>10.3</td>
<td>-O</td>
<td>98</td>
</tr>
<tr>
<td>10.4</td>
<td>-O+</td>
<td>98</td>
</tr>
<tr>
<td>10.5</td>
<td>-opt</td>
<td>99</td>
</tr>
</tbody>
</table>
## Chapter 11
### Command-Line for Power Architecture Processors

<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.1</td>
<td>Naming Conventions</td>
<td>103</td>
</tr>
<tr>
<td>11.2</td>
<td>Specifying Source File Locations</td>
<td>103</td>
</tr>
<tr>
<td>11.3</td>
<td>Licensing Command-Line Options</td>
<td>104</td>
</tr>
<tr>
<td>11.3.1</td>
<td>-fullLicenseSearch</td>
<td>104</td>
</tr>
<tr>
<td>11.3.2</td>
<td>-license</td>
<td>105</td>
</tr>
<tr>
<td>11.4</td>
<td>Diagnostic Command-Line Options</td>
<td>105</td>
</tr>
<tr>
<td>11.4.1</td>
<td>-g</td>
<td>106</td>
</tr>
<tr>
<td>11.4.2</td>
<td>-gdwarf-2</td>
<td>106</td>
</tr>
<tr>
<td>11.4.3</td>
<td>-fmt</td>
<td>107</td>
</tr>
<tr>
<td>11.4.4</td>
<td>-format</td>
<td>107</td>
</tr>
<tr>
<td>11.4.5</td>
<td>-listclosure</td>
<td>107</td>
</tr>
<tr>
<td>11.4.6</td>
<td>-listdwarf</td>
<td>108</td>
</tr>
<tr>
<td>11.4.7</td>
<td>-map</td>
<td>108</td>
</tr>
<tr>
<td>11.4.8</td>
<td>-mapunused</td>
<td>109</td>
</tr>
<tr>
<td>11.4.9</td>
<td>-sym</td>
<td>109</td>
</tr>
<tr>
<td>11.4.10</td>
<td>-unused</td>
<td>110</td>
</tr>
<tr>
<td>11.5</td>
<td>Library and Linking Command-Line Options</td>
<td>110</td>
</tr>
<tr>
<td>11.5.1</td>
<td>-codeaddr</td>
<td>111</td>
</tr>
<tr>
<td>11.5.2</td>
<td>-ConvertArchiveToPartialLink</td>
<td>112</td>
</tr>
<tr>
<td>11.5.3</td>
<td>-dataaddr</td>
<td>113</td>
</tr>
<tr>
<td>11.5.4</td>
<td>-genbinary</td>
<td>113</td>
</tr>
<tr>
<td>11.5.5</td>
<td>-gap_fill</td>
<td>114</td>
</tr>
<tr>
<td>11.5.6</td>
<td>-heapaddr</td>
<td>114</td>
</tr>
<tr>
<td>11.5.7</td>
<td>-heapsize</td>
<td>115</td>
</tr>
<tr>
<td>11.5.8</td>
<td>-lcf</td>
<td>115</td>
</tr>
<tr>
<td>11.5.9</td>
<td>-library</td>
<td>116</td>
</tr>
<tr>
<td>11.5.10</td>
<td>-linkmode</td>
<td>116</td>
</tr>
<tr>
<td>Section number</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>----------------</td>
<td>--------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>11.5.11</td>
<td>-main</td>
<td>117</td>
</tr>
<tr>
<td>11.5.12</td>
<td>-model</td>
<td>117</td>
</tr>
<tr>
<td>11.5.13</td>
<td>-noentry</td>
<td>118</td>
</tr>
<tr>
<td>11.5.14</td>
<td>-nomain</td>
<td>118</td>
</tr>
<tr>
<td>11.5.15</td>
<td>-opt_partial</td>
<td>119</td>
</tr>
<tr>
<td>11.5.16</td>
<td>-partial</td>
<td>119</td>
</tr>
<tr>
<td>11.5.17</td>
<td>-r</td>
<td>119</td>
</tr>
<tr>
<td>11.5.18</td>
<td>-r1</td>
<td>120</td>
</tr>
<tr>
<td>11.5.19</td>
<td>-r2</td>
<td>120</td>
</tr>
<tr>
<td>11.5.20</td>
<td>-relocate</td>
<td>121</td>
</tr>
<tr>
<td>11.5.21</td>
<td>-resolved_partial</td>
<td>121</td>
</tr>
<tr>
<td>11.5.22</td>
<td>-sdataaddr</td>
<td>121</td>
</tr>
<tr>
<td>11.5.23</td>
<td>-sdata2addr</td>
<td>122</td>
</tr>
<tr>
<td>11.5.24</td>
<td>-sdatathreshold</td>
<td>122</td>
</tr>
<tr>
<td>11.5.25</td>
<td>-sdata2threshold</td>
<td>123</td>
</tr>
<tr>
<td>11.5.26</td>
<td>-show</td>
<td>124</td>
</tr>
<tr>
<td>11.5.27</td>
<td>-srec</td>
<td>125</td>
</tr>
<tr>
<td>11.5.28</td>
<td>-sreceol</td>
<td>126</td>
</tr>
<tr>
<td>11.5.29</td>
<td>-sreclength</td>
<td>126</td>
</tr>
<tr>
<td>11.5.30</td>
<td>-stackaddr</td>
<td>127</td>
</tr>
<tr>
<td>11.5.31</td>
<td>-stacksize</td>
<td>127</td>
</tr>
<tr>
<td>11.5.32</td>
<td>-strip_partial</td>
<td>128</td>
</tr>
<tr>
<td>11.5.33</td>
<td>-tune_relocations</td>
<td>128</td>
</tr>
<tr>
<td>11.5.34</td>
<td>-xtables</td>
<td>129</td>
</tr>
<tr>
<td>11.5.35</td>
<td>-stdlib</td>
<td>129</td>
</tr>
<tr>
<td>11.5.36</td>
<td>-L+</td>
<td>129</td>
</tr>
<tr>
<td>11.5.37</td>
<td>-lr</td>
<td>130</td>
</tr>
<tr>
<td>11.5.38</td>
<td>-l+</td>
<td>130</td>
</tr>
<tr>
<td>11.6</td>
<td>Code Generation Command-Line Options</td>
<td>131</td>
</tr>
<tr>
<td>Section number</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>----------------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>11.6.1</td>
<td>abi</td>
<td>132</td>
</tr>
<tr>
<td>11.6.2</td>
<td>align</td>
<td>132</td>
</tr>
<tr>
<td>11.6.3</td>
<td>altivec_move_block</td>
<td>133</td>
</tr>
<tr>
<td>11.6.4</td>
<td>big</td>
<td>134</td>
</tr>
<tr>
<td>11.6.5</td>
<td>common</td>
<td>134</td>
</tr>
<tr>
<td>11.6.6</td>
<td>fatext</td>
<td>134</td>
</tr>
<tr>
<td>11.6.7</td>
<td>fp</td>
<td>135</td>
</tr>
<tr>
<td>11.6.8</td>
<td>fp_contract</td>
<td>137</td>
</tr>
<tr>
<td>11.6.9</td>
<td>func_align</td>
<td>137</td>
</tr>
<tr>
<td>11.6.10</td>
<td>gen-fsel</td>
<td>138</td>
</tr>
<tr>
<td>11.6.11</td>
<td>little</td>
<td>138</td>
</tr>
<tr>
<td>11.6.12</td>
<td>maf</td>
<td>138</td>
</tr>
<tr>
<td>11.6.13</td>
<td>ordered-fp-compares</td>
<td>139</td>
</tr>
<tr>
<td>11.6.14</td>
<td>pool</td>
<td>139</td>
</tr>
<tr>
<td>11.6.15</td>
<td>processor</td>
<td>140</td>
</tr>
<tr>
<td>11.6.16</td>
<td>profile</td>
<td>141</td>
</tr>
<tr>
<td>11.6.17</td>
<td>ppc_asm_to_vle</td>
<td>141</td>
</tr>
<tr>
<td>11.6.18</td>
<td>rambuffer</td>
<td>142</td>
</tr>
<tr>
<td>11.6.19</td>
<td>readonlystrings</td>
<td>143</td>
</tr>
<tr>
<td>11.6.20</td>
<td>relax_ieee</td>
<td>143</td>
</tr>
<tr>
<td>11.6.21</td>
<td>romaddr</td>
<td>143</td>
</tr>
<tr>
<td>11.6.22</td>
<td>rostr</td>
<td>143</td>
</tr>
<tr>
<td>11.6.23</td>
<td>schedule</td>
<td>144</td>
</tr>
<tr>
<td>11.6.24</td>
<td>spe_vector</td>
<td>144</td>
</tr>
<tr>
<td>11.6.25</td>
<td>spe2_vector</td>
<td>144</td>
</tr>
<tr>
<td>11.6.26</td>
<td>spe_addl_vector</td>
<td>145</td>
</tr>
<tr>
<td>11.6.27</td>
<td>strict_ieee</td>
<td>146</td>
</tr>
<tr>
<td>11.6.28</td>
<td>use_lmw_stmw</td>
<td>146</td>
</tr>
<tr>
<td>11.6.29</td>
<td>use_fsel</td>
<td>146</td>
</tr>
</tbody>
</table>
Chapter 12
Assembler

12.1 Syntax................................................................................................................................. 153
  12.1.1 Assembly Language Statements.................................................................................. 153
  12.1.2 Statement Syntax.......................................................................................................... 154
  12.1.3 Symbols......................................................................................................................... 155
    12.1.3.1 Labels.................................................................................................................... 155
    12.1.3.2 Non-Local Labels................................................................................................... 156
    12.1.3.3 Local Labels........................................................................................................... 156
    12.1.3.4 Relocatable Labels.................................................................................................. 157
    12.1.3.5 Equates.................................................................................................................. 158
    12.1.3.6 Case-Sensitive Identifiers...................................................................................... 160
  12.1.4 Constants....................................................................................................................... 160
    12.1.4.1 Integer Constants................................................................................................... 160
    12.1.4.2 Floating-Point Constants...................................................................................... 161
    12.1.4.3 Character Constants............................................................................................. 161
  12.1.5 Expressions................................................................................................................... 162
  12.1.6 Comments..................................................................................................................... 163
  12.1.7 Data Alignment............................................................................................................. 164

12.2 Directives.......................................................................................................................... 164
<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.2.1</td>
<td>Macro Directives</td>
<td>164</td>
</tr>
<tr>
<td>12.2.1.1</td>
<td>macro</td>
<td>165</td>
</tr>
<tr>
<td>12.2.1.2</td>
<td>endm</td>
<td>165</td>
</tr>
<tr>
<td>12.2.1.3</td>
<td>mexit</td>
<td>165</td>
</tr>
<tr>
<td>12.2.1.4</td>
<td>#define</td>
<td>166</td>
</tr>
<tr>
<td>12.2.2</td>
<td>Conditional Preprocessor Directives</td>
<td>167</td>
</tr>
<tr>
<td>12.2.2.1</td>
<td>if</td>
<td>167</td>
</tr>
<tr>
<td>12.2.2.2</td>
<td>ifdef</td>
<td>168</td>
</tr>
<tr>
<td>12.2.2.3</td>
<td>ifndef</td>
<td>168</td>
</tr>
<tr>
<td>12.2.2.4</td>
<td>ifc</td>
<td>169</td>
</tr>
<tr>
<td>12.2.2.5</td>
<td>ifnc</td>
<td>170</td>
</tr>
<tr>
<td>12.2.2.6</td>
<td>endif</td>
<td>170</td>
</tr>
<tr>
<td>12.2.2.7</td>
<td>elseif</td>
<td>170</td>
</tr>
<tr>
<td>12.2.2.8</td>
<td>else</td>
<td>172</td>
</tr>
<tr>
<td>12.2.3</td>
<td>Compatibility Conditional Directives</td>
<td>173</td>
</tr>
<tr>
<td>12.2.3.1</td>
<td>.ifeq if equal</td>
<td>173</td>
</tr>
<tr>
<td>12.2.3.2</td>
<td>.ifne if not equal</td>
<td>174</td>
</tr>
<tr>
<td>12.2.3.3</td>
<td>.iflt if less than</td>
<td>174</td>
</tr>
<tr>
<td>12.2.3.4</td>
<td>.ifle if less than or equal</td>
<td>174</td>
</tr>
<tr>
<td>12.2.3.5</td>
<td>.ifgt if greater than</td>
<td>175</td>
</tr>
<tr>
<td>12.2.3.6</td>
<td>.ifge if greater than or equal</td>
<td>175</td>
</tr>
<tr>
<td>12.2.4</td>
<td>Section Control Directives</td>
<td>176</td>
</tr>
<tr>
<td>12.2.4.1</td>
<td>text</td>
<td>176</td>
</tr>
<tr>
<td>12.2.4.2</td>
<td>data</td>
<td>177</td>
</tr>
<tr>
<td>12.2.4.3</td>
<td>rodata</td>
<td>177</td>
</tr>
<tr>
<td>12.2.4.4</td>
<td>bss</td>
<td>177</td>
</tr>
<tr>
<td>12.2.4.5</td>
<td>sdata</td>
<td>177</td>
</tr>
<tr>
<td>12.2.4.6</td>
<td>sdata2</td>
<td>177</td>
</tr>
<tr>
<td>12.2.4.7</td>
<td>sbss</td>
<td>178</td>
</tr>
<tr>
<td>Section number</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>----------------</td>
<td>--------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>12.2.4.8</td>
<td>text_vle</td>
<td>178</td>
</tr>
<tr>
<td>12.2.4.9</td>
<td>debug</td>
<td>178</td>
</tr>
<tr>
<td>12.2.4.10</td>
<td>previous</td>
<td>178</td>
</tr>
<tr>
<td>12.2.4.11</td>
<td>offset</td>
<td>179</td>
</tr>
<tr>
<td>12.2.4.12</td>
<td>section</td>
<td>180</td>
</tr>
<tr>
<td>12.2.5</td>
<td>Scope Control Directives</td>
<td>183</td>
</tr>
<tr>
<td>12.2.5.1</td>
<td>global</td>
<td>184</td>
</tr>
<tr>
<td>12.2.5.2</td>
<td>extern</td>
<td>184</td>
</tr>
<tr>
<td>12.2.5.3</td>
<td>public</td>
<td>184</td>
</tr>
<tr>
<td>12.2.6</td>
<td>Symbol Definition Directives</td>
<td>185</td>
</tr>
<tr>
<td>12.2.6.1</td>
<td>set</td>
<td>185</td>
</tr>
<tr>
<td>12.2.6.2</td>
<td>equal sign (=)</td>
<td>185</td>
</tr>
<tr>
<td>12.2.6.3</td>
<td>equ</td>
<td>186</td>
</tr>
<tr>
<td>12.2.6.4</td>
<td>textequ</td>
<td>186</td>
</tr>
<tr>
<td>12.2.7</td>
<td>Data Declaration Directives</td>
<td>187</td>
</tr>
<tr>
<td>12.2.7.1</td>
<td>Integer Directives</td>
<td>187</td>
</tr>
<tr>
<td>12.2.7.1.1</td>
<td>byte</td>
<td>188</td>
</tr>
<tr>
<td>12.2.7.1.2</td>
<td>fill</td>
<td>188</td>
</tr>
<tr>
<td>12.2.7.1.3</td>
<td>incbin</td>
<td>188</td>
</tr>
<tr>
<td>12.2.7.1.4</td>
<td>long</td>
<td>189</td>
</tr>
<tr>
<td>12.2.7.1.5</td>
<td>short</td>
<td>189</td>
</tr>
<tr>
<td>12.2.7.1.6</td>
<td>space</td>
<td>190</td>
</tr>
<tr>
<td>12.2.7.2</td>
<td>String Directives</td>
<td>190</td>
</tr>
<tr>
<td>12.2.7.2.1</td>
<td>ascii</td>
<td>191</td>
</tr>
<tr>
<td>12.2.7.2.2</td>
<td>asciz</td>
<td>191</td>
</tr>
<tr>
<td>12.2.7.3</td>
<td>Floating-Point Directives</td>
<td>192</td>
</tr>
<tr>
<td>12.2.7.3.1</td>
<td>float</td>
<td>192</td>
</tr>
<tr>
<td>12.2.7.3.2</td>
<td>double</td>
<td>192</td>
</tr>
<tr>
<td>12.2.8</td>
<td>Assembler Control Directives</td>
<td>193</td>
</tr>
<tr>
<td>Section number</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>12.2.8.1</td>
<td>align</td>
<td>193</td>
</tr>
<tr>
<td>12.2.8.2</td>
<td>endian</td>
<td>193</td>
</tr>
<tr>
<td>12.2.8.3</td>
<td>error</td>
<td>194</td>
</tr>
<tr>
<td>12.2.8.4</td>
<td>include</td>
<td>194</td>
</tr>
<tr>
<td>12.2.8.5</td>
<td>pragma</td>
<td>194</td>
</tr>
<tr>
<td>12.2.8.6</td>
<td>org</td>
<td>195</td>
</tr>
<tr>
<td>12.2.8.7</td>
<td>option</td>
<td>196</td>
</tr>
<tr>
<td>12.2.9</td>
<td>Debugging Directives</td>
<td>197</td>
</tr>
<tr>
<td>12.2.9.1</td>
<td>file</td>
<td>197</td>
</tr>
<tr>
<td>12.2.9.2</td>
<td>function</td>
<td>198</td>
</tr>
<tr>
<td>12.2.9.3</td>
<td>line</td>
<td>199</td>
</tr>
<tr>
<td>12.2.9.4</td>
<td>size</td>
<td>199</td>
</tr>
<tr>
<td>12.2.9.5</td>
<td>type</td>
<td>199</td>
</tr>
<tr>
<td>12.3</td>
<td>Macros</td>
<td>200</td>
</tr>
<tr>
<td>12.3.1</td>
<td>Defining Macros</td>
<td>200</td>
</tr>
<tr>
<td>12.3.1.1</td>
<td>Macro Definition Syntax</td>
<td>201</td>
</tr>
<tr>
<td>12.3.1.1.1</td>
<td>Using the .macro Directive</td>
<td>201</td>
</tr>
<tr>
<td>12.3.1.2</td>
<td>Using Macro Arguments</td>
<td>203</td>
</tr>
<tr>
<td>12.3.1.3</td>
<td>Creating Unique Labels and Equates</td>
<td>204</td>
</tr>
<tr>
<td>12.3.1.4</td>
<td>Referring to the Number of Arguments</td>
<td>205</td>
</tr>
<tr>
<td>12.3.2</td>
<td>Invoking Macros</td>
<td>205</td>
</tr>
<tr>
<td>12.3.2.1</td>
<td>Using the #define Directive</td>
<td>206</td>
</tr>
<tr>
<td>12.4</td>
<td>GNU Compatibility</td>
<td>207</td>
</tr>
<tr>
<td>12.4.1</td>
<td>GNU Compatible Syntax option</td>
<td>207</td>
</tr>
<tr>
<td>12.4.2</td>
<td>Supported Extensions</td>
<td>208</td>
</tr>
<tr>
<td>12.4.3</td>
<td>Unsupported Extensions</td>
<td>210</td>
</tr>
</tbody>
</table>

**Chapter 13**

**Linker**

13.1 Specifying Link Order in the IDE ................................................................. 213
<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>13.2</td>
<td>Dead-Stripping</td>
<td>214</td>
</tr>
<tr>
<td>13.3</td>
<td>Defining the Target's Memory Map</td>
<td>215</td>
</tr>
<tr>
<td>13.4</td>
<td>Defining Sections in the Output File</td>
<td>215</td>
</tr>
<tr>
<td>13.5</td>
<td>Associating Input Sections With Output Sections</td>
<td>216</td>
</tr>
<tr>
<td>13.6</td>
<td>Controlling Alignment</td>
<td>217</td>
</tr>
<tr>
<td>13.7</td>
<td>Specifying Memory Area Locations and Sizes</td>
<td>218</td>
</tr>
<tr>
<td>13.8</td>
<td>Creating Memory Gaps</td>
<td>219</td>
</tr>
<tr>
<td>13.9</td>
<td>Creating Symbols</td>
<td>219</td>
</tr>
<tr>
<td>13.10</td>
<td>Linker Command File Syntax</td>
<td>220</td>
</tr>
<tr>
<td>13.10.1</td>
<td>Linker Command File Operator Precedence and Support</td>
<td>224</td>
</tr>
<tr>
<td>13.11</td>
<td>Commands, Directives, and Keywords</td>
<td>224</td>
</tr>
<tr>
<td>13.11.1</td>
<td>(location counter)</td>
<td>225</td>
</tr>
<tr>
<td>13.11.2</td>
<td>ADDR</td>
<td>226</td>
</tr>
<tr>
<td>13.11.3</td>
<td>ALIGN</td>
<td>227</td>
</tr>
<tr>
<td>13.11.4</td>
<td>ALIGNALL</td>
<td>227</td>
</tr>
<tr>
<td>13.11.5</td>
<td>BIND</td>
<td>228</td>
</tr>
<tr>
<td>13.11.6</td>
<td>BIN_FILE_TYPE</td>
<td>229</td>
</tr>
<tr>
<td>13.11.7</td>
<td>BYTE</td>
<td>230</td>
</tr>
<tr>
<td>13.11.8</td>
<td>EXCEPTION</td>
<td>230</td>
</tr>
<tr>
<td>13.11.9</td>
<td>EXCLUDEFILES</td>
<td>231</td>
</tr>
<tr>
<td>13.11.10</td>
<td>EXPORTSTRTAB</td>
<td>231</td>
</tr>
<tr>
<td>13.11.11</td>
<td>EXPORTSYMTAB</td>
<td>232</td>
</tr>
<tr>
<td>13.11.12</td>
<td>EXTERNAL_SYMBOL</td>
<td>233</td>
</tr>
<tr>
<td>13.11.13</td>
<td>FORCEACTIVE</td>
<td>233</td>
</tr>
<tr>
<td>13.11.14</td>
<td>GROUP</td>
<td>234</td>
</tr>
<tr>
<td>13.11.15</td>
<td>IMPORTSTRTAB</td>
<td>235</td>
</tr>
<tr>
<td>13.11.16</td>
<td>IMPORTSYMTAB</td>
<td>235</td>
</tr>
<tr>
<td>13.11.17</td>
<td>INCLUDE</td>
<td>236</td>
</tr>
<tr>
<td>13.11.18</td>
<td>INCLUDEDWARF</td>
<td>237</td>
</tr>
</tbody>
</table>
Chapter 14
Linker for Power Architecture Processors

14.1 Predefined Sections ................................................................. 253
14.1.1 Linking Binary Files ............................................................ 254
14.1.1.1 Using CodeWarrior IDE .................................................. 254
14.1.1.2 Using Command-Line .................................................... 256
14.2 Additional Small Data Sections ............................................... 256
14.3 Linker Map File ....................................................................... 259
14.3.1 Closure ............................................................................. 259
14.3.2 Section Layout .................................................................... 261
<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>14.3.3</td>
<td>Memory Map</td>
<td>262</td>
</tr>
<tr>
<td>14.3.4</td>
<td>Linker Generated Symbols</td>
<td>263</td>
</tr>
<tr>
<td>14.4</td>
<td>Deadstripping</td>
<td>264</td>
</tr>
<tr>
<td>14.5</td>
<td>CodeWarrior Linker Command File (LCF)</td>
<td>264</td>
</tr>
<tr>
<td>14.6</td>
<td>Creating an LCF from Scratch</td>
<td>265</td>
</tr>
<tr>
<td>14.7</td>
<td>Relocating Code in ROM</td>
<td>267</td>
</tr>
<tr>
<td>14.7.1</td>
<td>Relocating Function in ROM</td>
<td>268</td>
</tr>
<tr>
<td>14.7.2</td>
<td>Placing Code in ROM</td>
<td>268</td>
</tr>
<tr>
<td>14.7.2.1</td>
<td>Create New ROM Segment</td>
<td>269</td>
</tr>
<tr>
<td>14.7.2.2</td>
<td>Create New ROM Section</td>
<td>269</td>
</tr>
<tr>
<td>14.8</td>
<td>Relocating Code and Data in Internal RAM</td>
<td>270</td>
</tr>
<tr>
<td>14.8.1</td>
<td>Placing Code and Data in RAM</td>
<td>271</td>
</tr>
<tr>
<td>14.8.1.1</td>
<td>Create New RAM Segment</td>
<td>271</td>
</tr>
<tr>
<td>14.8.1.2</td>
<td>Create New RAM Section</td>
<td>272</td>
</tr>
<tr>
<td>14.9</td>
<td>Relocating Code and Data in External MRAM</td>
<td>272</td>
</tr>
<tr>
<td>14.10</td>
<td>Unique LCF Examples</td>
<td>273</td>
</tr>
<tr>
<td>14.10.1</td>
<td>Configuring Linker File to Several ROM Blocks</td>
<td>273</td>
</tr>
<tr>
<td>14.10.2</td>
<td>Place the Library File in the LCF</td>
<td>273</td>
</tr>
<tr>
<td>14.10.3</td>
<td>Place Symbols in Specific Memory Location</td>
<td>274</td>
</tr>
<tr>
<td>14.10.4</td>
<td>How to Relocate Code in RAM</td>
<td>275</td>
</tr>
<tr>
<td>14.11</td>
<td>Linker Command File Commands</td>
<td>276</td>
</tr>
<tr>
<td>14.11.1</td>
<td>AGGRESSIVE_MERGE</td>
<td>276</td>
</tr>
<tr>
<td>14.11.2</td>
<td>AGGRESSIVE_MERGE_FILES</td>
<td>276</td>
</tr>
<tr>
<td>14.11.3</td>
<td>AGGRESSIVE_MERGE_SECTIONS</td>
<td>277</td>
</tr>
<tr>
<td>14.11.4</td>
<td>DO_NOT_MERGE</td>
<td>277</td>
</tr>
<tr>
<td>14.11.5</td>
<td>DO_NOT_MERGE_FILES</td>
<td>277</td>
</tr>
<tr>
<td>14.11.6</td>
<td>DO_NOT_MERGE_SECTIONS</td>
<td>278</td>
</tr>
<tr>
<td>14.11.7</td>
<td>INIT</td>
<td>278</td>
</tr>
<tr>
<td>14.11.8</td>
<td>FORCEFILES</td>
<td>279</td>
</tr>
<tr>
<td>Section number</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>----------------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>14.11.9</td>
<td>SHORTEN_NAMES_FOR_TOR_101</td>
<td>279</td>
</tr>
</tbody>
</table>

**Chapter 15**

**C Compiler**

15.1 Extensions to Standard C ......................................................... 281
   15.1.1 Controlling Standard C Conformance .................................. 281
   15.1.2 C++-style Comments .......................................................... 282
   15.1.3 Unnamed Arguments ......................................................... 282
   15.1.4 Extensions to the Preprocessor ......................................... 282
   15.1.5 Non-Standard Keywords ................................................... 283
   15.1.6 Declaring Variables by Address ....................................... 283

15.2 C99 Extensions ................................................................. 283
   15.2.1 Controlling C99 Extensions .............................................. 284
   15.2.2 Trailing Commas in Enumerations .................................... 284
   15.2.3 Compound Literal Values ............................................... 285
   15.2.4 Designated Initializers .................................................. 285
   15.2.5 Predefined Symbol __func__ ........................................... 286
   15.2.6 Implicit Return From main() .......................................... 286
   15.2.7 Non-constant Static Data Initialization ............................. 286
   15.2.8 Variable Argument Macros ............................................. 286
   15.2.9 Extra C99 Keywords ..................................................... 287
   15.2.10 C++-Style Comments .................................................... 287
   15.2.11 C++-Style Digraphs ..................................................... 288
   15.2.12 Empty Arrays in Structures ......................................... 288
   15.2.13 Hexadecimal Floating-Point Constants ................................ 288
   15.2.14 Variable-Length Arrays .............................................. 289
   15.2.15 Unsuffixed Decimal Literal Values .................................. 290
   15.2.16 C99 Complex Data Types ............................................. 290

15.3 GCC Extensions ............................................................... 291
   15.3.1 Controlling GCC Extensions ......................................... 291
<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>15.3.2</td>
<td>Initializing Automatic Arrays and Structures</td>
<td>291</td>
</tr>
<tr>
<td>15.3.3</td>
<td>The sizeof() Operator</td>
<td>292</td>
</tr>
<tr>
<td>15.3.4</td>
<td>Statements in Expressions</td>
<td>292</td>
</tr>
<tr>
<td>15.3.5</td>
<td>Redefining Macros</td>
<td>293</td>
</tr>
<tr>
<td>15.3.6</td>
<td>The typeof() Operator</td>
<td>293</td>
</tr>
<tr>
<td>15.3.7</td>
<td>Void and Function Pointer Arithmetic</td>
<td>293</td>
</tr>
<tr>
<td>15.3.8</td>
<td>The __builtin_constant_p() Operator</td>
<td>294</td>
</tr>
<tr>
<td>15.3.9</td>
<td>Forward Declarations of Static Arrays</td>
<td>294</td>
</tr>
<tr>
<td>15.3.10</td>
<td>Omitted Operands in Conditional Expressions</td>
<td>294</td>
</tr>
<tr>
<td>15.3.11</td>
<td>The __builtin_expect() Operator</td>
<td>295</td>
</tr>
<tr>
<td>15.3.12</td>
<td>Void Return Statements</td>
<td>295</td>
</tr>
<tr>
<td>15.3.13</td>
<td>Minimum and Maximum Operators</td>
<td>296</td>
</tr>
<tr>
<td>15.3.14</td>
<td>Local Labels</td>
<td>296</td>
</tr>
</tbody>
</table>

**Chapter 16**

**C++ Compiler**

16.1 C++ Compiler Performance........................................................................................................ 297
16.1.1 Precompiling C++ Source Code................................................................................................. 297
16.1.2 Using the Instance Manager.................................................................................................... 298

16.2 Extensions to Standard C++.......................................................................................................... 298
16.2.1 __PRETTY_FUNCTION__ Identifier............................................................................................... 298
16.2.2 Standard and Non-Standard Template Parsing............................................................................. 298

16.3 Implementation-Defined Behavior................................................................................................. 301
16.4 GCC Extensions............................................................................................................................... 303

**Chapter 17**

**Precompiling**

17.1 What Can be Precompiled.............................................................................................................. 305
17.2 Using a Precompiled File............................................................................................................... 306
17.3 Creating a Precompiled File........................................................................................................... 306
17.3.1 Precompiling a File in the CodeWarrior IDE........................................................................... 306
<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>17.3.2</td>
<td>Precompiling a File on the Command Line</td>
<td>307</td>
</tr>
<tr>
<td>17.3.3</td>
<td>Updating a Precompiled File Automatically</td>
<td>307</td>
</tr>
<tr>
<td>17.3.4</td>
<td>Preprocessor Scope in Precompiled Files</td>
<td>307</td>
</tr>
</tbody>
</table>

**Chapter 18**

Intermediate Optimizations

<table>
<thead>
<tr>
<th>18.1</th>
<th>Interprocedural Analysis</th>
<th>309</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.1.1</td>
<td>Invoking Interprocedural Analysis</td>
<td>309</td>
</tr>
<tr>
<td>18.1.2</td>
<td>Function-Level Optimization</td>
<td>310</td>
</tr>
<tr>
<td>18.1.3</td>
<td>File-Level Optimization</td>
<td>310</td>
</tr>
<tr>
<td>18.1.4</td>
<td>Program-Level Optimization</td>
<td>310</td>
</tr>
<tr>
<td>18.1.5</td>
<td>Program-Level Requirements</td>
<td>311</td>
</tr>
<tr>
<td>18.1.5.1</td>
<td>Dependencies Among Source Files</td>
<td>311</td>
</tr>
<tr>
<td>18.1.5.2</td>
<td>Function and Top-level Variable Declarations</td>
<td>311</td>
</tr>
<tr>
<td>18.1.5.3</td>
<td>Type Definitions</td>
<td>312</td>
</tr>
<tr>
<td>18.1.5.4</td>
<td>Unnamed Structures and Enumerations in C</td>
<td>313</td>
</tr>
<tr>
<td>18.2</td>
<td>Intermediate Optimizations</td>
<td>313</td>
</tr>
<tr>
<td>18.2.1</td>
<td>Dead Code Elimination</td>
<td>313</td>
</tr>
<tr>
<td>18.2.2</td>
<td>Expression Simplification</td>
<td>314</td>
</tr>
<tr>
<td>18.2.3</td>
<td>Common Subexpression Elimination</td>
<td>315</td>
</tr>
<tr>
<td>18.2.4</td>
<td>Copy Propagation</td>
<td>316</td>
</tr>
<tr>
<td>18.2.5</td>
<td>Dead Store Elimination</td>
<td>317</td>
</tr>
<tr>
<td>18.2.6</td>
<td>Live Range Splitting</td>
<td>318</td>
</tr>
<tr>
<td>18.2.7</td>
<td>Loop-Invariant Code Motion</td>
<td>320</td>
</tr>
<tr>
<td>18.2.8</td>
<td>Strength Reduction</td>
<td>321</td>
</tr>
<tr>
<td>18.2.9</td>
<td>Loop Unrolling</td>
<td>322</td>
</tr>
<tr>
<td>18.3</td>
<td>Inlining</td>
<td>323</td>
</tr>
<tr>
<td>18.3.1</td>
<td>Choosing Which Functions to Inline</td>
<td>323</td>
</tr>
<tr>
<td>18.3.2</td>
<td>Inlining Techniques</td>
<td>325</td>
</tr>
</tbody>
</table>

**Chapter 19**

CodeWarrior Development Studio for Power Architecture Processors Build Tools Reference Manual, Rev. 10.x, 06/2015

Freescale Semiconductor, Inc.
<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>19.1</td>
<td>Code Merging</td>
<td>327</td>
</tr>
<tr>
<td>20.1</td>
<td>Assembly Syntax</td>
<td>329</td>
</tr>
<tr>
<td>20.1.1</td>
<td>Specifying Inline Assembly Statements</td>
<td>330</td>
</tr>
<tr>
<td>20.1.2</td>
<td>Function-Level Inline Assembly</td>
<td>330</td>
</tr>
<tr>
<td>20.1.3</td>
<td>Statement-Level Inline Assembly</td>
<td>331</td>
</tr>
<tr>
<td>20.1.4</td>
<td>GCC - Style Inline Assembly</td>
<td>331</td>
</tr>
<tr>
<td>20.1.5</td>
<td>Branch Prediction</td>
<td>332</td>
</tr>
<tr>
<td>20.1.6</td>
<td>PC-Relative Addressing</td>
<td>332</td>
</tr>
<tr>
<td>20.1.7</td>
<td>Normal, Record, and Overflow Forms</td>
<td>333</td>
</tr>
<tr>
<td>20.1.8</td>
<td>Creating Statement Labels</td>
<td>333</td>
</tr>
<tr>
<td>20.1.9</td>
<td>Using Comments</td>
<td>334</td>
</tr>
<tr>
<td>20.1.10</td>
<td>Using the Preprocessor</td>
<td>335</td>
</tr>
<tr>
<td>20.2</td>
<td>Referring to Assembly, C, and C++ Variables</td>
<td>335</td>
</tr>
<tr>
<td>20.2.1</td>
<td>Using Local Variables and Arguments</td>
<td>335</td>
</tr>
<tr>
<td>20.2.2</td>
<td>Creating a Stack Frame</td>
<td>336</td>
</tr>
<tr>
<td>20.2.3</td>
<td>Referring to Variables in Instruction Operands</td>
<td>337</td>
</tr>
<tr>
<td>20.2.4</td>
<td>Referring to Variables in Memory</td>
<td>337</td>
</tr>
<tr>
<td>20.2.5</td>
<td>Referring to Registers</td>
<td>338</td>
</tr>
<tr>
<td>20.2.6</td>
<td>Referring to Labels</td>
<td>339</td>
</tr>
<tr>
<td>20.2.7</td>
<td>Using Variable Names as Memory Locations</td>
<td>339</td>
</tr>
<tr>
<td>20.2.8</td>
<td>Using Immediate Operands</td>
<td>340</td>
</tr>
<tr>
<td>20.3</td>
<td>Assembler Directives</td>
<td>341</td>
</tr>
<tr>
<td>20.3.1</td>
<td>entry</td>
<td>342</td>
</tr>
<tr>
<td>20.3.2</td>
<td>fralloc</td>
<td>342</td>
</tr>
<tr>
<td>20.3.3</td>
<td>frfree</td>
<td>343</td>
</tr>
<tr>
<td>20.3.4</td>
<td>machine</td>
<td>343</td>
</tr>
<tr>
<td>Section number</td>
<td>Title</td>
<td></td>
</tr>
<tr>
<td>----------------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td>20.3.5</td>
<td>nofralloc</td>
<td></td>
</tr>
<tr>
<td>20.3.6</td>
<td>opword</td>
<td></td>
</tr>
<tr>
<td>20.3.7</td>
<td>equ</td>
<td></td>
</tr>
<tr>
<td>20.4</td>
<td>Intrinsic Functions</td>
<td></td>
</tr>
<tr>
<td>20.4.1</td>
<td>Low-Level Processor Synchronization</td>
<td></td>
</tr>
<tr>
<td>20.4.2</td>
<td>Absolute Value Functions</td>
<td></td>
</tr>
<tr>
<td>20.4.3</td>
<td>Byte-Reversing Functions</td>
<td></td>
</tr>
<tr>
<td>20.4.4</td>
<td>Setting the Floating-Point Environment</td>
<td></td>
</tr>
<tr>
<td>20.4.5</td>
<td>Manipulating the Contents of a Variable or Register</td>
<td></td>
</tr>
<tr>
<td>20.4.6</td>
<td>Data Cache Manipulation</td>
<td></td>
</tr>
<tr>
<td>20.4.7</td>
<td>Math Functions</td>
<td></td>
</tr>
<tr>
<td>20.4.8</td>
<td>Buffer Manipulation</td>
<td></td>
</tr>
</tbody>
</table>

**Chapter 21**

**Power Architecture Code Generation**

<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>21.1</td>
<td>ABI Conformance</td>
</tr>
<tr>
<td>21.2</td>
<td>Data Representation</td>
</tr>
<tr>
<td>21.2.1</td>
<td>Boolean Type</td>
</tr>
<tr>
<td>21.2.2</td>
<td>Character Types</td>
</tr>
<tr>
<td>21.2.3</td>
<td>Integer Types</td>
</tr>
<tr>
<td>21.2.4</td>
<td>Floating-Point</td>
</tr>
<tr>
<td>21.2.5</td>
<td>Altivec™ Data Types</td>
</tr>
<tr>
<td>21.3</td>
<td>Data Addressing</td>
</tr>
<tr>
<td>21.4</td>
<td>Aligning Data</td>
</tr>
<tr>
<td>21.4.1</td>
<td>Alignment Attribute Syntax</td>
</tr>
<tr>
<td>21.4.2</td>
<td>Aligning a Variable Declaration</td>
</tr>
<tr>
<td>21.4.3</td>
<td>Alignment in a Structure Definition</td>
</tr>
<tr>
<td>21.4.4</td>
<td>Structure Member</td>
</tr>
<tr>
<td>21.4.5</td>
<td>Typedef Declaration</td>
</tr>
<tr>
<td>21.4.6</td>
<td>Bitfields</td>
</tr>
<tr>
<td>Section number</td>
<td>Title</td>
</tr>
<tr>
<td>----------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>21.5</td>
<td>Small Data Area PIC/PID Support</td>
</tr>
<tr>
<td>21.5.1</td>
<td>Internal and External Segments and References</td>
</tr>
<tr>
<td>21.5.2</td>
<td>PIC/PID Linker Command File Directives</td>
</tr>
<tr>
<td>21.5.3</td>
<td>Linker-defined Symbols</td>
</tr>
<tr>
<td>21.5.4</td>
<td>Uses for SDA PIC/PID</td>
</tr>
<tr>
<td>21.5.5</td>
<td>Building an SDA PIC/PID Application</td>
</tr>
<tr>
<td>21.5.5.1</td>
<td>Linking Assembly Files</td>
</tr>
<tr>
<td>21.5.5.2</td>
<td>Modifications to the Section Pragma</td>
</tr>
<tr>
<td>21.5.6</td>
<td>Internal and External Addressing Modes</td>
</tr>
<tr>
<td>21.5.6.1</td>
<td>Specifying ROM addr_mode</td>
</tr>
<tr>
<td>21.5.6.2</td>
<td>Specifying RAM addr_mode</td>
</tr>
<tr>
<td>21.6</td>
<td>Variable Length Encoding</td>
</tr>
<tr>
<td>21.6.1</td>
<td>Processors With VLE Capability</td>
</tr>
<tr>
<td>21.6.2</td>
<td>Compiling VLE Instructions</td>
</tr>
<tr>
<td>21.6.3</td>
<td>Assembling VLE Instructions</td>
</tr>
<tr>
<td>21.6.4</td>
<td>Linking VLE Object Code</td>
</tr>
<tr>
<td>21.7</td>
<td>Building a ROM Image</td>
</tr>
<tr>
<td>21.7.1</td>
<td>Linking a ROM Image</td>
</tr>
<tr>
<td>21.7.2</td>
<td>ROM Image Addresses</td>
</tr>
<tr>
<td>21.7.3</td>
<td>Specifying A Single ROM Block</td>
</tr>
<tr>
<td>21.7.4</td>
<td>Specifying Several ROM Blocks</td>
</tr>
<tr>
<td>21.7.5</td>
<td>Specifying Jump Table Location</td>
</tr>
<tr>
<td>21.7.6</td>
<td>Specifying Constant Data Location</td>
</tr>
<tr>
<td>21.8</td>
<td>Embedded C++</td>
</tr>
<tr>
<td>21.8.1</td>
<td>Activating EC++</td>
</tr>
<tr>
<td>21.8.2</td>
<td>Differences Between ISO C++ and EC++</td>
</tr>
<tr>
<td>21.8.2.1</td>
<td>Templates</td>
</tr>
<tr>
<td>21.8.2.2</td>
<td>Libraries</td>
</tr>
<tr>
<td>21.8.2.3</td>
<td>File Operations</td>
</tr>
</tbody>
</table>
Chapter 22
Libraries and Support Code

22.1 Main Standard Libraries................................................................. 377
  22.1.1 Using the Main Standard Libraries........................................... 377
  22.1.2 Choosing an MSL Library......................................................... 378
  22.1.3 Using Console I/O................................................................. 379
  22.1.4 Allocating Additional Heap Space............................................ 380

22.2 Third Party Standard Libraries.................................................. 380

22.3 Embedded Warrior Library......................................................... 380
  22.3.1 Using the Embedded Warrior Libraries................................. 381
  22.3.2 EWL Naming Convention..................................................... 381
  22.3.3 How to Rebuild the EWL Libraries......................................... 383

22.4 Runtime Libraries................................................................. 384
  22.4.1 Required Libraries and Source Code Files.............................. 385
  22.4.2 Allocating Additional Heap Space........................................... 386
  22.4.3 Choosing a Runtime Library.................................................. 386

22.5 Board Initialization Code.......................................................... 388

Chapter 23
Declaration Specifications

23.1 Syntax for Declaration Specifications......................................... 391
  23.1.1 __declspec(never_inline)....................................................... 391

23.2 Syntax for Attribute Specifications............................................. 392
  23.2.1 __attribute__((noreturn))....................................................... 392
Chapter 24
Declaration Specifications for Power Architecture Code

24.1 Syntax for Declaration Specifications..........................................................................................................................399
24.2 Declaration Specifications ..................................................................................................................................................399
  24.2.1 __declspec(do_not_merge)......................................................................................................................................400
  24.2.2 __declspec(final)...............................................................................................................................................400
  24.2.3 __declspec(force_export)...................................................................................................................................401
  24.2.4 __declspec(interrupt)........................................................................................................................................401
  24.2.5 __declspec(no_linker_opts)..............................................................................................................................403
  24.2.6 __declspec(section name)....................................................................................................................................403
  24.2.7 __declspec(vle_off).........................................................................................................................................405
  24.2.8 __declspec(vle_on)........................................................................................................................................405
  24.2.9 __declspec(weak)...........................................................................................................................................406

Chapter 25
Predefined Macros

25.1 __ASSEMBLER__............................................................................................................................................................407
25.2 __COMPILER__...............................................................................................................................................................408
25.3 __COUNTER__.................................................................................................................................................................408
25.4 __TARGETARCH__..........................................................................................................................................................409
25.5 __VERSION_NUM__......................................................................................................................................................409
25.6 __VERSION__...............................................................................................................................................................409
<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>25.7</td>
<td>__cplusplus</td>
<td>410</td>
</tr>
<tr>
<td>25.8</td>
<td><strong>CWBUILD</strong></td>
<td>410</td>
</tr>
<tr>
<td>25.9</td>
<td><strong>CWCC</strong></td>
<td>410</td>
</tr>
<tr>
<td>25.10</td>
<td><strong>DATE</strong></td>
<td>411</td>
</tr>
<tr>
<td>25.11</td>
<td>__embedded_cplusplus</td>
<td>411</td>
</tr>
<tr>
<td>25.12</td>
<td><strong>FILE</strong></td>
<td>412</td>
</tr>
<tr>
<td>25.13</td>
<td><strong>func</strong></td>
<td>412</td>
</tr>
<tr>
<td>25.14</td>
<td><strong>FUNCTION</strong></td>
<td>412</td>
</tr>
<tr>
<td>25.15</td>
<td>__ide_target()</td>
<td>413</td>
</tr>
<tr>
<td>25.16</td>
<td><strong>LINE</strong></td>
<td>413</td>
</tr>
<tr>
<td>25.17</td>
<td><strong>MWERKS</strong></td>
<td>414</td>
</tr>
<tr>
<td>25.18</td>
<td>__optlevel</td>
<td>414</td>
</tr>
<tr>
<td>25.19</td>
<td><strong>PRETTY_FUNCTION</strong></td>
<td>415</td>
</tr>
<tr>
<td>25.20</td>
<td><strong>profile</strong></td>
<td>416</td>
</tr>
<tr>
<td>25.21</td>
<td><strong>STDC</strong></td>
<td>416</td>
</tr>
<tr>
<td>25.22</td>
<td><strong>TIME</strong></td>
<td>416</td>
</tr>
</tbody>
</table>

Chapter 26
Predefined Macros for Power Architecture Compiler

<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>26.1</td>
<td><strong>ALTIVEC</strong></td>
<td>419</td>
</tr>
<tr>
<td>26.2</td>
<td><strong>POWERPC</strong></td>
<td>419</td>
</tr>
<tr>
<td>26.3</td>
<td><strong>PPCBROADWAY</strong></td>
<td>420</td>
</tr>
<tr>
<td>26.4</td>
<td><strong>PPCGECKO</strong></td>
<td>420</td>
</tr>
<tr>
<td>26.5</td>
<td>Processor Core Specific Macros</td>
<td>420</td>
</tr>
</tbody>
</table>

Chapter 27
Using Pragmas

<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>27.1</td>
<td>Checking Pragma Settings</td>
<td>423</td>
</tr>
<tr>
<td>27.2</td>
<td>Saving and Restoring Pragma Settings</td>
<td>424</td>
</tr>
<tr>
<td>27.3</td>
<td>Determining Which Settings Are Saved and Restored</td>
<td>425</td>
</tr>
<tr>
<td>27.4</td>
<td>Invalid Pragmas</td>
<td>425</td>
</tr>
<tr>
<td>Section number</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>----------------</td>
<td>-----------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>27.5</td>
<td>Pragma Scope</td>
<td>426</td>
</tr>
<tr>
<td>28.1</td>
<td>ANSI_strict</td>
<td>427</td>
</tr>
<tr>
<td>28.2</td>
<td>c99</td>
<td>427</td>
</tr>
<tr>
<td>28.3</td>
<td>c9x</td>
<td>428</td>
</tr>
<tr>
<td>28.4</td>
<td>ignore_oldstyle</td>
<td>428</td>
</tr>
<tr>
<td>28.5</td>
<td>only_std_keywords</td>
<td>429</td>
</tr>
<tr>
<td>28.6</td>
<td>require_prototypes</td>
<td>430</td>
</tr>
<tr>
<td>29.1</td>
<td>access_errors</td>
<td>433</td>
</tr>
<tr>
<td>29.2</td>
<td>always_inline</td>
<td>433</td>
</tr>
<tr>
<td>29.3</td>
<td>arg_dep_lookup</td>
<td>434</td>
</tr>
<tr>
<td>29.4</td>
<td>ARM_conform</td>
<td>434</td>
</tr>
<tr>
<td>29.5</td>
<td>ARM_scoping</td>
<td>434</td>
</tr>
<tr>
<td>29.6</td>
<td>array_new_delete</td>
<td>435</td>
</tr>
<tr>
<td>29.7</td>
<td>auto_inline</td>
<td>435</td>
</tr>
<tr>
<td>29.8</td>
<td>bool</td>
<td>435</td>
</tr>
<tr>
<td>29.9</td>
<td>cplusplus</td>
<td>436</td>
</tr>
<tr>
<td>29.10</td>
<td>cpp1x</td>
<td>437</td>
</tr>
<tr>
<td>29.11</td>
<td>cpp_extensions</td>
<td>437</td>
</tr>
<tr>
<td>29.12</td>
<td>debuginline</td>
<td>438</td>
</tr>
<tr>
<td>29.13</td>
<td>def_inherited</td>
<td>439</td>
</tr>
<tr>
<td>29.14</td>
<td>defer_codegen</td>
<td>439</td>
</tr>
<tr>
<td>29.15</td>
<td>defer_defarg_parsing</td>
<td>439</td>
</tr>
<tr>
<td>29.16</td>
<td>direct_destruction</td>
<td>440</td>
</tr>
<tr>
<td>29.17</td>
<td>direct_to_kom</td>
<td>440</td>
</tr>
<tr>
<td>29.18</td>
<td>dont_inline</td>
<td>440</td>
</tr>
<tr>
<td>Section number</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>----------------</td>
<td>------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>29.19</td>
<td>ecplusplus ....................................................................</td>
<td>441</td>
</tr>
<tr>
<td>29.20</td>
<td>exceptions ..................................................................</td>
<td>441</td>
</tr>
<tr>
<td>29.21</td>
<td>inline_bottom_up ..................................................</td>
<td>442</td>
</tr>
<tr>
<td>29.22</td>
<td>inline_bottom_up_once ............................................</td>
<td>443</td>
</tr>
<tr>
<td>29.23</td>
<td>inline_depth ................................................................</td>
<td>443</td>
</tr>
<tr>
<td>29.24</td>
<td>inline_max_auto_size ...............................................</td>
<td>444</td>
</tr>
<tr>
<td>29.25</td>
<td>inline_max_size ......................................................</td>
<td>445</td>
</tr>
<tr>
<td>29.26</td>
<td>inline_max_total_size ...............................................</td>
<td>445</td>
</tr>
<tr>
<td>29.27</td>
<td>internal ....................................................................</td>
<td>445</td>
</tr>
<tr>
<td>29.28</td>
<td>iso_templates ................................................................</td>
<td>446</td>
</tr>
<tr>
<td>29.29</td>
<td>new_mangler ..................................................................</td>
<td>447</td>
</tr>
<tr>
<td>29.30</td>
<td>no_conststringconv ...................................................</td>
<td>447</td>
</tr>
<tr>
<td>29.31</td>
<td>no_static_dtors ..........................................................</td>
<td>447</td>
</tr>
<tr>
<td>29.32</td>
<td>nosyminline ..................................................................</td>
<td>448</td>
</tr>
<tr>
<td>29.33</td>
<td>old_friend_lookup .....................................................</td>
<td>448</td>
</tr>
<tr>
<td>29.34</td>
<td>old_pods .....................................................................</td>
<td>449</td>
</tr>
<tr>
<td>29.35</td>
<td>old_vtable ..................................................................</td>
<td>449</td>
</tr>
<tr>
<td>29.36</td>
<td>opt_classresults ......................................................</td>
<td>450</td>
</tr>
<tr>
<td>29.37</td>
<td>parse_func_templ ........................................................</td>
<td>450</td>
</tr>
<tr>
<td>29.38</td>
<td>parse_mfunc_templ ......................................................</td>
<td>451</td>
</tr>
<tr>
<td>29.39</td>
<td>RTTI .............................................................................</td>
<td>451</td>
</tr>
<tr>
<td>29.40</td>
<td>suppress_init_code ....................................................</td>
<td>452</td>
</tr>
<tr>
<td>29.41</td>
<td>template_depth ................................................................</td>
<td>452</td>
</tr>
<tr>
<td>29.42</td>
<td>thread_safe_init ..........................................................</td>
<td>452</td>
</tr>
<tr>
<td>29.43</td>
<td>warn_hidevirtual ........................................................</td>
<td>454</td>
</tr>
<tr>
<td>29.44</td>
<td>warn_no_explicit_virtual ............................................</td>
<td>454</td>
</tr>
<tr>
<td>29.45</td>
<td>warn_no_typename ........................................................</td>
<td>455</td>
</tr>
<tr>
<td>29.46</td>
<td>warn_notinlined ..........................................................</td>
<td>456</td>
</tr>
<tr>
<td>29.47</td>
<td>warn_structclass ........................................................</td>
<td>456</td>
</tr>
</tbody>
</table>
Chapter 30
Pragmas for Language Translation

30.1 asmpoundcomment
30.2 asmsemicolcomment
30.3 const_strings
30.4 dollar_identifiers
30.5 gcc_extensions
30.6 mark
30.7 mpwc_newline
30.8 mpwc_relax
30.9 multibyteaware
30.10 multibyteaware_preserve_literals
30.11 text_encoding
30.12 trigraphs
30.13 unsigned_char

Chapter 31
Pragmas for Diagnostic Messages

31.1 extended_errorcheck
31.2 maxerrorcount
31.3 message
31.4 showmessage_number
31.5 show_error_filestack
31.6 suppress_warnings
31.7 sym
31.8 unused
31.9 warning
31.10 warning_errors
31.11 warn_any_ptr_int_conv
<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>31.12</td>
<td>warn_emptydecl</td>
<td>474</td>
</tr>
<tr>
<td>31.13</td>
<td>warn_extracomma</td>
<td>474</td>
</tr>
<tr>
<td>31.14</td>
<td>warn_filenamecaps</td>
<td>475</td>
</tr>
<tr>
<td>31.15</td>
<td>warn_filenamecaps_system</td>
<td>476</td>
</tr>
<tr>
<td>31.16</td>
<td>warn_hiddenlocals</td>
<td>476</td>
</tr>
<tr>
<td>31.17</td>
<td>warn_illpragma</td>
<td>477</td>
</tr>
<tr>
<td>31.18</td>
<td>warn_illtokenpasting</td>
<td>477</td>
</tr>
<tr>
<td>31.19</td>
<td>warn_illusionmembers</td>
<td>478</td>
</tr>
<tr>
<td>31.20</td>
<td>warn_impl_f2i_conv</td>
<td>478</td>
</tr>
<tr>
<td>31.21</td>
<td>warn_impl_i2f_conv</td>
<td>479</td>
</tr>
<tr>
<td>31.22</td>
<td>warn_impl_s2u_conv</td>
<td>479</td>
</tr>
<tr>
<td>31.23</td>
<td>warn_implicitconv</td>
<td>480</td>
</tr>
<tr>
<td>31.24</td>
<td>warn_largeargs</td>
<td>481</td>
</tr>
<tr>
<td>31.25</td>
<td>warn_missingreturn</td>
<td>481</td>
</tr>
<tr>
<td>31.26</td>
<td>warn_no_side_effect</td>
<td>482</td>
</tr>
<tr>
<td>31.27</td>
<td>warn_padding</td>
<td>482</td>
</tr>
<tr>
<td>31.28</td>
<td>warn_pch_portability</td>
<td>483</td>
</tr>
<tr>
<td>31.29</td>
<td>warn_possunwant</td>
<td>483</td>
</tr>
<tr>
<td>31.30</td>
<td>warn_ptr_int_conv</td>
<td>484</td>
</tr>
<tr>
<td>31.31</td>
<td>warn_resultnotused</td>
<td>485</td>
</tr>
<tr>
<td>31.32</td>
<td>warn.Undefmacro</td>
<td>485</td>
</tr>
<tr>
<td>31.33</td>
<td>warn_uninitializedvar</td>
<td>486</td>
</tr>
<tr>
<td>31.34</td>
<td>warn_unusedarg</td>
<td>486</td>
</tr>
<tr>
<td>31.35</td>
<td>warn_unusedvar</td>
<td>487</td>
</tr>
</tbody>
</table>

**Chapter 32**  
**Pragmas for Preprocessing**

<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>32.1</td>
<td>check_header_flags</td>
<td>489</td>
</tr>
<tr>
<td>32.2</td>
<td>faster_pch_gen</td>
<td>489</td>
</tr>
<tr>
<td>32.3</td>
<td>flat_include</td>
<td>490</td>
</tr>
</tbody>
</table>

CodeWarrior Development Studio for Power Architecture Processors Build Tools Reference Manual, Rev. 10.x, 06/2015
Freescale Semiconductor, Inc.  29
<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>32.4</td>
<td>fullpath_file</td>
<td>490</td>
</tr>
<tr>
<td>32.5</td>
<td>fullpath_prepdump</td>
<td>491</td>
</tr>
<tr>
<td>32.6</td>
<td>keepcomments</td>
<td>491</td>
</tr>
<tr>
<td>32.7</td>
<td>line_prepdump</td>
<td>491</td>
</tr>
<tr>
<td>32.8</td>
<td>macro_prepdump</td>
<td>492</td>
</tr>
<tr>
<td>32.9</td>
<td>msg_show_liner</td>
<td>492</td>
</tr>
<tr>
<td>32.10</td>
<td>msg_show_realf</td>
<td>492</td>
</tr>
<tr>
<td>32.11</td>
<td>notonce</td>
<td>493</td>
</tr>
<tr>
<td>32.12</td>
<td>old pragma_once</td>
<td>493</td>
</tr>
<tr>
<td>32.13</td>
<td>once</td>
<td>493</td>
</tr>
<tr>
<td>32.14</td>
<td>pop, push</td>
<td>494</td>
</tr>
<tr>
<td>32.15</td>
<td>pragma_prepdump</td>
<td>495</td>
</tr>
<tr>
<td>32.16</td>
<td>precompile_target</td>
<td>495</td>
</tr>
<tr>
<td>32.17</td>
<td>simple_prepdump</td>
<td>496</td>
</tr>
<tr>
<td>32.18</td>
<td>space_prepdump</td>
<td>496</td>
</tr>
<tr>
<td>32.19</td>
<td>srcrelincludes</td>
<td>497</td>
</tr>
<tr>
<td>32.20</td>
<td>syspath_once</td>
<td>497</td>
</tr>
</tbody>
</table>

Chapter 33  
Pragmas for Library and Linking

<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>33.1</td>
<td>always_import</td>
<td>499</td>
</tr>
<tr>
<td>33.2</td>
<td>export</td>
<td>499</td>
</tr>
<tr>
<td>33.3</td>
<td>import</td>
<td>500</td>
</tr>
<tr>
<td>33.4</td>
<td>lib_export</td>
<td>501</td>
</tr>
</tbody>
</table>

Chapter 34  
Pragmas for Code Generation

<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>34.1</td>
<td>aggressive_inline</td>
<td>503</td>
</tr>
<tr>
<td>34.2</td>
<td>dont_reuse_strings</td>
<td>503</td>
</tr>
<tr>
<td>34.3</td>
<td>enumsalwaysint</td>
<td>504</td>
</tr>
<tr>
<td>34.4</td>
<td>enums_signed</td>
<td>505</td>
</tr>
<tr>
<td>Section number</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>----------------</td>
<td>--------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>34.5</td>
<td>errno_name</td>
<td>506</td>
</tr>
<tr>
<td>34.6</td>
<td>explicit_zero_data</td>
<td>506</td>
</tr>
<tr>
<td>34.7</td>
<td>float_constants</td>
<td>507</td>
</tr>
<tr>
<td>34.8</td>
<td>instmgr_file</td>
<td>507</td>
</tr>
<tr>
<td>34.9</td>
<td>longlong</td>
<td>508</td>
</tr>
<tr>
<td>34.10</td>
<td>longlong Enums</td>
<td>508</td>
</tr>
<tr>
<td>34.11</td>
<td>min_enum_size</td>
<td>509</td>
</tr>
<tr>
<td>34.12</td>
<td>pool_strings</td>
<td>509</td>
</tr>
<tr>
<td>34.13</td>
<td>readonly_strings</td>
<td>510</td>
</tr>
<tr>
<td>34.14</td>
<td>reverse_bitfields</td>
<td>510</td>
</tr>
<tr>
<td>34.15</td>
<td>store_object_files</td>
<td>511</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35.1</td>
<td>global_optimizer</td>
<td>513</td>
</tr>
<tr>
<td>35.2</td>
<td>ipa</td>
<td>514</td>
</tr>
<tr>
<td>35.3</td>
<td>ipa_inline_max_auto_size</td>
<td>514</td>
</tr>
<tr>
<td>35.4</td>
<td>ipa_not_complete</td>
<td>515</td>
</tr>
<tr>
<td>35.5</td>
<td>load_store_elimination</td>
<td>516</td>
</tr>
<tr>
<td>35.6</td>
<td>opt_common_subs</td>
<td>516</td>
</tr>
<tr>
<td>35.7</td>
<td>opt_dead_assignments</td>
<td>516</td>
</tr>
<tr>
<td>35.8</td>
<td>opt_dead_code</td>
<td>517</td>
</tr>
<tr>
<td>35.9</td>
<td>opt_lifetimes</td>
<td>517</td>
</tr>
<tr>
<td>35.10</td>
<td>opt_loop_invariants</td>
<td>518</td>
</tr>
<tr>
<td>35.11</td>
<td>opt_propagation</td>
<td>518</td>
</tr>
<tr>
<td>35.12</td>
<td>opt_strength_reduction</td>
<td>518</td>
</tr>
<tr>
<td>35.13</td>
<td>opt_strength_reduction_strict</td>
<td>519</td>
</tr>
<tr>
<td>35.14</td>
<td>opt_unroll_loops</td>
<td>519</td>
</tr>
<tr>
<td>35.15</td>
<td>opt_vectorize_loops</td>
<td>520</td>
</tr>
<tr>
<td>35.16</td>
<td>optimization_level</td>
<td>520</td>
</tr>
</tbody>
</table>

Chapter 35
Pragmas for Optimization

35.1 global_optimizer.................................................................................................................. 513
35.2 ipa............................................................................................................................................. 514
35.3 ipa_inline_max_auto_size....................................................................................................... 514
35.4 ipa_not_complete.................................................................................................................... 515
35.5 load_store_elimination........................................................................................................... 516
35.6 opt_common_subs.................................................................................................................... 516
35.7 opt_dead_assignments............................................................................................................. 516
35.8 opt_dead_code......................................................................................................................... 517
35.9 opt_lifetimes.......................................................................................................................... 517
35.10 opt_loop_invariants............................................................................................................... 518
35.11 opt_propagation.................................................................................................................... 518
35.12 opt_strength_reduction......................................................................................................... 518
35.13 opt_strength_reduction_strict............................................................................................... 519
35.14 opt_unroll_loops.................................................................................................................... 519
35.15 opt_vectorize_loops............................................................................................................... 520
35.16 optimization_level............................................................................................................... 520
<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>35.17</td>
<td>optimize_for_size</td>
<td>521</td>
</tr>
<tr>
<td>35.18</td>
<td>optimizewithasm</td>
<td>521</td>
</tr>
<tr>
<td>35.19</td>
<td>pack</td>
<td>521</td>
</tr>
<tr>
<td>35.20</td>
<td>strictheaderchecking</td>
<td>522</td>
</tr>
<tr>
<td>36.1</td>
<td>Diagnostic Pragmas</td>
<td>525</td>
</tr>
<tr>
<td>36.1.1</td>
<td>incompatible_return_small_structs</td>
<td>525</td>
</tr>
<tr>
<td>36.1.2</td>
<td>incompatible_sfpe_double_params</td>
<td>526</td>
</tr>
<tr>
<td>36.2</td>
<td>Debugging Information Pragmas</td>
<td>526</td>
</tr>
<tr>
<td>36.2.1</td>
<td>dwarf2typedefchains</td>
<td>527</td>
</tr>
<tr>
<td>36.2.2</td>
<td>dwarf2lexblockcodeaddrcalc</td>
<td>527</td>
</tr>
<tr>
<td>36.3</td>
<td>Library and Linking Pragmas</td>
<td>527</td>
</tr>
<tr>
<td>36.3.1</td>
<td>force_active</td>
<td>528</td>
</tr>
<tr>
<td>36.3.2</td>
<td>section</td>
<td>528</td>
</tr>
<tr>
<td>36.4</td>
<td>Code generation Pragmas</td>
<td>534</td>
</tr>
<tr>
<td>36.4.1</td>
<td>alignment_metrowerks</td>
<td>535</td>
</tr>
<tr>
<td>36.4.2</td>
<td>altivec_codegen</td>
<td>535</td>
</tr>
<tr>
<td>36.4.3</td>
<td>altivec_model</td>
<td>535</td>
</tr>
<tr>
<td>36.4.4</td>
<td>altivec_pim_warnings</td>
<td>536</td>
</tr>
<tr>
<td>36.4.5</td>
<td>altivec_vrsave</td>
<td>537</td>
</tr>
<tr>
<td>36.4.6</td>
<td>cats</td>
<td>537</td>
</tr>
<tr>
<td>36.4.7</td>
<td>disable_registers</td>
<td>538</td>
</tr>
<tr>
<td>36.4.8</td>
<td>e500_floatingpoint</td>
<td>538</td>
</tr>
<tr>
<td>36.4.9</td>
<td>e500v2_floatingpoint</td>
<td>538</td>
</tr>
<tr>
<td>36.4.10</td>
<td>function_align</td>
<td>539</td>
</tr>
<tr>
<td>36.4.11</td>
<td>gen_fsel</td>
<td>539</td>
</tr>
<tr>
<td>36.4.12</td>
<td>gen_isel</td>
<td>539</td>
</tr>
<tr>
<td>36.4.13</td>
<td>gprfloatcopy</td>
<td>540</td>
</tr>
<tr>
<td>Section number</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>----------------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>36.4.14</td>
<td>has8bytebitfields</td>
<td>540</td>
</tr>
<tr>
<td>36.4.15</td>
<td>interrupt</td>
<td>541</td>
</tr>
<tr>
<td>36.4.16</td>
<td>legacy_struct_alignment</td>
<td>541</td>
</tr>
<tr>
<td>36.4.17</td>
<td>merge_float_consts</td>
<td>541</td>
</tr>
<tr>
<td>36.4.18</td>
<td>min_struct_align</td>
<td>542</td>
</tr>
<tr>
<td>36.4.19</td>
<td>misaligned_mem_access</td>
<td>542</td>
</tr>
<tr>
<td>36.4.20</td>
<td>no_register_save_helpers</td>
<td>543</td>
</tr>
<tr>
<td>36.4.21</td>
<td>options</td>
<td>543</td>
</tr>
<tr>
<td>36.4.22</td>
<td>pool_data</td>
<td>544</td>
</tr>
<tr>
<td>36.4.23</td>
<td>ppc_lvxl_stvxl_errata</td>
<td>544</td>
</tr>
<tr>
<td>36.4.24</td>
<td>profile</td>
<td>545</td>
</tr>
<tr>
<td>36.4.25</td>
<td>read_only_switch_tables</td>
<td>545</td>
</tr>
<tr>
<td>36.4.26</td>
<td>strict_ieee_fp</td>
<td>546</td>
</tr>
<tr>
<td>36.4.27</td>
<td>read_only_vtable_RTTI</td>
<td>546</td>
</tr>
<tr>
<td>36.4.28</td>
<td>switch_tables</td>
<td>547</td>
</tr>
<tr>
<td>36.4.29</td>
<td>uchar_bool</td>
<td>547</td>
</tr>
<tr>
<td>36.4.30</td>
<td>use_lmw_stmw</td>
<td>548</td>
</tr>
<tr>
<td>36.4.31</td>
<td>ushort_wchar_t</td>
<td>548</td>
</tr>
<tr>
<td>36.4.32</td>
<td>vec2x32float_align_4</td>
<td>548</td>
</tr>
<tr>
<td>36.4.33</td>
<td>z4_16bit_cond_branch_errata_5116</td>
<td>549</td>
</tr>
<tr>
<td>36.4.34</td>
<td>z4_mtlr_se_rfi_errata_26553</td>
<td>549</td>
</tr>
<tr>
<td>36.5</td>
<td>Optimization Pragmas</td>
<td>550</td>
</tr>
<tr>
<td>36.5.1</td>
<td>aggressive_hoisting</td>
<td>550</td>
</tr>
<tr>
<td>36.5.2</td>
<td>c9x_alias_by_type</td>
<td>551</td>
</tr>
<tr>
<td>36.5.3</td>
<td>epilogue_helper_functions</td>
<td>551</td>
</tr>
<tr>
<td>36.5.4</td>
<td>fp_contract</td>
<td>552</td>
</tr>
<tr>
<td>36.5.5</td>
<td>fp_contract_aggressive</td>
<td>552</td>
</tr>
<tr>
<td>36.5.6</td>
<td>ipa_rescopesGlobals</td>
<td>552</td>
</tr>
<tr>
<td>36.5.7</td>
<td>peephole</td>
<td>554</td>
</tr>
<tr>
<td>Section number</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>----------------</td>
<td>------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>36.5.8</td>
<td>peephole_enable_16bit_load_store_inst</td>
<td>555</td>
</tr>
<tr>
<td>36.5.9</td>
<td>ppc_opt_defuse_mem_limit</td>
<td>556</td>
</tr>
<tr>
<td>36.5.10</td>
<td>ppc_unroll_instructions_limit</td>
<td>556</td>
</tr>
<tr>
<td>36.5.11</td>
<td>ppc_unroll_speculative</td>
<td>557</td>
</tr>
<tr>
<td>36.5.12</td>
<td>processor</td>
<td>558</td>
</tr>
<tr>
<td>36.5.13</td>
<td>prologue_helper_functions</td>
<td>558</td>
</tr>
<tr>
<td>36.5.14</td>
<td>remove_frsp_aggressive</td>
<td>558</td>
</tr>
<tr>
<td>36.5.15</td>
<td>schedule</td>
<td>559</td>
</tr>
<tr>
<td>36.5.16</td>
<td>schedule z760</td>
<td>559</td>
</tr>
<tr>
<td>36.5.17</td>
<td>scheduling</td>
<td>560</td>
</tr>
<tr>
<td>36.5.18</td>
<td>switch_op</td>
<td>560</td>
</tr>
<tr>
<td>36.5.19</td>
<td>switch_cases_array</td>
<td>561</td>
</tr>
<tr>
<td>36.5.20</td>
<td>volatileasm</td>
<td>562</td>
</tr>
</tbody>
</table>
Chapter 1
Introduction

This reference explains how to use CodeWarrior tools to build programs. CodeWarrior build tools translate source code into object code then organize that object code to create a program that is ready to execute. CodeWarrior build tools run on the host system to generate software that runs on the target system. Sometimes the host and target are the same system. Usually, these systems are different.

This reference covers the CodeWarrior compiler and its linker, versions 4.0 and higher. This chapter explains the processes that CodeWarrior build tools use to create software:

- Compiler Architecture
- Linker Architecture

1.1 Compiler Architecture

From a programmer's point of view, the CodeWarrior compiler translates source code into object code. Internally, however, the CodeWarrior compiler organizes its work between its front-end and back-end, each end taking several steps Figure 1-1 shows the steps the compiler takes.
Front-end steps:

- read settings: retrieves your settings from the host's integrated development environment (IDE) or the command line to configure how to perform subsequent steps
- read and preprocess source code: reads your program's source code files and applies preprocessor directives
- translate to intermediate representation: translates your program's preprocessed source code into a platform-independent intermediate representation
- optimize intermediate representation: rearranges the intermediate representation to reduce your program's size, improve its performance, or both

Back-end steps:

- translate to processor object code: converts the optimized intermediate representation into native object code, containing data and instructions, for the target processor
- optimize object code: rearranges the native object code to reduce its size, improve performance, or both
- output object code and diagnostic data: writes output files on the host system, ready for the linker and diagnostic tools such as a debugger or profiler
1.2 Linker Architecture

The linker combines and arranges data and instructions from one or more object code files into a single file, or *image*. This image is ready to execute on the target platform. The CodeWarrior linker uses settings from the host's integrated development environment (IDE) or command line to determine how to generate the image file.

The linker also optionally reads a linker command file. A linker command file allows you to specify precise details of how data and instructions should be arranged in the image file.

*Figure 1-2* shows the steps the CodeWarrior linker takes to build an executable image.

![Figure 1-2. CodeWarrior Linker Steps](image)

- read settings: retrieves your settings from the IDE or the command line to determine how to perform subsequent steps
- read linker command file: retrieves commands to determine how to arrange object code in the final image
- read object code: retrieves data and executable objects that are the result of compilation or assembly
- delete unused objects ("deadstripping"): deletes objects that are not referred to by the rest of the program
• resolve references among objects: arranges objects to compose the image then computes the addresses of the objects
• output link map and image files: writes files on the host system, ready to load onto the target system
Chapter 2
Using Build Tools on the Command Line

CodeWarrior build tools may be invoked from the command-line. These command-line tools operate almost identically to their counterparts in an integrated development environment (IDE). CodeWarrior command-line compilers and assemblers translate source code files into object code files. CodeWarrior command-line linkers then combine one or more object code files to produce an executable image file, ready to load and execute on the target platform. Each command-line tool has options that you configure when you invoke the tool.

- Configuring Command-Line Tools
- Invoking Command-Line Tools
- Getting Help
- File Name Extensions

2.1 Configuring Command-Line Tools

This chapter explains:

- Setting CodeWarrior Environment Variables
- Setting the PATH Environment Variable

2.1.1 Setting CodeWarrior Environment Variables
Use environment variables on the host system to specify to the CodeWarrior command line tools where to find CodeWarrior files for compiling and linking. The table below describes these environment variables.

<table>
<thead>
<tr>
<th>The environment variable...</th>
<th>specifies this information</th>
</tr>
</thead>
<tbody>
<tr>
<td>CWFolder</td>
<td>CodeWarrior installation path on the host system.</td>
</tr>
<tr>
<td>MWCIncludes</td>
<td>Directories on the host system for system header files for the CodeWarrior compiler.</td>
</tr>
<tr>
<td>MWLibraries</td>
<td>Directories on the host system for system libraries for the CodeWarrior linker.</td>
</tr>
</tbody>
</table>

A system header file is a header file that is enclosed with the "<" and ">" characters in include directives. For example

```c
#include <stdlib.h> /* stdlib.h system header. */
```

Typically, you define the `MWCIncludes` and `MWLibraries` environment variables to refer to the header files and libraries in the subdirectories of your CodeWarrior software.

To specify more than one directory for the `MWCIncludes` and `MWLibraries` variables, use the conventional separator for your host operating system command-line shell.

**NOTE**

The `CWFolder` environment variable specifies the CodeWarrior installation path on a host system. Verify the CodeWarrior installation path on the host system before setting the `CWFolder` environment variable.

**Listing 2-1. Setting environment variables in Microsoft® Windows® operating systems**

```cmd
rem Use ; to separate directory paths
set CWFolder=Replace this text with the CodeWarrior installation path on the host system.

set MWCIncludes=%CWFolder%\PA_Support\ewl\EWL_C\include;%CWFolder%\PA_Support\ewl\EWL_C++\include;%CWFolder%\PA_Support\ewl\EWL_Runtime\Runtime_PA\Include

set MWLibraries=%CWFolder%\PA_Support\ewl\lib
```

### 2.1.2 Setting the PATH Environment Variable
The \texttt{PATH} variable should include the paths for your CodeWarrior tools, shown in the listing below. \textit{Toolset} represents the name of the folder that contains the command line tools for your build target.

\begin{verbatim}
set CWFolder=Replace this text with the CodeWarrior installation path on the host system.
set PATH=%PATH%;%CWFolder%\PA\Bin;%CWFolder%\PA\Command_Line_Tools
\end{verbatim}

### 2.2 Invoking Command-Line Tools

To compile, assemble, link, or perform some other programming task with the CodeWarrior command-line tools, you type a command at a command line's prompt. This command specifies the tool you want to run, what options to use while the tool runs, and what files the tool should operate on. For example,

\begin{verbatim}
   tool options files
\end{verbatim}

Where \texttt{tool} is the name of the CodeWarrior command-line tool to invoke, \texttt{options} is a list of zero or more options that specify to the tool what operation it should perform and how it should be performed, and \texttt{files} is a list of files zero or more files that the tool should operate on.

Which options and files you should specify depend on what operation you want the tool to perform.

The tool then performs the operation on the files you specify. If the tool is successful it simply finishes its operation and a new prompt appears at the command line. If the tool encounters problems it reports these problems as text messages on the command-line before a new prompt appears.

Scripts that automate the process to build a piece of software contain commands to invoke command-line tools. For example, the \texttt{make} tool, a common software development tool, uses scripts to manage dependencies among source code files and invoke command-line compilers, assemblers and linkers as needed.

Same can be done using response file.

\begin{verbatim}
   tool [@response_file] OR tool [@@response_file]
\end{verbatim}

Where \texttt{response_file} is used to insert command-line arguments from a file.

The response file is parsed such that arguments are separated by whitespace except where surrounded by quote marks. Anything followed by a pound character `\#` is used to indicate the rest of the line is a comment. Use `\#` in a response file if an argument actually starts with `\#`. 
2.3 Getting Help

To show short descriptions of a tool's options, type this command at the command line:

tool -help

Where tool is the name of the CodeWarrior build tool.

To show only a few lines of help information at a time, pipe the tool's output to a pager program. For example,

tool -help | more

Will use the more pager program to display the help information.

Enter the following command in a Command Prompt window to see a list of specifications that describe how options are formatted:
tool -help usage

Where *tool* is the name of the CodeWarrior build tool.

### 2.3.1 Parameter Formats

Parameters in an option are formatted as follows:

- A parameter included in brackets " []" is optional.
- Use of the ellipsis " . . . " character indicates that the previous type of parameter may be repeated as a list.

### 2.3.2 Option Formats

Options are formatted as follows:

- For most options, the option and the parameters are separated by a space as in " -xxx param".

  When the option's name is " -xxx", however, the parameter must directly follow the option, without the " +" character (as in " -xxx45") and with no space separator.

- An option given as " -[no]xxx" may be issued as " -xxx" or " -noxxx".
  The use of " -noxxx" reverses the meaning of the option.

- When an option is specified as " -xxx | yy[y] | zzz", then either " -xxx", " -yy", " -yyy", or " -zzz" matches the option.

- The symbols ", " and ", " separate options and parameters unconditionally; to include one of these symbols in a parameter or filename, escape it (e.g., as " \," in mwcc file.c\,v).

### 2.3.3 Common Terms

These common terms appear in many option descriptions:

- A "cased" option is considered case-sensitive. By default, no options are case-sensitive.
**2.4 File Name Extensions**

Files specified on the command line are identified by contents and file extension. The command-line version of the CodeWarrior C/C++ compiler accepts non-standard file extensions as source code but also emits a warning message. By default, the compiler assumes that a file with any extensions besides `.c`, `.h`, `.pch` is C++ source code. The linker ignores all files that it cannot identify as object code, libraries, or command files.

Linker command files must end in `.lcf`. They may be simply added to the link line, for example (Listing 2-2 on page 44).

**Listing 2-2. Example of using linker command files**

```bash
mwldtarget file.o lib.a commandfile.lcf
```

For more information on linker command files, refer to the Targeting manual for your platform.
Chapter 3
Command-Line Options for Standard C Conformance

This chapter explains the following command-line options for standard C conformance:

- `-ansi`
- `-stdkeywords`
- `-strict`

3.1 `-ansi`

Controls the ISO/IEC 9899-1990 ("C90") conformance options, overriding the given settings.

**Syntax**

```
-ansi [keyword]
```

The arguments for `keyword` are:

* `off`
  
  Turns ISO conformance off. Same as
  
  `-stdkeywords off -enum min -strict off`.

* `on | relaxed`
  
  Turns ISO conformance on in relaxed mode. Same as
  
  `-stdkeywords on -enum min -strict on`.

* `strict`
  
  Turns ISO conformance on in strict mode. Same as
  
  `-stdkeywords on -enum int -strict on`.
3.2  -stdkeywords

Controls the use of ISO/IEC 9899-1990 ("C90") keywords.

Syntax

   -stdkeywords on | off

Remarks

Default setting is off.

3.3  -strict

Controls the use of non-standard ISO/IEC 9899-1990 ("C90") language features.

Syntax

   -strict on | off

Remarks

If this option is on, the compiler generates an error message if it encounters some CodeWarrior extensions to the C language defined by the ISO/IEC 9899-1990 ("C90") standard:

   • C++-style comments
   • unnamed arguments in function definitions
   • non-standard keywords

The default setting is off.
Chapter 4
Command-Line Options for Standard C++ Conformance

This chapter explains the following command-line options for standard C++ conformance:

- -ARM
- -bool
- -Cpp_exceptions
- -dialect
- -for_scoping
- -instmgr
- -iso_templates
- -RTTI
- -som
- -som_env_check
- -wchar_t

4.1 -ARM

Deprecated. Use -for_scoping instead.

4.2 -bool

Controls the use of true and false keywords for the C++ bool data type.

Syntax
Remarks
When on, the compiler recognizes the true and false keywords in expressions of type bool. When off, the compiler does not recognize the keywords, forcing the source code to provide definitions for these names. The default is on.

4.3 -Cpp_exceptions
Controls the use of C++ exceptions.
Syntax

-Cpp_exceptions on | off

Remarks
When on, the compiler recognizes the try, catch, and throw keywords and generates extra executable code and data to handle exception throwing and catching. The default is on.

4.4 -dialect
Specifies the source language.
Syntax

-dialect keyword
-lang keyword

The arguments for keyword are:
c
Expect source code to conform to the language specified by the ISO/IEC 9899-1990 ("C90") standard.
c99
Expect source code to conform to the language specified by the ISO/IEC 9899-1999 ("C99") standard.
c++ | cplus
Always treat source as the C++ language.

ec++
Generate error messages for use of C++ features outside the Embedded C++ subset.
Implies -dialect cplus.

objc
Always treat source as the Objective-C language.

### 4.5 -for_scoping

Controls legacy scope behavior in for loops.

**Syntax**

- `for_scoping`

**Remarks**

When enabled, variables declared in for loops are visible to the enclosing scope; when disabled, such variables are scoped to the loop only. The default is `off`.

### 4.6 -instmgr

Controls whether the instance manager for templates is active.

**Syntax**

- `instmgr` keyword [, ...]

The options for `keyword` are:

- `off`
  
  Turn off the C++ instance manager. This is the default.

- `on`
Turn on the C++ instance manager.

```bash
file= path
```

Specify the path to the database used for the C++ instance manager. Unless specified the default database is `cwinst.db`.

**Remarks**

This command is global. The default setting is `off`.

### 4.7 -iso_templates

Controls whether the ISO/IEC 14882:2003 standard C++ template parser is active.

**Syntax**

```bash
-iso_templates on | off
```

**Remarks**

Default setting is `on`.

### 4.8 -RTTI

Controls the availability of runtime type information (RTTI).

**Syntax**

```bash
-RTTI on | off
```

**Remarks**

Default setting is `on`.

### 4.9 -som

Obsoleted. This option is no longer available.
4.10  -som_env_check

Obsolete. This option is no longer available.

4.11  -wchar_t

Controls the use of the wchar_t data type in C++ source code.

Syntax

-wchar_t on | off

Remarks

The -wchar on option tells the C++ compiler to recognize the wchar_t type as a built-in type for wide characters. The -wchar off option tells the compiler not to allow this built-in type, forcing the user to provide a definition for this type. Default setting is on.
Chapter 5
Command-Line Options for Language Translation

This chapter explains the following command-line options for language translation:

- -char
- -defaults
- -encoding
- -flag
- -gccext
- -gcc_extensions
- -M
- -make
- -mapcr
- -MM
- -MD
- -MMD
- -msext
- -once
- -pragma
- -relax_pointers
- -requireprotos
- -search
- -trigraphs

5.1 -char

Controls the default sign of the char data type.

Syntax

- char keyword
The arguments for \textit{keyword} are:

\begin{itemize}
  \item \texttt{signed} char data items are signed.
  \item \texttt{unsigned} char data items are unsigned.
\end{itemize}

\textbf{Remarks}

The default is \texttt{signed}.

\section*{5.2 \texttt{-defaults}}

Controls whether the compiler uses additional environment variables to provide default settings.

\textbf{Syntax}

\begin{itemize}
  \item \texttt{-defaults}
  \item \texttt{-nodefaults}
\end{itemize}

\textbf{Remarks}

This option is global. To tell the command-line compiler to use the same set of default settings as the CodeWarrior IDE, use \texttt{-defaults}. For example, in the IDE, all access paths and libraries are explicit. \texttt{nodefaults} is the default setting.

Use \texttt{-nodefaults} to disable the use of additional environment variables.

\section*{5.3 \texttt{-encoding}}

Specifies the default source encoding used by the compiler.

\textbf{Syntax}

\begin{itemize}
  \item \texttt{-encoding} \texttt{keyword}
\end{itemize}

The options for \texttt{keyword} are:

\begin{itemize}
  \item \texttt{ascii}
\end{itemize}
American Standard Code for Information Interchange (ASCII) format. This is the default.

**autodetect | multibyte | mb**

Scan file for multibyte encoding.

**system**

Uses local system format.

**UTF[8 | -8]**

Unicode Transformation Format (UTF).

**SJIS | Shift-JIS | ShiftJIS**

Shift Japanese Industrial Standard (Shift-JIS) format.

**EUC[JP | -JP]**

Japanese Extended UNIX Code (EUCJP) format.


International Organization of Standards (ISO) Japanese format.

**Remarks**

The compiler automatically detects UTF-8 (Unicode Transformation Format) header or UCS-2/UCS-4 (Uniform Communications Standard) encodings regardless of setting. The default setting is `ascii`.

### 5.4 -flag

Specifies compiler `#pragma` as either on or off.

**Syntax**

```
-flags [no-]pragma
```

**Remarks**

For example, this option setting

```
-flags require_prototypes
```

is equivalent to

```
#pragma require_prototypes on
```
This option setting

- flag no-require_prototypes

is the same as

#pragma require_prototypes off

5.5  -gccext

Enables GCC (Gnu Compiler Collection) C language extensions.

Syntax

-gcc[ext] on | off

Remarks

See GCC Extensions for a list of language extensions that the compiler recognizes when this option is on.

The default setting is off.

5.6  -gcc_extensions

Equivalent to the -gccext option.

Syntax

-gcc[_extensions] on | off

5.7  -M

Scans source files for dependencies and emit a Makefile, without generating object code.

Syntax

-M

Remarks

CodeWarrior Development Studio for Power Architecture Processors Build Tools Reference Manual, Rev. 10.x, 06/2015
This command is global and case-sensitive.

5.8  -make

Scans source files for dependencies and emit a Makefile, without generating object code.

Syntax
   -make

Remarks
This command is global.

5.9  -mapcr

Swaps the values of the \n and \r escape characters.

Syntax
   -mapcr
   -nomapcr

Remarks
The -mapcr option tells the compiler to treat the '\n' character as ASCII 13 and the '\r' character as ASCII 10. The -nomapcr option tells the compiler to treat these characters as ASCII 10 and 13, respectively.

5.10  -MM

Scans source files for dependencies and emit a Makefile, without generating object code or listing system include files.

Syntax
   -MM

Remarks

This command is global and case-sensitive.

5.11  -MD

Scans source files for dependencies and emit a Makefile, generate object code, and write a dependency map.

Syntax

-MD

Remarks

This command is global and case-sensitive.

5.12  -MMD

Scans source files for dependencies and emit a Makefile, generate object code, write a dependency map, without listing system #include files.

Syntax

-MMD

Remarks

This command is global and case-sensitive.

5.13  -msext

Allows Microsoft® Visual C++ extensions.

Syntax

-msext on | off

Remarks

Turn on this option to allow Microsoft Visual C++ extensions:

• Redefinition of macros


- Allows \texttt{xxx:y\_yy} syntax when declaring method \texttt{yy\_y} of class \texttt{xxx}
- Allows extra commas
- Ignores casts to the same type
- Treats function types with equivalent parameter lists but different return types as equal
- Allows pointer-to-integer conversions, and various syntactical differences

### 5.14 -\texttt{once}

Prevents header files from being processed more than once.

**Syntax**

\begin{verbatim}
-\texttt{once}
\end{verbatim}

**Remarks**

You can also add \texttt{#pragma once} on in a prefix file.

### 5.15 -\texttt{pragma}

Defines a pragma for the compiler.

**Syntax**

\begin{verbatim}
-\texttt{pragma "name [setting]"}
\end{verbatim}

The arguments are:

- name
  Same of the pragma.
- setting
  Arguments to give to the pragma

**Remarks**

For example, this command-line option

\begin{verbatim}
-\texttt{pragma "c99 on"}
\end{verbatim}

Is equivalent to inserting this directive in source code
5.16  -relax_pointers

Relaxes the pointer type-checking rules in C.

Syntax

-relax_pointers

Remarks

This option is equivalent to

#pragma mpwc_relax on

5.17  -requireprotos

Controls whether or not the compiler should expect function prototypes.

Syntax

-r[quireprotos]

5.18  -search

Globally searches across paths for source files, object code, and libraries specified in the command line.

Syntax

-search

5.19  -trigraphs

Controls the use of trigraph sequences specified by the ISO/IEC standards for C and C++.

Syntax
-trigraphs on | off

Remarks
Default setting is off.
Chapter 6
Command-Line Options for Diagnostic Messages

This chapter explains the following command-line options for diagnostic messages:

- -disassemble
- -warningerror
- -help
- -maxerrors
- -maxwarnings
- -msgstyle
- -ide_err
- -nofail
- -progress
- -S
- -stderr
- -verbose
- -version
- -timing
- -warnings
- -wraplines

6.1 -disassemble

Instructs the command-line tool to disassemble files and send result to stdout.

Syntax

-disassemble

Remarks

This option is global.
6.2 -warningerror

Controls whether or not warnings are treated as errors.

**Syntax**

-warningerror [on | off]

**Remarks**

If you enable this option, the compiler treats all warning messages as though they were errors and does not translate your file until you resolve them.

6.3 -help

Lists descriptions of the CodeWarrior tool's command-line options.

**Syntax**

-help [keyword [, ...]]

The options for keyword are:

- **all**
  
  Show all standard options

- **group= keyword**
  
  Show help for groups whose names contain keyword (case-sensitive).

- **[no]compatible**
  
  Use compatible to show options compatible with this compiler. Use nocompatible to show options that do not work with this compiler.

- **[no]deprecated**
  
  Shows deprecated options

- **[no]ignored**
  
  Shows ignored options

- **[no]meaningless**
Shows options meaningless for this target

[no]normal

Shows only standard options

[no]obsolete

Shows obsolete options

[no]spaces

Inserts blank lines between options in printout.

opt[ion]= name

Shows help for a given option; for name, maximum length 63 chars

search= keyword

Shows help for an option whose name or help contains keyword (case-sensitive), maximum length 63 chars

tool=keyword[ all | this | other | skipped | both ]

Categorizes groups of options by tool; default.

- all-show all options available in this tool
- this-show options executed by this tool; default
- other | skipped-show options passed to another tool
- both-show options used in all tools

usage

Displays usage information.

6.4 -maxerrors

Specifies the maximum number of errors messages to show.

Syntax

-maxerrors max

max

Use max to specify the number of error messages. Common values are:
-maxwarnings

- maxwarnings 0
  - (zero) - disable maximum count, show all error messages (default).
- maxwarnings n
  - Maximum number of errors to show, such as -maxwarnings.

### 6.5 -maxwarnings

Specifies the maximum number of warning messages to show.

**Syntax**

```
-maxwarnings max
```

*max*

Specifies the number of warning messages. Common values are:

- 0 (zero) - Disable maximum count (default).
- n - Maximum number of warnings to show.

### 6.6 -msgstyle

Controls the style used to show error and warning messages.

**Syntax**

```
-msgstyle keyword
```

The options for *keyword* are:

- **gcc**
  Uses the message style that the Gnu Compiler Collection tools use.

- **IDE**
  Uses CodeWarrior's Integrated Development Environment (IDE) message style.

- **IDE_ERR**
  Uses CodeWarrior's Integrated Development Environment (IDE) error message style.

- **ide**
  Uses CodeWarrior's Integrated Development Environment (IDE) message style.

- **mpw**
  Uses CodeWarrior's Integrated Development Environment (IDE) message style.
Uses Macintosh Programmer's Workshop (MPW®) message style.

```
parseable
```

Uses context-free machine parseable message style.

```
std
```

Uses standard message style. This is the default.

```
enterpriseIDE
```

Uses Enterprise-IDE message style.

### 6.7 -ide_err

Uses CodeWarrior's integrated development environment (IDE) error message style.

**Syntax**

```
-ide_err
```

### 6.8 -nofail

Continues processing after getting error messages in earlier files.

**Syntax**

```
-nofail
```

### 6.9 -progress

Shows progress and version information.

**Syntax**

```
-progress
```
6.10  -S

Disassembles all files and send output to a file. This command is global and case-sensitive.

Syntax

-S

6.11  -stderr

Uses the standard error stream to report error and warning messages.

Syntax

-stderr

-nostderr

Remarks

The -stderr option specifies to the compiler, and other tools that it invokes, that error and warning messages should be sent to the standard error stream.

The -nostderr option specifies that error and warning messages should be sent to the standard output stream.

6.12  -verbose

Instructs the compiler to provide extra, cumulative information in messages.

Syntax

-v[erbose]
Remarks
This option also gives progress and version information.

6.13 -version

Displays version, configuration, and build data.

Syntax

   -v[ersion]

6.14 -timing

Shows the amount of time that the tool used to perform an action.

Syntax

   -timing

6.15 -warnings

Specifies which warning messages the command-line tool issues. This command is global.

Syntax

   -w[arnings] keyword [, ...]

The options for keyword are:

   off

Turns off all warning messages. Passed to all tools. Equivalent to

   #pragma warning off
-warnings

on

Turns on most warning messages. Passed to all tools. Refer Table 6-1 for a list of warning messages turned on by the -warnings on command.

Equivalent to #pragma warning on

most

Turns on most warnings.

all

Turns on almost all warnings and require prototypes.

full

Turns on all warning messages and require prototypes. This option is likely to generate spurious warnings.

NOTE
-warnings full should be used before using any other options that affect warnings. For example, use -warnings full -warnings noanyptrintconv instead of -warnings noanyptrintconv -warnings full.

[no]cmdline

Passed to all tools.

[no]err[or] | [no]iserr[or]

Treats warnings as errors. Passed to all tools. Equivalent to

#pragma warning_errors

[no]pragmas | [no]illpragmas

Issues warning messages on invalid pragmas. Enabled when most is used. Equivalent to

#pragma warn_illpragma

[no]empty[decl]

Issues warning messages on empty declarations. Enabled when most is used. Equivalent to

#pragma warn_emptydecl

[no]possible | [no]unwanted
Issues warning messages on possible unwanted effects. Enabled when `most` is used. Equivalent to

```
#pragma warn_possunwanted
```

`[no]unusedarg`

Issues warning messages on unused arguments. Enabled when `most` is used. Equivalent to

```
#pragma warn_unusedarg
```

`[no]unusedvar`

Issues warning messages on unused variables. Enabled when `most` is used. Equivalent to

```
#pragma warn_unusedvar
```

`[no]unused`

Same as

```
-w [no]unusedarg,[no]unusedvar
```

Enabled when `most` is used.

`[no]extracomma | [no]comma`

Issues warning messages on extra commas in enumerations. The compiler ignores terminating commas in enumerations when compiling source code that conforms to the ISO/IEC 9899-1999 ("C99") standard. Enabled when `most` is used. Equivalent to

```
#pragma warn_extracomma
```

`[no]extended`

Extended error checking. Enabled when `most` is used. Equivalent to either:

```
#pragma extended_errorcheck
```

`[no]hidevirtual | [no]hidden[virtual]`
Issues warning messages on hidden virtual functions. Enabled when most is used. Equivalent to

    #pragma warn_hidevirtual

[no]implicit[conv]

Issues warning messages on implicit arithmetic conversions. Enabled when all is used. Implies

    -warn impl_float2int,impl_signedunsigned

[no]impl_int2float

Issues warning messages on implicit integral to floating conversions. Enabled when all is used. Equivalent to

    #pragma warn_impl_i2f_conv

[no]impl_float2int

Issues warning messages on implicit floating to integral conversions. Enabled when all is used. Equivalent to

    #pragma warn_impl_f2i_conv

[no]impl_signedunsigned

Issues warning messages on implicit signed/unsigned conversions. Enabled when all is used.

[no]notinlined

Issues warning messages for functions declared with the inline qualifier that are not inlined. Enabled when full is used. Equivalent to

    #pragma warn_notinlined

[no]largeargs

Issues warning messages when passing large arguments to unprototyped functions. Enabled when most is used. Equivalent to
#pragma warn_largeargs

[no]structclass
Issues warning messages on inconsistent use of `class` and `struct`. Enabled when `most` is used. Equivalent to

    #pragma warn_structclass

[no]padding
Issue warning messages when padding is added between `struct` members. Enabled when `full` is used. Equivalent to

    #pragma warn_padding

[no]notused
Issues warning messages when the result of non-void-returning functions are not used. Enabled when `full` is used. Equivalent to

    #pragma warn_resultnotused

[no]missingreturn
Issues warning messages when a return without a value in non-void-returning function occurs. Enabled when `most` is used. Equivalent to

    #pragma warn_missingreturn

[no]unusedexpr
Issues warning messages when encountering the use of expressions as statements without side effects. Equivalent to

    #pragma warn_no_side_effect

[no]printfconv
Issues warning messages when lossy conversions occur from pointers to integers. Enabled when `full` is used.

`[no]anyptrintconv`

Issues warning messages on any conversion of pointers to integers. Enabled when `full` is used. Equivalent to

```c
#pragma warn_ptr_int_conv
```

`[no]undef [macro]`

Issues warning messages on the use of undefined macros in `#if` and `#elif` conditionals. Enabled when `full` is used. Equivalent to

```c
#pragma warn_UNDEFmacro
```

`[no]filecaps`

Issues warning messages when `# include ""` directives use incorrect capitalization. Enabled when `most` is used. Equivalent to

```c
#pragma warn_filenamecaps
```

`[no]sysfilecaps`

Issue warning messages when `# include <>` statements use incorrect capitalization. Enabled when `most` is used. Equivalent to

```c
#pragma warn_filenamecaps_system
```

`[no]tokenpasting`

Issue warning messages when token is not formed by the `##` preprocessor operator. Enabled when `most` is used. Equivalent to

```c
#pragma warn_illtokenpasting
```

`[no]relax_i2i_conv`

Relax implicit arithmetic conversion warnings on certain implicit conversions. Equivalent to
#pragma relax_i2i_conv

noSymRedef

Suppress symbol redefined warnings. Passed to linker (By default linker emits Symbol Redefined warnings).

display | dump

Display list of active warnings.

Remarks

The table below lists the equivalent command option of the warning messages turned on by the `-w[arnings] on` command.

<table>
<thead>
<tr>
<th>on/most</th>
<th>all (includes most)</th>
<th>full (includes all and most)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[no]empty[decl]</td>
<td></td>
<td>[no]impl_float2int</td>
</tr>
<tr>
<td>[no]unused</td>
<td></td>
<td>[no]undef[macro]</td>
</tr>
<tr>
<td>[no]extracomma</td>
<td>[no]comma</td>
<td></td>
</tr>
<tr>
<td>[no]extended</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[no]hidevirtual</td>
<td>[no]hidden[virtual]</td>
<td></td>
</tr>
<tr>
<td>[no]largeargs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[no]structclass</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[no]missingreturn</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[no]unusedexpr</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[no]filecaps</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[no]sysfilecaps</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[no]tokenpasting</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6.16  -wraplines

Controls the word wrapping of messages.

Syntax
Chapter 7
Command-Line Options for Preprocessing

This chapter explains the following command-line options for preprocessing:

- -convertpaths
- -cwd
- -D+
- -define
- -E
- -EP
- -gccincludes
- -I-
- -I+
- -include
- -ir
- -P
- -precompile
- -preprocess
- -ppopt
- -prefix
- -noprecompile
- -nosyspath
- -stdinc
- -U+
- -undefine

7.1 -convertpaths

Instructs the compiler to interpret # include file paths specified for a foreign operating system. This command is global.
Remarks

The CodeWarrior compiler can interpret file paths from several different operating systems. Each operating system uses unique characters as path separators. These separators include:

- Mac OS® - colon " : " ( :sys:stat.h )
- UNIX - forward slash " / " ( sys/stat.h )
- Windows® operating systems - backward slash " \ " ( sys\stat.h )

When \convertpaths is enabled, the compiler can correctly interpret and use paths like \ <sys/stat.h> or \ <:sys:stat.h>. However, when enabled, ( / ) and ( : ) separate directories and cannot be used in filenames.

**NOTE**

This is not a problem on Windows systems since these characters are already disallowed in file names. It is safe to leave this option on.

When \noconvertpaths is enabled, the compiler can only interpret paths that use the Windows form, like \ <\sys\stat.h>.

### 7.2 \-cwd

Controls where a search begins for \# include files.

**Syntax**

\-cwd keyword

The options for \keyword are:

- explicit
  
  No implicit directory. Search \-I or \-Ir paths.

- include
Begins searching in directory of referencing file.

proj

Begins searching in current working directory (default).

source

Begins searching in directory that contains the source file.

Remarks

The path represented by keyword is searched before searching access paths defined for the build target.

7.3 -D+

Same as the -define option.

Syntax

-D+ name

The parameters are:

name

The symbol name to define. Symbol is set to 1.

7.4 -define

Defines a preprocessor symbol.

Syntax

-define name=value

The parameters are:

name

The symbol name to define.
value

The value to assign to symbol name. If no value is specified, set symbol value equal to 1.

7.5  -E

Tells the command-line tool to preprocess source files.

Syntax

    -E

Remarks

This option is global and case sensitive.

7.6  -EP

Tells the command-line tool to preprocess source files that are stripped of #line directives.

Syntax

    -EP

Remarks

This option is global and case sensitive.

7.7  -gccincludes

Controls the compilers use of GCC #include semantics.

Syntax
Remarks

Use `-gccinc[ludes]` to control the CodeWarrior compiler understanding of Gnu Compiler Collection (GCC) semantics. When enabled, the semantics include:

- Adds `-I-` paths to the systems list if `-I-` is not already specified
- Search referencing file's directory first for `#include` files (same as `-cwd include`) The compiler and IDE only search access paths, and do not take the currently `#include` file into account.

This command is global.

7.8  `-I-`

Changes the build target's search order of access paths to start with the system paths list.

Syntax

```
-I-
```

Remarks

The compiler can search `#include` files in several different ways. Use `-I-` to set the search order as follows:

- For include statements of the form `#include "xyz"`, the compiler first searches user paths, then the system paths
- For include statements of the form `#include <xyz>`, the compiler searches only system paths

This command is global.

7.9  `-I+`
Appends a non-recursive access path to the current `#include` list.

**Syntax**

```
-I+path
-i path
```

The parameters are:

`path`

The non-recursive access path to append.

**Remarks**

This command is global and case-sensitive.

### 7.10 `-include`

Defines the name of the text file or precompiled header file to add to every source file processed.

**Syntax**

```
-include file
```

`file`

Name of text file or precompiled header file to prefix to all source files.

**Remarks**

With the command line tool, you can add multiple prefix files all of which are included in a meta-prefix file.

### 7.11 `-ir`

Appends a recursive access path to the current `#include` list. This command is global.

**Syntax**

CodeWarrior Development Studio for Power Architecture Processors Build Tools Reference Manual, Rev. 10.x, 06/2015

Freescale Semiconductor, Inc.
-ir path

path
The recursive access path to append.

7.12 -P

Preprocesses the source files without generating object code, and send output to file.

Syntax

-P

Remarks
This option is global and case-sensitive.

7.13 -precompile

Precompiles a header file from selected source files.

Syntax

-precompile file | dir | ""

file
If specified, the precompiled header name.

dir
If specified, the directory to store the header file.

If "" is specified, write header file to location specified in source code. If neither argument is specified, the header file name is derived from the source file name.

Remarks
The driver determines whether to precompile a file based on its extension. The option
is equivalent to

    -c -o
    filesource

7.14  -preprocess

Preprocesses the source files. This command is global.

Syntax

    -preprocess

7.15  -ppopt

Specifies options affecting the preprocessed output.

Syntax

    -ppopt keyword [, ...]

The arguments for keyword are:

[no]break

Emits file and line breaks. This is the default.

[no]line

Controls whether #line directives are emitted or just comments. The default is line.

[no]full[path]

Controls whether full paths are emitted or just the base filename. The default is fullpath.

[no]pragma

Controls whether #pragma directives are kept or stripped. The default is pragma.

[no]comment

Controls whether comments are kept or stripped.
[no]space

Controls whether whitespace is kept or stripped. The default is space.

Remarks
The default setting is break.

7.16  -prefix

Adds contents of a text file or precompiled header as a prefix to all source files.

Syntax

-prefix file

7.17  -noprecompile

Do not precompile any source files based upon the filename extension.

Syntax

-noprecompile

7.18  -nosyspath

Performs a search of both the user and system paths, treating #include statements of the form #include <xyz> the same as the form #include "xyz".

Syntax

-nosyspath

Remarks
This command is global.

### 7.19 -stdinc

Uses standard system include paths as specified by the environment variable `%MWCIncludes`.

**Syntax**

```
-stdinc

-nostdinc
```

**Remarks**

Add this option after all system `-I` paths.

### 7.20 -U+

Same as the `-undefine` option.

**Syntax**

```
-U+ name
```

### 7.21 -undefine

Undefines the specified symbol name.

**Syntax**

```
-u[ndefine] name

-U+name
```
name

The symbol name to undefine.

Remarks

This option is case-sensitive.
Chapter 8
Command-Line Options for Library and Linking

This chapter explains the following command-line options for library and linking.

- -keepobjects
- -nolink
- -o

8.1 -keepobjects

Retains or deletes object files after invoking the linker.

Syntax
- -keepobjects
- -nolink

Remarks
Use -keepobjects to retain object files after invoking the linker. Use -nolink to delete object files after linking. This option is global.

NOTE
Object files are always kept when compiling.

8.2 -nolink

Compiles the source files, without linking.

Syntax
- -nolink
Remarks
This command is global.

8.3  -o

Specifies the output filename or directory for storing object files or text output during compilation, or the output file if calling the linker.

Syntax

    -o file | dir

file

The output file name.

dir

The directory to store object files or text output.
Chapter 9
Command-Line Options for Object Code

This chapter explains the following command-line options for object code:

- -c
- -codegen
- -enum
- -min_enum_size
- -ext
- -strings

9.1 -c

Instructs the compiler to compile but not invoke the linker to link the object code.

Syntax
- -c

Remarks
This option is global.

9.2 -codegen

Instructs the compiler to compile without generating object code.

Syntax
- -codegen
- -nocodegen
Remarks
This option is global.

9.3  -enum

Specifies the default size for enumeration types.

Syntax
-enum keyword

The arguments for keyword are:

int

Uses int size for enumerated types.

min

Uses minimum size for enumerated types. This is the default.

9.4  -min_enum_size

Specifies the size, in bytes, of enumerated types.

Syntax
-min_enum_size 1 | 2 | 4

Remarks
Specifying this option also invokes the -enum min option by default.

9.5  -ext

Specifies which file name extension to apply to object files.

Syntax
-ext extension
The extension to apply to object files. Use these rules to specify the extension:

- Limited to a maximum length of 14 characters
- Extensions specified without a leading period replace the source file's extension. For example, if `extension` is "o" (without quotes), then `source.cpp` becomes `source.o`.
- Extensions specified with a leading period (`.extension`) are appended to the object files name. For example, if `extension` is ".o" (without quotes), then `source.cpp` becomes `source.cpp.o`.

**Remarks**

This command is global. The default setting is `.o`.

### 9.6 -strings

Controls how string literals are stored and used.

**Remarks**

- `-str[ings]` keyword[, ...]

The `keyword` arguments are:

- [no]pool

All string constants are stored as a single data object so your program needs one data section for all of them.

- [no]reuse

All equivalent string constants are stored as a single data object so your program can reuse them. This is the default.

- [no]readonly

Make all string constants read-only. This is the default.
Chapter 10
Command-Line Options for Optimization

This chapter explains the following command-line options for optimization:

- inline
- -ipa
- -O
- -O+
- -opt

10.1 -inline

Specifies inline options. Default settings are smart, noauto.

**Syntax**

```
-inline keyword
```

The options for `keyword` are:

- `off` | `none`
  Turns off inlining.

- `on` | `smart`
  Turns on inlining for functions declared with the `inline` qualifier. This is the default.

- `auto`
  Attempts to inline small functions even if they are declared with `inline`.

- `noauto`
Does not auto-inline. This is the default auto-inline setting.

defferred

Refrains from inlining until a file has been translated. This allows inlining of functions in both directions.

level=n

Inlines functions up to $n$ levels deep. Level 0 is the same as -inline on. For $n$, enter 1 to 8 levels. This argument is case-sensitive.

all

Turns on aggressive inlining. This option is the same as -inlineon, -inlineauto. This does not turn on the aggressive_inline feature.

10.2 -ipa

Controls Interprocedural Analysis (IPA) that lets the compiler generate better optimizations by evaluating all the functions and data objects in a file or program before generating code.

Syntax

-IPA file | function | off | program | program-final

function | off

Per-function optimization. This is the default option.

file

Per file optimization.

program

Per-program optimization

Remarks

See Interprocedural Analysis.

NOTE

-ipa program mode is available only with command line compiler.
Using IPA mode from command-line tools is more complicated. If you specify all source files on the command-line you can just use `-ipa program`:

Use the `off` or `function` arguments to turn interprocedural analysis off. This is the default setting.

Use the `file` argument to apply interprocedural analysis at the file level. For example, if the name of the compiler is `mwcc`, the command: `mwcc -ipa file -c file1.c file2.c` generates object code and applies this optimization to file `file1.c` and then `file2.c`, but does not apply the optimization across both files. For each source file, this command generates a regular object code file (a file with a name that ends with ".o" or ".obj"), which is empty. It also generates an additional file ending with ".irobj". This additional object code file contains the object code to which the compiler has applied interprocedural analysis.

This example compiles the same source files again, applies file-level analysis, then links object code into an output file named `myprog`:

```
    mwcc -o myprog -ipa file -c file1.c file2.c
```

Use the `program` argument to apply this optimization among all files being compiled and linked. For example: `mwcc -o myprog -ipa program file1.c file2.c` generates object code, applies this optimization among all resulting object code files to link it into an output file named `myprog`.

To separate compiling and linking steps for program-level interprocedural analysis, use the `program` and `program-final` arguments. For example:

```
    mwcc -ipa program -c file1.c
    mwcc -ipa program -c file2.c
```

compiles `file1.c` and `file2.c` into empty regular object files named `file1.o` and `file2.o`, respectively, and optimized object files named `file1.irobj` and `file2.irobj`.

To link object files, refer to the regular object files or the optimized object files. For example:

```
    mwcc -o myprog -ipa program file1.o file2.o
```

or, equivalently:

```
    mwcc -o myprog -ipa program file1.irobj file2.irobj
```

To invoke the linker directly, use the `program-final` argument. For example, these commands prepare object code for linking, then invoke the linker directly (named `mwld` in this example):

```
    mwcc -ipa program-final file1.irobj file2.irobj
    mwld -o myprog file1.o file2.o
```
It is possible to mix program and IPA modes using the command-line tool options. To mix modes in the plug-in compiler you will have to use "#pragma ipa".

10.3 -O

Sets optimization settings to -opt level=2.

Syntax

-0

Remarks

Provided for backwards compatibility.

10.4 -O+

Controls optimization settings.

Syntax

-0+keyword [, ...]

The keyword arguments are:

0

Equivalent to -opt off.

1

Equivalent to -opt level=1.

2

Equivalent to -opt level=2, peephole

3

Equivalent to -opt level=3, peephole.
Equivalent to `-opt level=4, peephole, schedule, autoinline, func_align 16.`

Equivalent to `-opt speed.`

Equivalent to `-opt space.`

**Remarks**

Options can be combined into a single command. Command is case-sensitive.

### 10.5 `-opt`

Specifies code optimization options to apply to object code.

**Remarks**

- `-optkeyword [, ...]`

The *keyword* arguments are:

`off | none`

Suppresses all optimizations. This is the default.

`on`

Same as `-opt level=2, peephole`

`all | full`

Same as `-opt speed,level=4, schedule, intrinsics, noframe peephole, autoline, func_align 16`

`l[evel]=num`

Sets a specific optimization level. The options for *num* are:

- **0** - Global register allocation only for temporary values. Equivalent to `#pragma optimization_level 0`.
- **1** - Adds dead code elimination, branch and arithmetic optimizations, expression simplification. Equivalent to `#pragma optimization_level 1`.
- **2** - Adds common subexpression elimination, copy and expression propagation, stack frame compression, stack alignment, fast floating-point to integer conversions. Equivalent to: `#pragma optimization_level 2`. 
• 3 - Adds dead store elimination, live range splitting, loop-invariant code motion, strength reduction, loop transformations, loop unrolling (with -opt speed only), loop vectorization, lifetime-based register allocation. Equivalent to optimization_level 3.

• 4 - Like level 3, but with more comprehensive optimizations from levels 1 and 2. Equivalent to #pragma optimization_level 4.

For num options 0 through 4 inclusive, the default is 0.

[no] space

Optimizes object code for size. Equivalent to #pragma optimize_for_size on.

[no] speed

Optimizes object code for speed. Equivalent to #pragma optimize_for_size off.

[no] cse | [no] commonsubs

Common subexpression elimination. Equivalent to #pragma opt_common_subs.

[no] deadcode

Removes dead code. Equivalent to #pragma opt_dead_code.

[no] deadstore

Removes dead assignments. Equivalent to #pragma opt_dead_assignments.

[no] lifetimes

Computes variable lifetimes. Equivalent to #pragma opt_lifetimes.

[no] loop[invariants]

Removes loop invariants. Equivalent to #pragma opt_loop_invariants.

[no] prop[agation]

Propagation of constant and copy assignments. Equivalent to #pragma opt_propagation.

[no] strength

Strength reduction. Reducing multiplication by an array index variable to addition. Equivalent to #pragma opt_strength_reduction.

[no] dead

Same as -opt [no] deadcode and [no] deadstore. Equivalent to #pragma opt_dead_code on|off and #pragma opt_dead_assignments.

[no] alias_by_type

Type based alias optimizations. Equivalent to #pragma c9x_alias_by_type.

[no] peep[hole]
Peephole optimization. Equivalent to `#pragma peephole`.

`[no] schedule`

Performs instruction scheduling.

`display | dump`

Displays complete list of active optimizations.

*Table 10-1* explains the supported optimizations and their availability at certain optimization levels.

**Table 10-1. Optimization Routines**

<table>
<thead>
<tr>
<th>Optimization Routine</th>
<th>Explanation</th>
<th>Optimization Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global Register Allocation or Global Register Allocation Only for Temporary Values</td>
<td>Stores working values of heavily used variables in registers instead of memory.</td>
<td>1, 2, 3, 4</td>
</tr>
<tr>
<td>Dead Code Elimination</td>
<td>Removes statements never logically executed or referred to by other statements.</td>
<td>1, 2, 3, 4</td>
</tr>
<tr>
<td>Branch Optimizations</td>
<td>Merges and restructures portions of the intermediate code translation in order to reduce branch instructions.</td>
<td>1, 2, 3, 4</td>
</tr>
<tr>
<td>Arithmetic Operations</td>
<td>Replaces intensive computational instructions with faster equivalent instructions that produce the same result.</td>
<td>1, 2, 3, 4</td>
</tr>
<tr>
<td>Expression Simplification</td>
<td>Replaces complex arithmetic expressions with simplified equivalent expressions.</td>
<td>1, 2, 3, 4</td>
</tr>
<tr>
<td>Common Subexpression Elimination</td>
<td>Replaces redundant expressions with a single expression.</td>
<td>2, 3, 4</td>
</tr>
<tr>
<td>Copy Propagation or Copy and Expression Propagation</td>
<td>Replaces multiple occurrences of one variable with a single occurrence.</td>
<td>2, 3, 4</td>
</tr>
<tr>
<td>Peephole Optimization</td>
<td>Applies local optimization routines to small sections of code.</td>
<td>2, 3, 4</td>
</tr>
<tr>
<td>Dead Store Elimination</td>
<td>Removes assignments to a variable that goes unused before being reassigned again.</td>
<td>3, 4</td>
</tr>
<tr>
<td>Live Range Splitting</td>
<td>Reduces variable lifetimes to achieve optimal allocation. Shorter variable lifetimes reduce register spilling.</td>
<td>3, 4</td>
</tr>
<tr>
<td>Loop-Invariant Code Motion</td>
<td>Moves static computations outside of a loop</td>
<td>3, 4</td>
</tr>
<tr>
<td>Strength Reduction</td>
<td>Inside loops, replaces multiplication instructions with addition instructions.</td>
<td>3, 4</td>
</tr>
<tr>
<td>Loop Transformations</td>
<td>Reorganizes loop object code in order to reduce setup and completion-test overhead.</td>
<td>3, 4</td>
</tr>
</tbody>
</table>

*Table continues on the next page...*
<table>
<thead>
<tr>
<th>Optimization Routine</th>
<th>Explanation</th>
<th>Optimization Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop Unrolling or Loop Unrolling (Opt for Speed Only)</td>
<td>Duplicates code inside a loop in order to spread branch and completion-test overhead over more operations.</td>
<td>3, 4</td>
</tr>
<tr>
<td>Vectorization</td>
<td>For processors that support vector optimizations, translates computations with code-loop arrays into equivalent vector instructions.</td>
<td>3, 4</td>
</tr>
<tr>
<td>Lifetime Based Register Allocation or Register Coloring</td>
<td>In a particular routine, uses the same processor register to store different variables, as long as no statement uses those variables simultaneously.</td>
<td>3, 4</td>
</tr>
<tr>
<td>Instruction Scheduling</td>
<td>Rearranges the instruction sequence to reduce conflicts among registers and processor resources.</td>
<td>None</td>
</tr>
<tr>
<td>Repeated</td>
<td>Iterates the optimization routines listed between { * and * }.</td>
<td>4</td>
</tr>
<tr>
<td>Stack alignment</td>
<td>Aligns the stack pointer. Required when load/store instruction has an alignment requirement.</td>
<td>2,3,4</td>
</tr>
<tr>
<td>Stack frame compression</td>
<td>Performs live/dead analysis on all non-static symbols (compiler generated temporaries, automatics, input parameters, and outgoing parameters). Using the live/dead information, all the dead stores are eliminated.</td>
<td>2,3,4</td>
</tr>
</tbody>
</table>
Chapter 11
Command-Line for Power Architecture Processors

This chapter describes how to use the command-line tools to generate, examine, and manage source code and object code for Power Architecture processors.

- Naming Conventions
- Specifying Source File Locations
- Licensing Command-Line Options
- Diagnostic Command-Line Options
- Library and Linking Command-Line Options
- Code Generation Command-Line Options
- Optimization Command-Line Options

11.1 Naming Conventions

Table 11-1 lists the names of the CodeWarrior command line tools.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>mwasmeppc</td>
<td>translates assembly language source code into object code</td>
</tr>
<tr>
<td>mwcceppc</td>
<td>translates C and C++ source code into object code</td>
</tr>
<tr>
<td>mwldeppc</td>
<td>links object code into a loadable image file</td>
</tr>
</tbody>
</table>

11.2 Specifying Source File Locations
The build tools use several environment variables at build time to search for include files, libraries, and other source files. All of the variables mentioned here are lists which are separated by semicolons (";" in Windows operating systems and colons (":") in Solaris operating systems.

Unless `-nodefaults` is passed to on the command line, the compiler searches for an environment variable called `MWCEABIPPCIncludes` or `MWCIncludes` (in that order). These variables contain a list of system access paths to be searched after the system access paths specified by the user. The assembler also does this, using the variables `MWAasmEABIPPCIncludes` or `MWAasmIncludes`.

Analogously, unless `-nodefaults` or `-disassemble` is given, the linker will search the environment for a list of system access paths and system library files to be added to the end of the search and link orders. The variable `MWEABIPPCLibraries` or `MWLibraries` contains a list of system library paths to search for files, libraries, and command files.

Associated with this list is the variable `MWEABIPPCLibraryFiles` or `MWLibraryFiles` which contains a list of libraries (or object files or command files) to add to the end of the link order. These files may be located in any of the cumulative access paths at runtime.

If you are only building for one target, it is okay to use `MWCIncludes`, `MWAasmIncludes`, `MWLibraries`, and `MWLibraryFiles`. The target-specific versions of the variables come in handy when targeting multiple targets, since the target-specific variables override the generic variables. Note that if the target-specific variable exists, the generic variable will not be used; the contents of the two variables will not be combined.

## 11.3 Licensing Command-Line Options

This chapter explains the following command-line options:

- `-fullLicenseSearch`
- `-license`

### 11.3.1 `-fullLicenseSearch`

Continues the search for a license file on the host computer.

**Syntax**
**Remarks**

A license file unlocks features and capabilities in CodeWarrior tools. This option extends the normal search for a valid `license.dat` file.

Each time they are invoked, the command-line compiler, stand-alone assembler, and linker search on the host computer in this order until they find a valid license file in this order:

- the directory specified in a `-license` option
- the directory containing the command-line tool
- the current working directory
- the directory containing the CodeWarrior IDE

When this option is not used, the tool stops when it finds a valid license file. With this option, the tool searches all paths to read all valid licenses.

### 11.3.2 `-license`

Specifies a location on the host computer to search for a license file.

**Syntax**

```
-license location
```

where `location` is the path of a directory that contains a valid license file named `license.dat`.

**Remarks**

A license file unlocks features and capabilities in CodeWarrior tools.

### 11.4 Diagnostic Command-Line Options

This chapter explains the following command-line options:
Diagnostic Command-Line Options

- **-g**
- **-gdwarf-2**
- **-fmt**
- **-format**
- **-listclosure**
- **-listdwarf**
- **-map**
- **-mapunused**
- **-sym**
- **-unused**

### 11.4.1 **-g**

Generates DWARF 1. `x`-conforming debugging information.

**Syntax**

```
-g[dwarf]
```

**Remarks**

This option is global. This option is equivalent to

```
-sym dwarf-1,full
```

### 11.4.2 **-gdwarf-2**

Generates DWARF-2. `x`-conforming debugging information.

**Syntax**

```
-gdwarf-2
```

**Remarks**

This option is global. This option is equivalent to
11.4.3 -fmt

Equivalent to the -format option.

Syntax

```
-fmt x | nox
```

11.4.4 -format

Specifies the style of mnemonics to show in disassemblies.

Syntax

```
-format x | nox
```

Remarks

To show extended mnemonics in a disassembly, use

```
-format x
```

This option is the default.

To show regular mnemonics in a disassembly, use

```
-format nox
```

This is a linker option.
11.4.5  -listclosure

Controls the appearance of symbol closures in the linker's map file.

Syntax

   -listclosure

   -nolistclosure

Remarks
This option also generates a map file if the -map option has not already been specified.
This is a linker option.

11.4.6  -listdwarf

Controls the appearance of DWARF debugging information in the linker's map file.

Syntax

   -listdwarf

   -nolistdwarf

Remarks
This option also generates a map file if the -map option has not already been specified.
This is a linker option.

11.4.7  -map

Generates a text file that describes the contents of the linker's output file.
Syntax

```
-map [filename]
```

Remarks
The default value for `filename` is the name of the linker's output file with a `.MAP` file name extension.
This is a linker option.

11.4.8  `-mapunused`

Controls the appearance of a list of unused symbols in the map file.

Syntax

```
-mapunused
-nomapunused
```

Remarks
This option also generates a map file if the `-map` option has not already been specified.
This is a linker option.

11.4.9  `-sym`

Specifies global debugging options.

Syntax

```
-sym
keyword[,...]
```

The choices for `keyword` are:
Library and Linking Command-Line Options

off

Do not generate debugging information. This option is the default.

on

Generate DWARF-1-conforming debugging information.

dwarf-1

Generate DWARF-1-conforming debugging information.

full[Oath]

Store absolute paths of source files instead of relative paths.

dwarf-2

Generate DWARF-2-conforming debugging information.

11.4.10 -unused

Equivalent to the -mapunused option.

Syntax

- unused

- nounused

11.5 Library and Linking Command-Line Options

This chapter explains the following command-line options:

- -codeaddr
- -ConvertArchiveToPartialLink
- -dataaddr
- -genbinary
- -gap_fill
- -heapaddr
• -heapsize
• -lcf
• -library
• -linkmode
• -main
• -model
• -noentry
• -nomain
• -opt_partial
• -partial
• -r
• -r1
• -r2
• -relocate
• -resolved_partial
• -sdataaddr
• -sdata2addr
• -sdathreshold
• -sdata2threshold
• -show
• -srec
• -srecol
• -sreclength
• -stackaddr
• -stacksize
• -strip_partial
• -tune_relocations
• -xtables
• -stdlib
• -L+
• -lr
• -l+
-codeaddr
addr

**Remarks**

The *addr* value is an address, in decimal or hexadecimal format. Hexadecimal values must begin with 0x. The default is 65536.

The linker ignores this option if you invoke the linker with the `-lcf` option.

This is a linker option.

### 11.5.2  -ConvertArchiveToPartialLink

Extracts all objects from the library files (.a) and then puts them into a partially linked file (plf).

**Syntax**

```
-ConvertArchiveToPartialLink
archives -o
filename
```

where *archives* is the list of archive files, and *filename* is the name of the output PLF file.

**Example**

```
$mwldeppc.exe -ConvertArchiveToPartialLink
MSL_C.PPCEABI.bare.E.UC.a Runtime.PPCEABI.E.UC.a -o
XXX.plf
```

**Remarks**

This linker command can be used for a project with only archive files (MSL C archive) as the project would normally generate an empty plf.

Use `-o` option to specify the name of the output PLF file. If `-o` option is not provided to the linker then the linker will generate the file with a default `a.out` filename.

While working with this linker command, if we link any object file (`*.o`), other than archive (`*.a`), then the output file (`*.plf`) will even contain the contents of linked object file, along with the usual archive contents.
It has been observed that all .plf files converted from the MSL archives have the .ctor and .dtor section. .plf files converted from Wii archives do not have the .ctor and .dtor section.

While working with the CodeWarrior IDE:
- the output file is set by default to *.elf extension, it should be changed to *.plf.
- the PLF also contains dwarf info because default dwarf info option is enabled.

### 11.5.3 -dataaddr

Sets the loading address of the data.

**Syntax**

```
-dataaddr
  addr
```

**Remarks**

The `addr` value is an address, in decimal or hexadecimal format. Hexadecimal values must begin with 0x. The default is the address after the code and large constant sections.

The linker ignores this option if the linker is invoked with the `-lcf` option.

This is a linker option.

### 11.5.4 -genbinary

Controls the generation of a binary file.

**Syntax**

```
-genbinary
  none
  one
  multiple
```

**Remarks**


To generate no binary file even if s-record generation is on, use

```
-genbinary none
```

This option is the default.

To generate a single binary file with all the loadable code and data, even if s-record generation is off, use

```
-genbinary one
```

To generate separate binary files for each MEMORY directive, even if s-record generation is off, use

```
-genbinary multiple
```

This is a linker option.

### 11.5.5 -gap_fill

Fills the gap in s-record file.

**Syntax**

```
-gap_fill
```

**Remarks**

This option fill the gaps caused by alignment.

### 11.5.6 -heapaddr

Sets the runtime address of the heap.

**Syntax**

```
-heapaddr
addr
```
Remarks

The \textit{addr} value is an address, in decimal or hexadecimal format. Hexadecimal values must begin with \texttt{0x}. The default is

\begin{verbatim}
stack_address - (heap_size + stack_size)
\end{verbatim}

where \textit{stack_address} is the address of the stack, \textit{heap_size} is the size of the heap, and \textit{stack_size} is the size of the stack.

This is a linker option.

\subsection*{11.5.7 \texttt{-heapsize}}

Sets the runtime size of the heap, in kilobytes.

\textbf{Syntax}

\begin{verbatim}
-heapsize size
\end{verbatim}

\textbf{Remarks}

The default value for \textit{size} is \texttt{1024}.

This is a linker option.

\subsection*{11.5.8 \texttt{-lcf}}

Uses the code and data addresses specified in a linker command file.

\textbf{Syntax}

\begin{verbatim}
-lcf filename
\end{verbatim}

\textbf{Remarks}
The filename argument is the name of a linker command file. The file must have a .lcf file name extension. The linker ignores the -codeaddr, -dataaddr, -sdataaddr, and -sdata2addr options if it uses the -lcf option.

This is a linker option.

11.5.9  -library

Generates a static library.

Syntax

-library

Remarks

This option is global. This is a linker option.

11.5.10  -linkmode

Controls the performance of the linker.

Syntax

-linkmode keyword

The choices for keyword are:

lessram
Use little memory but take more processing time.

normal
Use a medium amount of memory for medium processing time. This is the default.

moreram
Use lots of memory to improve processing time.
Remarks
This is a linker option.

11.5.11 -main

Specifies the main entry point for the executable image.

Syntax

```
-m[ain]
symbol
```

Remarks
The maximum length of `symbol` is 63 characters. The default is `__start`.
This is a linker option.

11.5.12 -model

Specifies the addressing mode that the linker uses when resolving references.

Syntax

```
-model
keyword
```

The choices for `keyword` are:

- `absolute`
  Use absolute executable and data addressing. This choice is the default.

- `sda_pic_pid`
  Use position-independent addressing executable code and data.

Remarks
This is a linker option.
11.5.13 -noentry

Specifies no entry point for the executable image.

Syntax

    -noentry

Remarks

The linker uses the main entry point to determine which objects/functions to add to your application that are referenced from that entry point. In absence of an entry point, the application will be empty (completely deadstripped) resulting in an linker error.

There are several ways to pass other entry points to the linker for objects that are not referenced from the main entry point.

    • use the linker command file directives TERM or INIT
    • use __declspec(export)
    • use the lcf directives FORCEFILES or FORCEACTIVE

For example, if you have a simple reset vector function which simply calls your startup code (call the startup code __start and __reset for the reset vector function for this example), you could do the following:

    • use -m __start at the command prompt
    • use ENTRY(__start) in the Linker Command File
    • use INIT(__reset) at the command prompt
    • use FORCEACTIVE(__reset) in the Linker Command File
    • use __declspec(export) void __reset(void) {__start;} in the source.

11.5.14 -nomain

Equivalent to -noentry.

Syntax

    -nomain
11.5.15  -opt_partial

Finishes a partial link operation.

Syntax

    -opt_partial

Remarks

This option allows the use of a linker command file, creates tables for C++ static constructors, C++ static destructors, and C++ exceptions. This option also tells the linker to build an executable image even if some symbols cannot be resolved.

This is a linker option.

11.5.16  -partial

Does not report error messages for unresolved symbols.

Syntax

    -partial

Remarks

This option tells the linker to build a reloadable object file even if some symbols cannot be resolved.

This is a linker option.

11.5.17  -r

Equivalent to -partial.

Syntax
-r

Remarks
This option tells the linker to build a reloadable object file even if some symbols cannot be resolved.
This is a linker option.

11.5.18  -r1

Equivalent to -opt_partial.

Syntax

-r1

Remarks
This option allows the use of a linker command file, creates tables for C++ static constructors, C++ static destructors, and C++ exceptions. This option tells the linker to build a reloadable object file even if some symbols cannot be resolved.
This is a linker option.

11.5.19  -r2

Equivalent to -resolved_partial.

Syntax

-r2

Remarks
This option first allows the use of a linker command file, creates tables for C++ static constructors, C++ static destructors, and C++ exceptions.
This is a linker option.
11.5.20  -relocate

Controls the use of relocations contained in DWARF 1.\text{x} sections when generating a disassembly of non-executable files.

**Syntax**

```
-relocate
```

```
-norelocate
```

**Remarks**

The `-relocate` option tells the linker to use data in the `.rela.text` and `.rela.debug` parts of the DWARF 1.x when generating disassemblies.

The `-relocate` option is the default when disassembling executable files with DWARF 1.\text{x} sections and any type of file with DWARF 2.\text{x} sections.

This is a linker option  

11.5.21  -resolved_partial

Finishes a partial link operation and issues error messages for unresolved symbols.

**Syntax**

```
-resolved_partial
```

**Remarks**

This option first allows the use of a linker command file, creates tables for C++ static constructors, C++ static destructors, and C++ exceptions.

This is a linker option
11.5.22  -sdataaddr

Sets the loading address of small data.

Syntax

-sdataaddr
  addr

Remarks

The \textit{addr} value is an address, in decimal or hexadecimal format. Hexadecimal values must begin with \texttt{0x}. The default is the address after the large data section.

The linker ignores this option if the linker is invoked with the \texttt{-lcf} option.

This is a linker option.

11.5.23  -sdata2addr

Sets the loading address of small constant data.

Syntax

-sdata2addr
  addr

Remarks

The \textit{addr} value is an address, in decimal or hexadecimal format. Hexadecimal values must begin with \texttt{0x}. The default is the address after the small data section.

The linker ignores this option if the linker is invoked with the \texttt{-lcf} option.

This is a linker option.

11.5.24  -sdatathreshold

Limits the size of the largest objects in the small data section.
Syntax

```
-sdata[threshold]
size
```

Remarks

The `size` value specifies the maximum size, in bytes, of all objects in the small data section (typically named ".sdata"). The linker places objects that are greater than this size in the data section (typically named ".data") instead.

You can override this option for a variable in your source code like this

```
__declspec(section ".sdata") extern int bigobj[25];
```

The default value for `size` is 8.

This is a linker option.

### 11.5.25  -sdata2threshold

Limits the size of the largest objects in the small constant data section.

Syntax

```
-sdata2[threshold]
size
```

Remarks

The `size` value specifies the maximum size, in bytes, of all objects in the small constant data section (typically named ".sdata2"). The linker places constant objects that are greater than this size in the constant data section (typically named ".rodata") instead.

You can override this option for a variable in your source code like this

```
__declspec(section ".sdata2") extern int bigobj[] =
{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

The default for `size` is 8.
This is a linker option.

11.5.26  -show

Specifies the information to list in a disassembly.

Syntax

```
-show
keyword[,\ldots]
```

The choices for `keyword` are:

- **only** | **none**
  Shows no disassembly. Begin a list of choices with only or none to prevent default information from appearing in the disassembly.

- **all**
  Shows binary, executable code, detailed, data, extended, and exception information in the disassembly.

- **binary** | **nobinary**
  Shows or does not show address and op-code values.

- **code** | **ncode**
  Shows or does not show executable code sections.

- **text** | **notext**
  Equivalent to the code and nocode choices, respectively.

- **data** | **ndata**
  Shows or does not show data sections.

- **detail** | **ndetail**
  Shows or does not show extra information.

- **extended** | **noextended**
  Shows or does not show extended mnemonics.
exceptions | noexceptions

Shows or does not show C++ exception tables. This option also shows data sections.

xtab[les] | noxtab[les]

Equivalent to the exceptions and noexceptions choices, respectively.

headers | noheaders

Shows or does not show object header information.

debug | nodebug

Shows or does not show debugging information.

dwarf | nodwarf

Equivalent to the debug and nodebug choices, respectively.

tables | notables

Shows or does not show character string and symbol tables.

source | nosource

Interleaves the code disassembly with c or c++ source code.

Remarks

The default setting for this option is

-show binary,code,data,extended,headers,tables

This is a linker option.

11.5.27 -srec

Generates an S-record file.

Syntax

-srec [ file-name]

Remarks

Chapter 11 Command-Line for Power Architecture Processors

CodeWarrior Development Studio for Power Architecture Processors Build Tools Reference Manual, Rev. 10.x, 06/2015

Freescale Semiconductor, Inc.
The default value for file-name is the name of the linker's output file with a .mot file name extension.

This is a linker option.

11.5.28 -sreceol

Specifies the end-of-line style to use in an S-record file.

Syntax

    -sreceol
    keyword

The choices for keyword are:

    mac
    Use Mac OS®-style end-of-line format.

    dos
    Use Microsoft® Windows®-style end-of-line format. This is the default choice.

    unix
    Use a UNIX-style end-of-line format.

Remarks

This option also generates an S-record file if the -srec option has not already been specified.

This is a linker option.

11.5.29 -sreclength

Specify the length of S-records.

Syntax
The choices for value are from 8 to 255. The default is 26.

**Remarks**
This option also generates an S-record file if the -srec option has not already been specified.
This is a linker option.

**11.5.30 -stackaddr**

Sets the runtime address of the stack.

**Syntax**

```
-stackaddr addr
```

**Remarks**
The addr value is an address, in decimal or hexadecimal format. Hexadecimal values must begin with 0x. The default is 0x3dff0.
This is a linker option.

**11.5.31 -stacksize**

Sets the runtime size of the stack, in kilobytes.

**Syntax**

```
-stacksize size
```

**Remarks**
The default value for size is 64.
11.5.32  -strip_partial

Removes unreferenced objects on a partially linked image.

Syntax

- strip_partial

Remarks

Use this option with either the -opt_partial or -resolved_partial options.

This is a linker option.

11.5.33  -tune_relocations

Ensures that references made by the linker conform to the PowerPC EABI (Embedded Application Binary Interface) or position-independent ABI (Application Binary Interface).

Syntax

- tune_relocations

Remarks

Use this option only with the -abi eabi and -abi sda_pic_pid option to ensure that references in the executable image conform to these ABIs.

To conform to both of these ABIs, the linker will modify relocations that do not reach the desired executable code. The linker first converts near branch instructions to far branch instructions. Then it will convert absolute branches to PC-relative branches. For branches that cannot be converted to far or PC-relative addressing, the linker will generate branch islands.

To conform to the SDA PIC/PID ABI, the linker will generate the appropriate style of addressing.
This option is global. This is a linker option.

11.5.34  -xtables

Equivalent to -show exceptions or -show noexceptions.

Syntax

-xtables on | off

Remarks

This is a linker option.

11.5.35  -stdlib

Uses standard system library access paths as specified by the environment variable

%MWLibraries% to add system libraries as specified by the environment variable

%MWLibraryFiles% at the end of link order.

Syntax

-stdlib

-nostdlib

Remarks

This command is global. This is a linker option.

11.5.36  -L+

Adds a new library search path to the default settings list.

Syntax
-L+path

-l path

The parameters are:

path

The search path to append.

Remarks

This command is global and case-sensitive.

11.5.37  -lr

Adds a recursive library search path to the default settings list.

Syntax

   -lr path

The parameters are:

path

The recursive library search path to append.

Remarks

This command is global. This is a linker option.

11.5.38  -l+

Adds a library by searching access path for a specified library filename.

Syntax

   -l+file
The parameters are:

`file`

Name of the library file to search.

**Remarks**

The linker searches access path for the specified lib<file>_<ext>, where <ext> is a typical library extension. If the file is not found then search for <file>. This command is case-sensitive.

### 11.6 Code Generation Command-Line Options

This chapter explains the following command-line options:

- `abi`
- `align`
- `altivec_move_block`
- `big`
- `common`
- `fatext`
- `fp`
- `fp_contract`
- `func_align`
- `gen-fsel`
- `little`
- `maf`
- `ordered-fp-compares`
- `pool`
- `processor`
- `profile`
- `ppc_asm_to_vle`
- `rambuffer`
- `readonlystrings`
- `relax_ieee`
- `romaddr`
- `rostr`
- `schedule`
- `spe_vector`
- `spe2_vector`
11.6.1   -abi

Chooses which ABI (Application Binary Interface) to conform to.

Syntax

\[-abi\]

\[\text{keyword}\]

The choices for \textit{keyword} are:

\texttt{eabi}

Use the Power Architecture Embedded ABI. This choice is the default.

\texttt{SysV}

Use the UNIX System V ABI without GNU extensions.

\texttt{SuSE}

Use the SuSE® Linux ABI with GNU extensions.

\texttt{YellowDog}

Use the Yellow Dog™ Linux ABI with GNU extensions.

\texttt{sda_pic_pid}

Use position-independent addressing executable code and data.

Remarks

This option is global.
11.6.2 -align

Specifies structure and array alignment.

Syntax

```
-align
keyword[,...]
```

The choices for `keyword` are:

- `power[pc]`
  Use conventional Power Architecture alignment. This choice is the default.

- `mac68k`
  Use conventional Mac OS® 68K alignment.

- `mac68k4byte`
  Use Mac OS® 68K 4-byte alignment.

- `array[members]`
  Align members of arrays, too.

11.6.3 -altivec_move_block

Controls the use of Altivec instructions to optimize block moves.

Syntax

```
-altivec_move_block

-noaltivec_move_block
```

Remarks

The default setting is `-noaltivec_move_block`. 
11.6.4  -big

Generates object code and links an executable image to use big-endian data formats.

Syntax

    -big

Remarks

This is the default setting for the compiler and linker.

11.6.5  -common

Moves uninitialized data into a common section.

Syntax

    -common on | off

Remarks

The default is off.

11.6.6  -fatext

Use eppc.o as the file name extension for object files.

Syntax

    -fatext

Remarks
Normally, the compiler generates object code files that have a file name extension of .o. This option tells the compiler to use eppc.o as a file name extension instead. If the compiler is invoked with this option and the compiler invokes the linker, the linker will search for object files that use the eppc.o file name extension.

11.6.7 -fp

Controls floating-point code generation.

Syntax

```
-fp
keyword
```

The choices for `keyword` are:

none | off

No floating point code generation.

soft[ware]

Use software libraries to perform floating-point operations. This is the default.

hard[ware]

Use the processor's built-in floating-point capabilities to perform floating-point operations.

dpfp

Use the processor's double-precision floating-point capabilities on the e500v2 processor.

spfp

Use software libraries for floating-point operations that use the `double` data type and use the e500 SPE-EFPU floating-point capabilities for other floating-point operations.

spfp_only

Use to have the compiler consider `double` and `long double` data types as floating point. This option is only supported for e200 (Zen or VLE) and e500v1 processors that support SPFP APU.
NOTE

When you downgrade from double data type to a floating point you will lose precision and range. If your expected numbers are within the range supported by a floating point data type, then this option might dramatically speed up and shrink your code. Do not use this option if you have instances in your project that depend on the size of a double data type.

fmadd

Equivalent to -fp hard -fp_contract.

Remarks

When using the -fp spfp_only option, the size of a double data type changes to a floating point data type, if you have existing code that is expecting to find certain bits at certain locations of the exponent or significand, then you will have to change that code to expect 4 byte doubles. Your code can make a test as shown in Example Test Code.

Listing: Example Test Code

```c
if (sizeof(double) == 4) {
    ...
} else {
    ...
}
```

The e500 and VLE library project files have targets and pre-built libraries (with SP in the name) that support this feature. Ensure you pick the right libraries to include in a project that supports this feature else you may call a function with a 8 byte double parameter and only pass a 4 byte double argument. The linker will report with a warning if you mix up the libraries - make sure you have linker warnings enabled.

If you have a library that doesn't use floating point, try setting it to none for the floating point model by using the -fp none option. Libraries with none floating point do not cause a warning when added to projects using another floating point model.

The sample code in Sample Code assumes that you are using the -fp spfp_only option and have included SP libraries. Your existing code makes a call to a MSL math function and a user defined function that takes a double argument and returns a double data type.

Listing: Sample Code

```c
#include <math.h>
extern double my_func(double);
extern double d1, d2;
void main()
```
Following can be observed while executing the sample code in Sample Code:

- `2.0` will be treated as a 4 byte `double` constant (exactly like `2.0f`).
- Storage for `d1` and `d2` will be 4 bytes each (exactly like floats).
- MSL will either inline or call a stub function for `pow` which will call `powf`.
- `my_func` will receive and return a 4 byte `double`. As long as `my_func` doesn’t do bit twiddling or require numbers not representable in a float, it will do its job correctly.

**NOTE**
If you are using a Zen processor and are using the `-fp spfp_only` option, ensure passing `-spe_addl_vector` instead of `-spe_vector` in order to have the compiler generate Multiply-Add instructions.

### 11.6.8 `-fp_contract`

Generates fused multiply-addition instructions.

**Syntax**

```
-fp_contract
```

**Remarks**
This option is the same as the `-maf` option.

### 11.6.9 `-func_align`

Specifies alignment of functions in executable code.

**Syntax**

```
Remarks

The default alignment is 4. However, at an optimization level 4, the alignment changes to 16. If you are using -func_align 4 (or none) and if you are compiling for VLE, then the linker will compress gaps between VLE functions:

- if those functions are not called by a Classic PPC function
- the function has an alignment greater than 4.

**NOTE**
Compression of the gaps will only happen on files compiled by the CodeWarrior compiler.

11.6.10  -gen-fsel

Deprecated. Use -use_fsel instead.

Syntax

- gen-fsel

- no-gen-fsel

11.6.11  -little

Generates object code and links an executable image to use little-endian data formats.

Syntax

- little
11.6.12  **-maf**

Controls the use of fused multiply-addition instructions.

**Syntax**

```bash
-maf on | off
```

**Remarks**

The `-maf on` option tells the compiler to generate fused multiply-addition operations instead of separate multiplication and addition instructions. The `-maf off` option tells the compiler to use separate multiplication and addition instructions.

11.6.13  **-ordered-fp-compares**

Controls the assumption of no unordered values in comparisons.

**Syntax**

```bash
-ordered-fp-compares
```

```bash
-no-ordered-fp-compares
```

**Remarks**

The default is `-no-ordered-fp-compares`.

11.6.14  **-pool**

Controls the grouping of similar-sized data objects.

**Syntax**

```bash
-pool[data] on | off
```
Remarks

Use this option to reduce the size of executable object code in functions that refer to many object of the same size. These similar-sized objects do not need to be of the same type. The compiler only applies this option to a function if the function refers to at least 3 similar-sized objects. The objects must be global or static.

At the beginning of the function, the compiler generates instructions to load the address of the first similar-sized object. The compiler then uses this address to generate 1 instruction for each subsequent reference to other similar-sized objects instead of the usual 2 instructions for loading an object using absolute addressing.

This option is equivalent to the pool_data pragma.

The default is on.

11.6.15 -processor

Generates and links object code for a specific processor.

Syntax

-processor
keyword

The choices for keyword are:

401 | 403 | 405 | 505 | 509 | 5100 | 5200 | 555 | 56x |
601 | 602 | 603 | 603e | 604 | 604e | 740 | 74x | 750 |
75x | 801 | 821 | 823 | 850 | 85x | 860 | 86x | 87x | 88x |
| 7400 | 744x | 7450 | 745x | 82xx | 8240 | 824x | 8260 |
827x | 8280 | 85xx | e300v1 | e300c1 | e300c2 | e300c3 |
e300c4 | e500v1 | e500v2 | e600 | Zen | 5565 | gekko |
generic

Remarks

The keyword parameter specifies the processor core. The default for keyword is generic.
To identify which core your target device uses, refer to the product page of the target device on the Freescale web site.

The e300v1 keyword is deprecated.
11.6.16  -profile

Controls the appearance of calls to a profiler library at the entry and exit points of each function.

Syntax

- profile on | off

Remarks

The default is off.

11.6.17  -ppc_asm_to_vle

Converts regular Power Architecture assembler mnemonics to equivalent VLE (Variable Length Encoded) assembler mnemonics in the inline assembler and standalone assembler.

Syntax

- ppc_asm_to_vle

Remarks

C/C++ source is input to compiler inline assembler where as assembly source is input to standalone assembler.

While translating assembly statements in C or C++ or assembly source code, the compiler inline assembler or standalone assembler will replace each regular Power Architecture assembler mnemonic with its matching VLE instruction if one exists. The mnemonics for VLE (Variable Length Encoding) instructions begin with se_ or e_. The compiler's inline assembler recognizes these mnemonics when the compiler is configured to generate VLE object code. The same is with assembler. If an equivalent VLE instruction does not exist for an UISA instruction then it might generate multiple VLE instructions for that operation.

Assembler changes the section .text to .text_vle. For user defined sections, LCF needs modification.
Example

Consider test.s

```assembly
.section .mytext
tmp:
do_loop00:
    B exit0
    Add r0, r0, r1
.text
exit0:
    blr
```

Assemble with option `-ppc_asm_to_vle`

In LCF

```assembly
.text_vle (VLECODE) : {
    *(.text)
    *(.text_vle)
    (.mytext)
}
```

**NOTE**

Assembler does not emit a 16-bit VLE instruction for branch instructions, if the label definition does not exist at that point.

### 11.6.18 `-rambuffer`

Specifies a runtime address in which to store the executable image in RAM so that it may be transferred to flash memory.

**Syntax**

```
-rambuffer
addr
```

**Remarks**

This option specifies information for a legacy flashing tool (some development boards that used the Power Architecture 821 processor). This tool required that the executable image must first be loaded to an area in RAM before being transferred to ROM. Do not use this option if your flash memory tool does not follow this behavior.

The CodeWarrior tools ignore this option if it is not used with the `-romaddr` option.
11.6.19  -readonlystrings

Places string constants in a read-only section.

Syntax

-readonlystrings

11.6.20  -relax_ieee

Controls the use of relaxed IEEE floating point operations.

Syntax

-relax_ieee

-norelax_ieee

Remarks

The default is -relax_ieee.

11.6.21  -romaddr

Generates a ROM image and specifies the image's starting address at runtime.

Syntax

-romaddr

 addr
11.6.22  -rostr

Equivalent to the -readonlystrings option.

Syntax

   -rostr

11.6.23  -schedule

Controls the rearrangement of instructions to reduce the effects of instruction latency.

Syntax

   -schedule on | off

Remarks
The default is off.

11.6.24  -spe_vector

Enables the SPE vector support.

Syntax

   -spe_vector

Remarks
This option needs to be enabled when the floating point is set to SPFP or DPFP as both SPFP and DPFP require support from the SPE vector unit. If the option is not turned on, the compiler generates a warning and automatically enables the SPE vector generation.
11.6.25 -spe2_vector

Enables the SPE2 vector support

Syntax

-spe2_vector

Remarks

In order to use the SPE2 intrinsics:

• Include `<spe.h>` in the source file.
• From the EPPC Processor settings panel:
  • Select Zen from the Processor list box.
  • Select spe2 from the vector list box.

NOTE
SPE2 instructions are supported in standalone assembler and compiler's inline assembler. These instructions are currently not fully validated, users must use them at their own risks.

11.6.26 -spe_addl_vector

Enables the additional SPE fused multiply-add and multiply-subtract instructions support.

Syntax

-spe_addl_vector

Remarks

The e200 z3 and z6 cores support 8 additional SPE fused multiply-add and multiply-subtract instructions. This option tells the compiler to generate the additional SPE instructions, when appropriate, for more optimized codes.

This option also turns on the -spe_vector option.
11.6.27 -strict_ieee

Specifies the use of strict IEEE floating point operations.

Syntax

-strict_ieee

Remarks
This option is the same as the -norelax_ieee option.

11.6.28 -use_lmw_stmw

Controls the use of multiple load and store instructions for function prologues and epilogues.

Syntax

-use_lmw_stmw on | off

Remarks
This option is only available for big-endian processors. This option is not available for big-endian e500v1 and e500v2 architectures when vector and double-precision floating-point instructions are used. The default is off.

11.6.29 -use_fsel

Controls the use of fsel instructions.

Syntax

-use_fsel on | off
Remarks
Do not turn on this option if the Power Architecture processor of your target platform does not have hardware floating-point capabilities that includes f sel. This option only has an effect if -relax_ieee is also specified on the command line. The default is off.

11.6.30  -use_isel

Controls the use of isel instructions.

Syntax
- use_isel on | off

Remarks
Do not turn on this option if the Power Architecture processor of your target platform does not implement the Freescale ISEL APU. The default is off.

11.6.31  -vector

Specifies AltiVec™ vector options.

Syntax
- vector
  keyword[,...]

The options for keyword are:
on
Generate AltiVec vectors and related instructions.
off
Do not generate AltiVec vectors and related instructions.
vrsave
Generate AltiVec vectors and instructions that use VRSAVE prologue and epilogue code.
Do not use VRSAVE code. This option is the default.

11.6.32  -vle

Controls the use of the Variable Length Encoded (VLE) instruction set.

Syntax

- vle

Remarks

This option tells the compiler and linker to generate and lay out Variable Length Encoded (VLE) instructions, available on Zen variants of Power Architecture processors. VLE instructions give extra flexibility in instruction encoding and alignment, allowing the compiler and linker to greatly reduce the size of runtime object code with only a small penalty in execution performance.

This option also turns on the -processor zen option.

11.7  Optimization Command-Line Options

This chapter explains the following command-line options:

- -code_merging
- -far_near_addressing
- -vle_bl_opt
- -vle_enhance_merging
- -volatileasm

11.7.1  -code_merging

Removes duplicated functions to reduce object code size.

Syntax
-code_merging keyword[,...]

The choices for keyword are:

all

Use the all argument to specify that the linker should remove all duplicate functions except one.

safe

Use the safe argument to specify that only duplicate functions marked as weak should be reduced to one function.

aggressive

Use the aggressive option to specify that the linker should ignore references to function addresses when considering which duplicate functions to remove.

off

Use the off argument to disable code merging optimization.

Remarks

This linker optimization removes duplicate copies of functions with identical executable code.

The linker does not apply this optimization to functions that have been declared with the __declspec(no_linker_opts) directive.

By default the code merging optimization is off.

11.7.2 -far_near_addressing

Simplifies address computations to reduce object code size and improve performance.

Syntax

-far_near_addressing
Remarks

This linker optimization simplifies address computations in object code. If an address value is within the range that can be stored in the immediate field of the load immediate instruction, the linker replaces the address's two-instruction computation with a single instruction. An address value that is outside this range still requires two instructions to compute.

The ranges of values that may be stored in the immediate field is \(-0x7fff\) to \(0x8000\) for the regular \(li\) instruction and \(-0x7ffff\) to \(0x80000\) for \(e_li\), the VLE (Variable Length Encoding) instruction.

The linker does not apply this optimization to functions that have been declared with the \_declspec(no_linker_opts) directive.

11.7.3 -vle_bl_opt

Replaces branch instructions to reduce object code size.

Syntax

-ble_bl_opt

-noble_bl_opt

Remarks

This linker optimization replaces each 32-bit \(e_bl\) instruction with a 16-bit \(se_bl\) instruction for a function call when the span of memory between the calling function and called function is sufficiently close.

This optimization requires that the target processor has the Variable Length Encoding (VLE) extension.

The linker does not apply this optimization to functions that have been declared with the \_declspec(no_linker_opts) directive.
11.7.4  -vle_enhance_merging

Removes duplicated functions that are called by functions that use VLE instructions to reduce object code size.

**Syntax**

```
-vle_enhance_merging
```

```
-novle_enhance_merging
```

**Remarks**

When applying the code merging optimization ( -code_merging), this linker optimization ensures that function calls that use VLE (Variable Length Encoding) instructions will still be able to reach a function that has been removed. This optimization replaces the 16-bit `se_bl` instruction with a 32-bit `e_bl` instruction.

When this option is not used, the linker does not merge functions that are called by functions that use VLE instructions.

This optimization requires that the target processor has the Variable Length Encoding (VLE) extension. This optimization has no effect when the linker is not applying the code merging optimization.

The linker does not apply this optimization to functions that have been declared with the `__declspec(no_linker_opts)` directive.

11.7.5  -volatileasm

Controls whether or not inline assembly statements will be optimized.

**Syntax**

```
-volatileasm
```
-novolatileasm
Chapter 12
Assembler

This chapter describes the assembler:

- Syntax
- Directives
- Macros
- GNU Compatibility

12.1 Syntax

This section describes the syntax of assembly language statements. It consists of these topics:

- Assembly Language Statements
- Statement Syntax
- Symbols
- Constants
- Expressions
- Comments
- Data Alignment

12.1.1 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 LWZ, Lwz, and lwz to be the same instruction.

Remember these rules for assembly language statements:

1. The maximum length of a statement or an expanded macro is 512 characters.
2. A statement must reside on a single line. However, you can concatenate two or more lines by typing a backslash (\) character at the end of the line.
3. 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.)

12.1.2 Statement Syntax

Statement Syntax shows the syntax of an assembly language statement. Table 12-1 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 12-1. 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>
</tbody>
</table>

Table continues on the next page...
12.1.3 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 symbols.

The syntax of a symbol is:

\[
\text{symbol ::= label | equate}
\]

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

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

However, it is possible for symbols to have a different scope, as described in the following sub-sections.

- Labels
- Non-Local Labels
- Local Labels
- Relocatable Labels
- Equates
- Case-Sensitive Identifiers

12.1.3.1 Labels

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

The syntax of a label is:
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. After you clear the checkbox, you may use labels that do not end with colons, but such labels must begin in column 1.

### 12.1.3.2 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 ($).

### 12.1.3.3 Local Labels

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

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

- letters (a-z or A-Z)
- numerals (0-9)
- underscores (_)
- question marks (?)
- dollar sign ($)
- periods (.)
NOTE
You cannot export local labels; local labels do not appear in debugging tables.

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

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

Local Label Scope in a Macro shows the scope of local labels in macros: the @SKIP label defined in the macro does not conflict with the @SKIP label defined in the main body of code.

Listing: Local Label Scope in a Macro

```
MAKEPOS .MACRO
  cmpwi           0,r3,0
  bge     @SKIP
  neg     r3,r3
@SKIP:                  ; Scope of this label is within
    ; the macro
  .ENDM

START:
  lwz     r3,COUNT
  cmpw   0,r3, r4
  beq     @SKIP
  MAKEPOS
@SKIP:                  ; Scope of this label is START to
    ; END excluding lines arising
    ; from macro expansion
  addic   r3,r3,1
END:    blr
```

12.1.3.4 Relocatable Labels

The assembler assumes a flat 32-bit memory space. You can use the expressions of Table 12-2 to specify the relocation of a 32-bit label.
NOTE

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

Table 12-2. 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. It also is the PC-relative target of a branch or call. It is a 32-bit address.</td>
</tr>
<tr>
<td>label@l</td>
<td>The low 16-bits of the relocated address of the symbol.</td>
</tr>
<tr>
<td>label@h</td>
<td>The high 16-bits of the relocated address of the symbol. You can OR this with label@l to produce the full 32-bit relocated address.</td>
</tr>
<tr>
<td>label@ha</td>
<td>The adjusted high 16-bits of the relocated address of the symbol. You can add this to label@l to produce the full 32-bit relocated address.</td>
</tr>
<tr>
<td>label@sdax</td>
<td>For labels in a small data section, the offset from the base of the small data section to the label. This syntax is not allowed for labels in other sections.</td>
</tr>
<tr>
<td>label@got</td>
<td>For 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>

12.1.3.5 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.

The code of **Valid Forward Equate** shows a valid forward equate.

**Listing: Valid Forward Equate**

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

However, the code of **Invalid Forward Equate** is not valid. The assembler cannot immediately resolve the expression in the `.space` directive, so the effect on the location counter is unknown.

**Listing: Invalid Forward Equate**

```
;invalid forward equate on next line
rec_size .set table_start-table_end
 .space rec_size
 .text; ...
table_start:
 ; ...
table_end:
```
12.1.3.6 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.)

12.1.4 Constants

The assembler recognizes three kinds of constants:

- **Integer Constants**
- **Floating-Point Constants**
- **Character Constants**

12.1.4.1 Integer Constants

Table 12-3 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>$ followed by string of hexadecimal digits, such as $deadbeef.</td>
<td>0x followed by a string of hexadecimal digits, such as 0xdeadbeef.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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>String of binary digits followed by b, such as 01010001b.</td>
</tr>
</tbody>
</table>
NOTE
The assembler uses 32-bit signed arithmetic to store and manipulate integer constants.

12.1.4.2 Floating-Point Constants

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

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

12.1.4.3 Character Constants

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

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

The maximum width of a character constant is 4 characters, depending on the context. Examples are 'A', 'ABC', and 'TEXT'.

A character constant can contain any of the escape sequences that Table 12-4 lists.

<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>&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>
</tbody>
</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.
12.1.5 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 (*).

Table 12-5 lists the expression operators that the assembler supports.

<table>
<thead>
<tr>
<th>Category</th>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>+</td>
<td>add</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>subtract</td>
</tr>
<tr>
<td></td>
<td>*</td>
<td>multiply</td>
</tr>
<tr>
<td></td>
<td>/</td>
<td>divide</td>
</tr>
<tr>
<td></td>
<td>%</td>
<td>modulo</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&amp;&amp;</td>
<td>logical AND</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&amp;</td>
<td>bitwise AND</td>
</tr>
<tr>
<td></td>
<td>^</td>
<td>bitwise XOR</td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;</td>
<td>shift left</td>
</tr>
<tr>
<td></td>
<td>&gt;&gt;</td>
<td>shift right (zeros are shifted into high order bits)</td>
</tr>
<tr>
<td></td>
<td>==</td>
<td>equal to</td>
</tr>
<tr>
<td></td>
<td>!=</td>
<td>not equal to</td>
</tr>
<tr>
<td></td>
<td>&lt;=</td>
<td>less than or equal to</td>
</tr>
<tr>
<td></td>
<td>&gt;=</td>
<td>greater than or equal to</td>
</tr>
<tr>
<td></td>
<td>&gt;</td>
<td>greater than or equal to</td>
</tr>
<tr>
<td></td>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td></td>
<td>@</td>
<td>at</td>
</tr>
<tr>
<td>Unary</td>
<td>+</td>
<td>unary plus</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>unary minus</td>
</tr>
<tr>
<td></td>
<td>~</td>
<td>unary bitwise complement</td>
</tr>
</tbody>
</table>

*Table continues on the next page...*
Table 12-5. Expression Operators (continued)

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

Operator precedence is:

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

12.1.6 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. or /* This is a comment. */
2. Start the comment with an asterisk (*) in the first column of the line.

   *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. 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.
4. Anything following a # character is considered to be a comment. For example, st r3,0(r4) # Store total
5. Anything following a ; character is considered to be a comment, except in GNU compatibility mode, where ; is a statement separator.

12.1.7 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.

12.2 Directives

• Some directives may not be available for the assembler for your target processor.
• The default starting character for most directives is the period (.). However, if you clear the Directives begin with '.' checkbox of the Assembler settings panel, you can omit the period.
• You can use the C/C++ preprocessor format to specify several preprocessor directives.

Explanations are in these sections:

• Macro Directives
• Conditional Preprocessor Directives
• Section Control Directives
• Scope Control Directives
• Symbol Definition Directives
• Data Declaration Directives
• Assembler Control Directives
• Debugging Directives

12.2.1 Macro Directives

These directives let you create macros:
• macro
• endm
• mexit
• #define

For more information on macros, see "Macros".

12.2.1.1 macro

Starts the definition of a macro.

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

Parameters

    label
Name you give the macro.

    parameter
Optional parameter for the macro.

12.2.1.2 endm

Ends the definition of a macro.

    .endm

12.2.1.3 mexit

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

    .mexit
12.2.1.4  #define

Defines a C pre-processor macro with the specified parameters. Note that the C pre-
processor is run on the assembler file before normal assembly. C pre-processor macros
should not be confused with normal macros declared using the MACRO and ENDM directives.

```
#define name [ {
  parms} ]
assembly_statement [ ; ] [ \ ]

assembly_statement [ ; ] [ \ ]

assembly_statement

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

Parameters

name

Name you give the macro.

parms

List of parameters, separated by commas.

assembly_statement

Any valid assembly statement.

Remarks

To extend an assembly_statement, type a backslash (\) and continue the statement on the
next line. To specify multiple assembly statements in the macro, type a semicolon and
backslash (;\), then type a new assembly statement on the next line. If the assembler is in
GNU mode, multiple statements can be on one line of code - separate them with
semicolon characters (;).
12.2.2 Conditional Preprocessor Directives

Conditional directives let you control whether compilation includes a block of code. These directives let you make multiple builds that are slightly different.

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

NOTE
You can use the C/C++ preprocessor format to specify these conditional directives:

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

With two exceptions, these directives function identically whether their starting character is a pound sign (#) or a period. The exceptions are:

1. You cannot use the pound sign format in a macro.
2. The period form of `elif` is `.elseif`.

The conditional preprocessor directives are:

- `if`
- `ifdef`
- `ifndef`
- `ifc`
- `ifnc`
- `endif`
- `elseif`
- `else`
- `Compatibility Conditional Directives`
12.2.2.1 if

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

.if bool-expr

Parameter

bool-expr

Any boolean expression.

Remarks

If bool-expr is true, the assembler processes the statements of the block. If bool-expr is false, the assembler skips the statements of the block.

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

12.2.2.2 ifdef

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

#ifndef symbol

Parameter

symbol

Any valid symbol.

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.
12.2.2.3  ifndef

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

    .ifndef symbol

**Parameter**

*symbol*

Any valid symbol.

**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.

12.2.2.4  ifc

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

    .ifc string1, string2

**Parameters**

*string1*

Any valid string.

*string2*

Any valid string.

**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.

### 12.2.2.5  `.ifnc`

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

```
.ifnc string1, string2
```

**Parameters**

- `string1`
  - Any valid string.
- `string2`
  - Any valid string.

**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.

### 12.2.2.6  `.endif`

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

```
.endif
```

### 12.2.2.7  `.elseif`

CodeWarrior Development Studio for Power Architecture Processors Build Tools Reference Manual, Rev. 10.x, 06/2015

Freescale Semiconductor, Inc.
Starts an alternative conditional assembly block, making assembly conditional on the truth of a boolean expression.

```
.elseif bool-expr
```

**Parameter**

**bool-expr**

Any boolean expression.

**Remarks**

If `bool-expr` is true, the assembler processes the statements of the block. If `bool-expr` is false, the assembler skips the statements of the block.

You can use this directive to create a logical, multilevel *if-then-else* statement, according to this syntax:

```
.if bool-expr statement-group

[ .elseif bool-expr statement-group ]...

[ .else statement-group ]

.endif
```

(In this syntax, `statement-group` is any group of assembly-language statements.)

The `.elseif` directive can be part of more complicated logical structures, such as:

```
.if bool-expr-1

    statement-group-1

.elseif bool-expr-2
```
• If this structure's bool-expr-1 is true, the assembler executes the statement-group-1 assembly-language 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.)

12.2.2.8 else
Starts an alternative conditional assembly block.

```
.else
```

**Remarks**

This directive is optional. The assembler processes the statements of the alternative conditional assembly block only if the expressions for an `.if` directive and any associated `.elseif` directives are false.

### 12.2.3 Compatibility Conditional Directives

For compatibility with other assemblers, the assembler supports these additional conditional directives:

- `.ifeq` if equal
- `.ifne` if not equal
- `.iflt` if less than
- `.ifle` if less than or equal
- `.ifgt` if greater than
- `.ifge` if greater than or equal

#### 12.2.3.1 `.ifeq` if equal

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

```
.ifeq string
```

**Parameter**

`string`

Any valid string.

**Remarks**

If the `string` value equals 0, the assembler processes the statements of the block. If the `string` value does *not* equal 0, the assembler skips the statements of the block.
12.2.3.2 .ifne if not equal

Starts a conditional assembly block, making assembly conditional on a string value \textit{not} being equal to zero.

\texttt{.ifne string}

**Parameter**

\texttt{string}

Any valid string.

**Remarks**

If the \texttt{string} value is \textit{not} equal to 0, the assembler processes the statements of the block. If the \texttt{string} value \textit{does} equal 0, the assembler skips the statements of the block.

12.2.3.3 .iflt if less than

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

\texttt{.iflt string}

**Parameter**

\texttt{string}

Any valid string.

**Remarks**

If the \texttt{string} value is less than 0, the assembler processes the statements of the block. If the \texttt{string} value equals or exceeds 0, the assembler skips the statements of the block.

12.2.3.4 .ifle if less than or equal
Starts a conditional assembly block, making assembly conditional on a string value being less than or equal to zero.

.ifle string

Parameter

string

Any valid string.

Remarks

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

12.2.3.5 .ifgt if greater than

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

.ifgt string

Parameter

string

Any valid string.

Remarks

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

12.2.3.6 .ifge if greater than or equal

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

**Parameter**

string

Any valid string.

**Remarks**

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

### 12.2.4 Section Control Directives

These directives identify the different sections of an assembly file:

- text
- data
- rodata
- bss
- sdata
- sdata2
- sbss
- text_vle
- debug
- previous
- offset
- section

### 12.2.4.1 text

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

.text
12.2.4.2  data

Specifies an initialized read-write data section.

.data

12.2.4.3  rodata

Specifies an initialized read-only data section.

.rodata

12.2.4.4  bss

Specifies an uninitialized read-write data section.

.bss

12.2.4.5  sdata

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

.sdata

12.2.4.6  sdata2

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

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

.sdata2

12.2.4.8  text_vle

Specifies a Variable length encoded section as read/execute.

text_vle

12.2.4.9  debug

Specifies a debug section.

d.debug

Remarks

If you enable the debugger, the assembler automatically generates some debug information for your project. However, you can use special directives in the debug section that provide the debugger with more detailed information. For more information on the debug directives, see "Debugging Directives".

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

```
.previous
```

### 12.2.4.11 offset

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

```
.offset [expression]
```

**Parameter**

**expression**

Optional initial location-counter value.

**Remarks**

Table 12-6 lists the only directives you can use inside a record.

#### Table 12-6. Directives within a Record

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

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

**Example**

Figure 12-1 shows a sample record definition.

#### Figure 12-1. Record Definition with Offset Directive

```
top:
    .offset
    .short 0

left:
    .short 0

bottom:
    .short 0

right:
    .short 0
```
12.2.4.12 section

Defines a section of an ELF (Executable and Linkable Format) object file.

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

**Parameters**

name

Name of the section.

alignment

Alignment boundary.

type

Numeric value for the ELF section type, per Table 12-7. The default type value is 1: (SHT_PROGBITS).

flags

Numeric value for the ELF section flags, per Table 12-8. The default flags value is 0x00000002, 0x00000001: (SHF_ALLOC+SHF_WRITE).

### Table 12-7. 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>
<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>
</tbody>
</table>

*Table continues on the next page...*
Table 12-7. ELF Section Header Types (SHT) (continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<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 12-8. 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>

**Remark**

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

**Possible syntax forms**

The section directive accepts a number of different syntax forms, partly for convenience and partly for compatibility with other assemblers. A section declaration requires four pieces of information: a section name, alignment, ELF section type (for example, SHT_PROGBITS) and ELF section flags (for example, SHF_ALLOC + SHF_EXECINSTR).

The possible syntax forms are as follows:

- Specify built-in section name.

  .section text

  This example specifies a built-in section name text. Equivalently, .text is also a valid syntax form.
Table 12-9 provides a list of all the possible values, together with their ELF types and ELF Section Header Flags.

### Table 12-9. Built-in Section names with their ELF Types and Flags

<table>
<thead>
<tr>
<th>Name</th>
<th>ELF Type</th>
<th>ELF Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text</td>
<td>SHT_PROGBITS</td>
<td>SHF_ALLOC +SHF_EXECINSTR</td>
</tr>
<tr>
<td>.data</td>
<td>SHT_PROGBITS</td>
<td>SHF_ALLOC+SHF_WRITE</td>
</tr>
<tr>
<td>.rodata</td>
<td>SHT_PROGBITS</td>
<td>SHF_ALLOC</td>
</tr>
<tr>
<td>.bss</td>
<td>SHT_NOBITS</td>
<td>SHF_ALLOC+SHF_WRITE</td>
</tr>
<tr>
<td>.sdata</td>
<td>SHT_PROGBITS</td>
<td>SHF_ALLOC+SHF_WRITE</td>
</tr>
<tr>
<td>.sdata0</td>
<td>SHT_PROGBITS</td>
<td>SHF_ALLOC+SHF_WRITE</td>
</tr>
<tr>
<td>.sdata2</td>
<td>SHT_PROGBITS</td>
<td>SHF_ALLOC</td>
</tr>
<tr>
<td>.sbss</td>
<td>SHT_NOBITS</td>
<td>SHF_ALLOC+SHF_WRITE</td>
</tr>
<tr>
<td>.sbss0</td>
<td>SHT_NOBITS</td>
<td>SHF_ALLOC+SHF_WRITE</td>
</tr>
<tr>
<td>.sbss2</td>
<td>SHT_PROGBITS</td>
<td>SHF_ALLOC</td>
</tr>
<tr>
<td>.debug</td>
<td>SHT_PROGBITS</td>
<td>0</td>
</tr>
<tr>
<td>.text_vle</td>
<td>SHT_PROGBITS</td>
<td>SHF_ALLOC +SHF_EXECINSTR +SHF_PE_EXECINSTR</td>
</tr>
<tr>
<td>.PPC.EMB.sdata0</td>
<td>SHT_PROGBITS</td>
<td>SHF_ALLOC+SHF_WRITE</td>
</tr>
<tr>
<td>.PPC.EMB.sbss0</td>
<td>SHT_PROGBITS</td>
<td>SHF_ALLOC+SHF_WRITE</td>
</tr>
</tbody>
</table>

In general, .text is for instructions, .data for initialised data, .rodata for read-only data (constants) and .bss for uninitialised data. The additional forms like .sdata are for small data areas. The built-in section names are architecture-specific, and are intended to make access to data more efficient. The alignment used for these sections is architecture-specific and is usually 4.

- Specify your own section name but get the attributes of the built-in section type.
  .section mySection,text
  
  This example is equivalent to writing .text except that the section will be called mySection.

- Specify the alignment, and optionally the ELF type and flags.
  .section name [,alignment [,type [,flags]]]
  
  In the syntax above, if the alignment is not specified it defaults to 16. If the type or flags are not specified, the defaults are as follows:
  
  - If the name parameter is a built-in section name, the type and the flags are taken as specified in the Table 12-9. For example, in the syntax form .section text,8 the
type is SHT_PROGBITS and the flags value is SHF_ALLOC +SHF_EXEandinstr.

- In all other cases, the default type is SHT_PROGBITS and the default flags value is SHF_ALLOC+SHF_WRITE, corresponding to a writeable data section.
- Specify the type and flags parameters in pre-defined characters, optionally in double quotes.

```
.section mySection,4,"rx" or .section mySection,4,rx
```

The values are additive. For example, rx is equivalent to SHF_ALLOC +SHF_WRITE+SHF_EXEandinstr

**NOTE**

If the syntax doesn't specify a type it defaults to SHT_PROGBITS

Table 12-10 provides a list of all the possible characters and their corresponding ELF Type and ELF Flags.

**Table 12-10. Characters and their corresponding ELF Type and ELF Flags**

<table>
<thead>
<tr>
<th>Character</th>
<th>ELF Type</th>
<th>ELF Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>SHT_NOBITS</td>
<td>SHF_ALLOC+SHF_WRITE</td>
</tr>
<tr>
<td>c</td>
<td>SHT_PROGBITS</td>
<td>SHF_ALLOC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SHF_EXECINSTR</td>
</tr>
<tr>
<td>d</td>
<td>SHT_PROGBITS</td>
<td>SHF_ALLOC+SHF_WRITE</td>
</tr>
<tr>
<td>m</td>
<td>SHT_PROGBITS</td>
<td>SHF_ALLOC+SHF_WRITE+SHF_EXECINSTR</td>
</tr>
<tr>
<td>r</td>
<td>0</td>
<td>SHF_ALLOC</td>
</tr>
<tr>
<td>w</td>
<td>0</td>
<td>SHF_ALLOC+SHF_WRITE</td>
</tr>
<tr>
<td>x</td>
<td>0</td>
<td>SHF_ALLOC+SHF_EXECINSTR</td>
</tr>
</tbody>
</table>

**12.2.5 Scope Control Directives**

These directives let you import and export labels:

- `global`
- `extern`
- `public`

For more information on labels, see "Labels". 
NOTE
You cannot import or export equates or local labels.

12.2.5.1 global

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

.globa l label [ ,label ]...

Parameter

label

Any valid label.

12.2.5.2 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.

12.2.5.3 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).

12.2.6 Symbol Definition Directives

These directives let you create equates:

- set
- equal sign (=)
- equ
- textequ

12.2.6.1 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.

12.2.6.2 equal sign (=)

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

**Parameters**

equate

Name of the equate.

expression

Temporary initial value for the equate.

**Remarks**

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

### 12.2.6.3 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.

### 12.2.6.4 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"
```
12.2.7.1.1  byte

Declarations an initialized block of bytes.

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

Parameters

label
Name of the block of bytes.

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

12.2.7.1.2  fill

Declarations a block of bytes, initializing each byte to zero.

```
[ label ] .fill expression
```

Parameters

label
Name of the block of bytes.

expression
Number of bytes in the block.

12.2.7.1.3  incbin

Tells the assembler to include the binary content of the specified file.
.incbin filename[,start[,length]]

**Parameters**

**filename**
Name of a binary file.

**start**
Offset from start of file at which to begin including data.

**length**
Number of bytes of data to include.

**Remarks**
This directive is useful when you wish to include arbitrary data directly in the program being assembled, e.g.

```
logopicture: .incbin "CompanyLogo.jpg"
```

### 12.2.7.1.4 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.

### 12.2.7.1.5 short

Declares an initialized block of 16-bit short integers.
[ label ] .short expression [ ,expression ]...

Parameters

label
Name of the block of integers.

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

12.2.7.1.6 space

Declares a block of bytes, initializing each byte to zero.

[ label ] .space expression

Parameters

label
Name of the block of bytes.

eexpression
Number of bytes in the block.

12.2.7.2 String Directives

These directives initialize blocks of character data:

• ascii
• ascz

A string can contain any of the escape sequences Table 12-11 lists.

Table 12-11. Escape sequences

<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>
</tbody>
</table>

Table continues on the next page...
### Table 12-11. Escape sequences (continued)

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

#### 12.2.7.2.1 ascii

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

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

**Parameters**

- **label**
  
  Name of the storage block.

- **string**
  
  String value to be stored, in double quotes.

#### 12.2.7.2.2 asciz

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

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

**Parameters**

- **label**
  
  Name of the storage block.

- **string**
  
  String value to be stored, in double quotes.

**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.

12.2.7.3 Floating-Point Directives

These directives initialize blocks of floating-point data:

- **float**
- **double**

12.2.7.3.1 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.

12.2.7.3.2 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.
Floating-point value; must fit into 64 bits.

### 12.2.8 Assembler Control Directives

These directives let you control code assembly:

- **align**
- **endian**
- **error**
- **include**
- **pragma**
- **org**
- **option**

#### 12.2.8.1 align

Aligns the location counter to the next multiple of the specified power of 2.

```
.align expression
```

**Parameter**

**expression**

Power of 2, such as 2, 4, 8, 16, or 32.

#### 12.2.8.2 endian

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

```
.endian big | little
```

**Parameters**
big

Big-endian specifier.

little

Little-endian specifier.

12.2.8.3  error

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

.error "error"

Parameter

error

Error message, in double quotes.

12.2.8.4  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.
12.2.8.5  **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.

12.2.8.6  **org**

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

```
.org expression
```

**Parameter**

expression

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

**Remarks**

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

**Example**

In [Listing 12-1 on page 195](#), 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 12-1. Address-Change Example

.text
.org 0x1000
```
NOTE
You must use the CodeWarrior IDE and linker to place code at an absolute address.

12.2.8.7  option

Sets an assembler control option as Table 12-12 describes.

.option keyword setting

Parameters

keyword

Control option.

setting

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

Table 12-12. Option Keywords

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

Table continues on the next page...
### Table 12-12. Option Keywords (continued)

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>period off/on/reset</td>
<td>Controls period usage for directives. The ON setting means that each directive must start with a period. Corresponds to the Directives begin with '.' checkbox of the Assembler settings panel.</td>
</tr>
<tr>
<td>reorder off/on/reset</td>
<td>Controls NOP instructions after jumps and branches. The ON setting means that the assembler inserts a NOP instruction, possibly preventing pipeline problems. The OFF setting means that the assembler does not insert a NOP instruction, so that you can specify a different instruction after jumps and branches. Applies only to the MIPS Assembler.</td>
</tr>
<tr>
<td>space off/on/reset</td>
<td>Controls spaces in operand fields. The OFF setting means that a space in an operand field starts a comment. Corresponds to the Allow space in operand field checkbox of the Assembler settings panel.</td>
</tr>
</tbody>
</table>

### 12.2.9 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:

- `file`
- `function`
- `line`
- `size`
- `type`

**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; you use the Project window to enable this debugging.

### 12.2.9.1 file

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

filename

Name of source-code file, in double quotes.

Remarks

Must precede other debugging directives in the assembly file. If you write your own DWARF code, you must use the .function and .line directives as well as this .file directive.

Example

Listing 12-2 on page 198 shows how to use the .file directive for your own DWARF code.

Listing 12-2. DWARF Code Example

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

12.2.9.2 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.

12.2.9.3  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.

12.2.9.4  size

Specifies a length for a symbol.

.size  symbol, expression

Parameters

symbol

Symbol name.

expression

Number of bytes.
12.2.9.5 type

Specifies the type of a symbol.

.type symbol, @function | @object

Parameters
	symbol
Symbol name.

@function
Function type specifier.

@object
Variable specifier.

12.3 Macros

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

This chapter includes these topics:

• Defining Macros
• Invoking Macros

12.3.1 Defining Macros

This section explains how to define macros.

• Macro Definition Syntax
• Using Macro Arguments
• Creating Unique Labels and Equates
• Referring to the Number of Arguments
12.3.1.1 Macro Definition Syntax

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

- the name of a macro
- the format of the macro call
- the assembly statements of the macro

To define a macro, use either the `macro` or the `#define` 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.)

12.3.1.1.1 Using the .macro Directive

The `.macro` directive is part of the first line of a macro definition. Every macro definition ends with the `.endm` directive. Listing 12-3 on page 201 shows the full syntax, and Table 12-13 explains the syntax elements.


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

<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 us 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.

Listing 12-4 on page 202 is the definition of macro addto, which includes an .mexit directive. Listing 12-5 on page 202 shows the assembly-language code that calls the addto macro. Listing 12-6 on page 202 shows the expanded addto macro calls.

**Listing 12-4. Conditional Macro Definition**

```assembly
#define a macro
doto: .macro dest,val
    .if val==0
    nop
    .elseif val >= -32768 && val <= 32767
    addi dest,dest,val      // use compact instruction
    .else
    addi dest,dest,val@l    // use 32-bit add
    addis dest,dest,val@ha
    .endif
// end macro definition
.endm
```

**Listing 12-5. Assembly Code that Calls addto Macro**

```assembly
// specify an executable code section
.text
li      r3,0
// call the addto macro
doto r3,0
doto r3,1
doto r3,2
doto r3,0x12345678
```

**Listing 12-6. Expanded addto Macro Calls**

```assembly
li      r3,0
nop
addi    r3,r3,1
addi    r3,r3,2
addi    r3,r3,0x12345678@l
```
12.3.1.2 Using Macro Arguments

You can refer to parameters directly by name. Listing 12-7 on page 203 shows the setup macro, which moves an integer into a register and branches to the label _final_setup. Listing 12-8 on page 203 shows a way to invoke the setup macro., and Listing 12-9 on page 203 shows how the assembler expands the setup macro.

**Listing 12-7. Setup Macro Definition**

```assembly
setup: .macro name
    li r3,name
    bl _final_setup
.endm
```

**Listing 12-8. Calling Setup Macro**

```assembly
VECT: .equ 0
setup      VECT
```

**Listing 12-9. Expanding Setup Macro**

```assembly
li   r3,VECT
bl   _final_setup
```

If you refer to named macro parameters in the macro body, you can precede or follow the macro parameter with &&. This lets you embed the parameter in a string. For example, Listing 12-10 on page 203 shows the smallnum macro, which creates a small float by appending the string E-20 to the macro argument. Listing 12-11 on page 203 shows a way to invoke the smallnum macro, and Listing 12-12 on page 204 shows how the assembler expands the smallnum macro.

**Listing 12-10. Smallnum Macro Definition**

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

**Listing 12-11. Invoking Smallnum Macro**

```assembly
smallnum  10
```
12.3.1.3 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.

Listing 12-13 on page 204 shows a macro that uses unique labels and equates. Listing 12-14 on page 204 shows two calls to the putstr macro. Listing 12-15 on page 204 shows the expanded code after the two calls.

Listing 12-13. Unique Label Macro Definition

```
putstr: .macro  string
    lis     r3,(str@)@h
    oris    r3,r3,(str@)@l
    bl      put_string
    b       skip@
str@:  .asciz  string
    .align  4
skip@
    .endm
```

Listing 12-14. Invoking putstr Macro

```
putstr 'SuperSoft Version 1.3'
putstr 'Initializing...'
```

Listing 12-15. Expanding putstr Calls

```
lis     r3,(str??0000)@h
oris    r3,r3,(str??0000)@l
```
12.3.1.4 Referring to the Number of Arguments

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

12.3.2 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, Listing 12-16 on page 205 shows macro `pattern`, which repeats a pattern of bytes passed to it the number of times specified in the macro call. Listing 12-17 on page 206 shows a statement that calls `pattern`, passing a parameter that includes a comma. Listing 12-18 on page 206 is another example calling statement; the assembler generates the same code in response to the calling statement of either Listing 12-17 on page 206 or Listing 12-18 on page 206.

Listing 12-16. Pattern Macro Definition

```
pattern:   .macro times,bytes
            .rept times
            .byte bytes
            .endr
```
Listing 12-17. Macro Argument with Commas

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

Listing 12-18. Alternate Byte-Pattern Method

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

12.3.2.1 Using the #define Directive

Another way to define a macro is to use the #define directive. This will define a pre-
processor style macro using a syntax that will be familiar to C programmers. Note that C
pre-processor macros are complementary to the assembler's main MACRO...ENDM macro
language.

Listing 12-19 on page 206 shows the full syntax, and explains the syntax elements.


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

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

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

Table 12-14. Syntax Elements: #define Directive

<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.</td>
</tr>
</tbody>
</table>
| assembly_statement  | An assembly language statement. To extend the statement beyond the length of
|                     | one physical line, type a backslash (\) at the end of a line, then continue
|                     | the statement on the next line.                                           |
|                     | To specify multiple statements on the same line, separate them with semi-
|                     | colon and backslash characters (;
|                     | ).                                                                         |
12.4 GNU Compatibility

The CodeWarrior Assembler supports several GNU-format assembly language extensions.

• GNU Compatible Syntax option
• Supported Extensions
• Unsupported Extensions

12.4.1 GNU Compatible Syntax option

Only in cases where GNU's assembler format conflicts with that of the CodeWarrior assembler does the GNU Compatible Syntax option have any effect. Specifically:

• Defining Equates
  Whether defined using .equ or .set, all equates can be re-defined.

• Ignored directives
  The .type directive ignored.

• Undefined Symbols
  Undefined symbols are automatically treated as imported

• 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

### 12.4.2 Supported Extensions

Some GNU extensions are always available, regardless whether you enable GNU compatible syntax. Specifically:

- 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 may begin with 0b.
- Supports the GNU macro language, with macros defined by:

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

Arguments may have default values as shown, and when called may be specified by value or position. See the GNU documentation for details.

- New or enhanced directives (see GNU documentation for details)

<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>
</tbody>
</table>

*Table continues on the next page...*
Table 12-15. Supported GNU Assembler Directives
(continued)

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<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>.comm 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>See .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 'size', value '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>.ifndef name</td>
<td>Conditional</td>
<td>Accepted but ignored</td>
</tr>
<tr>
<td>.include name</td>
<td>Include file</td>
<td>Now accepts single, double or no quotes</td>
</tr>
<tr>
<td>.int 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>.lflags</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 ot be specified</td>
</tr>
<tr>
<td>.p2align[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>
<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>
</tbody>
</table>

Table continues on the next page...
### 12.4.3 Unsupported Extensions

Among the GNU extensions that the CodeWarrior Assembler does not support are:

- **Sub-sections** (such as "\.text 2"). The sub-section number will be ignored.

  As a workaround, you can create your own sections with the `.section <name>` directive. You may have an arbitrary number of text subsections with the names `.text1`, `.text2`, etc.

- **Assignment to location counter** (such as "\. = .+4")

  As a workaround, you can advance the location counter with `.space <expr>`

- **Empty expressions defaulting to 0**. Example: ".byte ," equivalent to ".byte 0,0"

  There is no workaround for this. You must always supply the arguments.

- **.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>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<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>
We do not support 16-byte numbers directly. As a workaround, you may use consecutive `.long` directives to build a large number in memory.

- `.quad`

We do not support eight-byte numbers directly. As a workaround, you may use consecutive `.long` directives to build a large number in memory.
Chapter 13
Linker

The compiler organizes its object code into sections that the linker arranges when it creates its output file.

To generate an output file, the linker reads from input ELF (Executable and Linkable Format) files generated by compiler and other tools. The linker also reads a linker command file to determine how to build its output file. The linker then writes to its output file, an ELF file. This output file is the executable image, ready to load and run on the target platform.

This chapter explains the sections in the object code of and how to arrange them in the linker's output file:

- Specifying Link Order in the IDE
- Dead-Stripping
- Defining the Target's Memory Map
- Defining Sections in the Output File
- Associating Input Sections With Output Sections
- Controlling Alignment
- Specifying Memory Area Locations and Sizes
- Creating Memory Gaps
- Creating Symbols
- Linker Command File Syntax
- Commands, Directives, and Keywords

13.1 Specifying Link Order in the IDE

To specify link order, use the Link Order page of the CodeWarrior IDE's Project window. (For certain targets, the name of this page is Segments.)
Regardless of the order that the **Link Order** page specifies, the linker always processes source code files before it processes relocatable (.o) files or archive (.a) files. This policy means that the linker prefers using a symbol definition from a source file rather than a library file definition for the same symbol.

There is an exception, however: if the source file defines a weak symbol, the linker uses a global-symbol definition from a library. Use `#pragma overload` to create weak symbols.

Well-constructed projects usually do not have strong link-order dependencies.

The linker ignores executable files of the project. You may find it convenient to keep the executable files in the project folder so that you can disassemble it. If a build is successful, a check mark disappears in the touch column on the left side of the project window. The check mark indicates that the new file in the project is out of date. If a build is unsuccessful, the IDE will not be able to find the executable file and it stops the build with an appropriate message.

### 13.2 Dead-Stripping

Normally, the CodeWarrior linker ignores object code that is not referred to by other object code. If the linker detects that an object is not referred to by the rest of the program being linked, the linker will not place that object in its output file. In other words, the linker "dead-strips" objects that are not used.

Dead-stripping ensures the smallest possible output file. Also, dead-stripping relieves you from having to manually exclude unused source code from the compiler and unused object code from the linker.

There are some objects, however, that need to be in the linker's output file even if these objects are not explicitly referred to by other parts of your program. For example, an executable image might contain an interrupt table that the target platform needs, but this interrupt table is not referred to by the rest of the image.

Use the `FORCEACTIVE` directive in a linker command file to specify to the linker which objects must not be dead-stripped.

**Listing 13-1 on page 214** shows an example from a linker command file that tells the linker not to dead-strip an object named `InterruptVectorTable`.

```
Listing 13-1. FORCEACTIVE example

FORCEACTIVE { InterruptVectorTable }
```
Use FORCEFILES directive to prevent deadstripping entire files. Listing 13-1 on page 214 shows an example from a linker command file that prevents the linker dead-stripping entire files.

Listing 13-2. FORCEFILES example

FORCEFILES { segfault.o }

13.3 Defining the Target's Memory Map

Use the linker command file's MEMORY directive to delineate areas in the target platform's memory map and associate a name for each of these areas. Names defined in a MEMORY directive may be used later in the linker command file to specify where object code should be stored. Listing 13-3 on page 215 shows an example.

Listing 13-3. MEMORY directive example

MEMORY
{
   ISR_table : org = 0x00000000, len = 0x400
   data : org = 0x00000400, len = 0x10000
   flash : org = 0x10000000, len = 0x10000
   text : org = 0x80000000
}

This example defines 4 memory areas named ISR_table, data, flash, and text. The org argument specifies the beginning byte address of a memory area. The len argument is optional, it specifies how many bytes of data or executable code the linker may store in an area. The linker issues a warning message if an attempt to store object code in an area exceeds its length.

13.4 Defining Sections in the Output File

Use the linker command file's SECTIONS directive to

- define sections in the linker's output file
- to specify in which memory area on the target platform a section in the output file should be loaded at runtime

Use GROUP directives in a SECTIONS directive to organize objects.
The linker will only create a section in the output file if the section is not empty, even if the section is defined in a \texttt{SECTIONS} or \texttt{GROUP} directive.

Listing 13-4 on page 216 shows an example.

\begin{verbatim}
SECTIONS {
  GROUP :
    { .text : {} .rodata : {} } > text
  GROUP {
    .sdata : {} .sbss : {}
  } > data
  GROUP {
    .sdata2 : {} .sbss2 : {}
  } > data
}
\end{verbatim}

This example defines the \texttt{.text} and \texttt{.rodata} sections in the output file and specifies that they should be loaded in the memory area named \texttt{text} on the target platform at runtime. The example then defines sections named \texttt{.sdata} and \texttt{.sbss}. These sections will be loaded in the memory named \texttt{data}. The last \texttt{GROUP} directive in the example defines sections named \texttt{.sdata2} and \texttt{.sbss2}. These sections will also be loaded in the memory area named \texttt{data}, after the sections \texttt{.sdata} and \texttt{.sbss}.

\subsection{13.5 Associating Input Sections With Output Sections}

Normally the linker stores sections from input object code in the sections of the linker's output file that have the same name. The linker command file's \texttt{SECTIONS} and \texttt{GROUP} directives allow you to specify other ways to associate input object code with sections in linker output. Listing 13-5 on page 216 shows an example.

\begin{verbatim}
SECTIONS {
  GROUP :
    { .myText : { main.o (.text) } .text : { *(.text) } } > text
}
\end{verbatim}
This example defines a section in the output file named .myText. This section will contain the objects that are in the .text section in the object code taken from the input file named main.o. The example also defines a section in the output file named .text. This section will contain all objects in the .text sections of all input files containing object code. Both these sections in the output file, .myText and .text, will be loaded in the memory area named text on the target platform.

The SECTIONS and GROUP directives also allow you to filter what kinds of object code from input files will be stored in a section in the output file. Table 13-1 shows the kinds of data that may be filtered.

<table>
<thead>
<tr>
<th>This filter</th>
<th>allows input objects that have these permissions</th>
<th>and contain this kind of object code</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEXT</td>
<td>readable, executable</td>
<td>initialized</td>
</tr>
<tr>
<td>CODE</td>
<td>readable, executable</td>
<td>initialized</td>
</tr>
<tr>
<td>DATA</td>
<td>readable, writable</td>
<td>initialized</td>
</tr>
<tr>
<td>BSS</td>
<td>readable, writable</td>
<td>uninitialized</td>
</tr>
<tr>
<td>CONST</td>
<td>readable</td>
<td>initialized</td>
</tr>
<tr>
<td>MIXED</td>
<td>readable, writable, executable</td>
<td>initialized</td>
</tr>
<tr>
<td>VLECODE</td>
<td>readable, executable</td>
<td>initialized</td>
</tr>
</tbody>
</table>

Listing 13-6 on page 217 shows an example.

Listing 13-6. Filtering objects from input files

```c
SECTIONS
{
  .text (TEXT) : { } > text
  .bss (BSS) : { } > data
}
```

This example defines a section in the output file named .text. The linker will only store objects from input object code that are readable, executable, and initialized. This example also defines a section in the output file named .bss. This section will only contain objects from the linker's input files that are readable, writable, and uninitialized.

13.6 Controlling Alignment
Specifying Memory Area Locations and Sizes

Use the `ALIGN` argument in a `SECTIONS` or `GROUP` directive to specify an alignment relative to the start of the physical address.

Listing 13-7 on page 218 shows an example.

Listing 13-7. Example of the ALIGN directive

```c
SECTIONS
{
    GROUP:
        {
            .init ALIGN(0x1000) : {}
            .text ALIGN(0x1000) : {}
        } > text
}
```

This example defines two sections named `.init` and `.text`. At runtime, each section will be loaded at the next available address that is evenly divisible by `0x1000` in the memory area named `text` on the target platform.

13.7 Specifying Memory Area Locations and Sizes

Normally, the linker stores sections in the output file in sequential order. Each object from the linker's output is stored after the last object in the output file. Use the `BIND`, `ADDR`, and `SIZEOF` keywords in `SECTIONS` and `GROUP` directives to precisely specify where sections in the output file will be loaded.

Listing 13-8 on page 218 shows an example.

Listing 13-8. BIND, ADDR, and SIZEOF example

```c
SECTIONS
{
    .text BIND(0x00010000) : ()
    .rodata : {}
    .data BIND(ADDR(.rodata + SIZEOF(.rodata)) ALIGN(0x010) : {})
}
```

This example defines a section in the output file named `.text`. This section will be loaded at address `0x00010000` on the target platform at runtime. The next section, `.rodata`, will be loaded at the address immediately proceeding the last byte in the `.text` section. The last section, `.data`, will be loaded at the address that is the sum of the beginning of the `.rodata` section's address and the size of the `.rodata` section. This last section will be aligned at the next address that is evenly divisible by `0x10`.
The dot keyword ("."), is a convenient way to set the linker's place in the current output section.

**Listing 13-9 on page 219** shows an example.

![Listing 13-9. Skipping areas of memory](image)

This example defines two sections. The first section, `.ISR_Table`, will be loaded at the beginning of the memory area named `flash` on the target platform at runtime. The second section, `.paramsection`, will be loaded at the address that is `0x2000` bytes past the beginning of the memory area named `flash`.

### 13.8 Creating Memory Gaps

You can create gaps in memory by performing alignment calculations such as

\[ . = ( . + 0x20 ) \& \sim 0x20; \]

This kind of calculation can occur between `output_specs`, between `input_specs`, or even in `address_modifiers`. A "." refers to the current address. You may assign the `. `to a specific unallocated address or just do alignment as the example shows. The gap is filled with zeroes, in the case of an alignment (but not with `ALIGN()`).

You can specify an alternate fill pattern with `= <short_value>`, as in

\[ .text : { . = ( . + 0x20 ) \& \sim 0x20; *(.text) } = 0xAB > text \]

`short_value` is 2 bytes long. Note that the fill pattern comes before the `memory_spec`. You can add a fill to a `GROUP` or to an individual `output_spec` section. Fills cannot be added between `.bss` type sections. All calculations must end in a ",".
13.9 Creating Symbols

You can create symbols that you can use in your program by assigning a symbol to some value in your linker command file.

```plaintext
.text : { _red_start = .; *(.text) _red_end = .;} > text
```

In the example above, the linker generates the symbols `_red_start` and `_red_end` as 32 bit values that you can access in your source files. `_red_start` is the address of the first byte of the `.text` section and `_red_end` is the byte that follows the last byte of the `.text` section.

You can use any of the pseudo functions in the `address_modifiers` in a calculation.

The CodeWarrior linker automatically generates symbols for the start address, the end address, and the start address for the section if it is to be burned into ROM. For a section `.red`, we create `_f_red`, `_e_red`, and `_f_red_rom`. In all cases, any `"."` in the name is replaced with a `"_"`. Addresses begin with an `"_f"`, addresses after the last byte in section begin with an `"_e"`, and ROM addresses end in a `"_rom"`. See the header file `__ppc_eabi_linker.h` for further details.

All user defined sections follow the preceding pattern. However, you can override one or more of the symbols that the linker generates by defining the symbol in the linker command file.

**NOTE**

BSS sections do not have a ROM symbol.

13.10 Linker Command File Syntax

Linker command file syntax is a notation and implies what an LCF file includes. Listing 13-10 on page 220 shows the syntax for linker command files.

**Listing 13-10. Linker Command File Syntax**

```plaintext
linker-command-file =
    command* memory? command* sections? command*
```

This syntax implies that an LCF file can contain:

Zero or more command directives followed by Zero or at most one memory directive followed by Zero or more command directives followed by Zero or at most one sections directive followed by Zero or more command directives.
The table below lists the notations used in the linker command file syntax.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Implies zero or any number of directives</td>
</tr>
<tr>
<td>?</td>
<td>Implies zero or at most one directive</td>
</tr>
</tbody>
</table>

Listing 13-11 on page 221 shows the syntax of all valid linker command file keywords, directives, and commands.

**Listing 13-11. Linker Command File Syntax (Commands, Directives, and Keywords)**

```plaintext
command =
  exclude-files | force-active | force-files | include-dwarf | keep | ref-include | shorten-names-for-tornado-101 | cats-bss-mod | cats-header-mod | data-type-converts | entry | init | term | external-symbol | internal-symbol | memory-gaps

exclude-files =
  "EXCLUDEFILES" "{ " file-name+ "}"

force-active =
  "FORCEACTIVE" "{ " symbol+ "}"

letter =
  'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm'
  'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z'
  'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M'
  'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z'

file-name =
  (letter |"_") (letter |digit |"_")*
  (".")?(letter |digit |"_")*

section-name =
  (letter |"_") (letter |digit |"_")*

object-file =
  (letter |"_") (letter |digit |"_")* (".") ("o"|"O")

archive-file =
  (letter |"_") (letter | digit |"_")* (".") ("a"|"A")

include-dwarf =
  "INCLUDEDWARF" "{ " file-name "}"

keep =
  "KEEP"" *( section-name )"

ref-include =
```
Linker Command File Syntax

"REF_INCLUDE" {" section-name+ "}"

shorten-names-for-tornado-101=
"SHORTEN_NAMES_FOR_TOR_101"

cats-bss-mod =
"CATS_BSS_MOD"

cats-header-mod =
"CATS_HEADER_MOD"

data-type-converts =
"DATA_TYPE_CONVERTS"

entry =
"ENTRY" "(" symbol ")"

init =
"INIT" "(" symbol ")"

term =
"TERM" "(" symbol ")"

external-symbol =
"EXTERNAL_SYMBOL" "{" symbol "," symbol "}"

internal-symbol =
"INTERNAL_SYMBOL" "{" symbol "," symbol "}"

group =
"GROUP" address-modifiers ":"
"{" (section-spec )* "}" ["=" fill-shortnumber ]
[ "\> " mem-area-symbol ]

hexadigit =
'0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'|
'A'|'B'|'C'|'D'|'E'|'a'|'b'|'c'|'d'|'e'

digit =
'0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'

hexadecimal-number =
"0"("x"|"X")hexadigit+

decimal-number =
digit+

number =
hexadecimal-number | decimal-number

binary-op =
'+' | '-' | '*=' | '/=' | '%=' | '===' | '!=' | '>' | '>=' | '<' | '<=' | '&&' | '||' |
'>' | '<' | '>' | '<' |

unary-op =
'++' | '--' | '~' | '!' |

postfix-unary-op =
'++' | '--'

symbol-declaration =
{ symbol "=" address-spec } |
{ "PROVIDE" "(" identifier "=" address=spec ")" } |
{ "EXPORTSYMTAB" } |
{ "EXPORTSTRTAB" } |
{ "BYTE" "(" address-spec ")" } |
{ "SHORT" "(" address-spec ")" } |
{ "LONG" "(" address-spec ")" }

symbol =
operand = (number | "ADDR" "(" output-section-spec | address-expr ")" | "ROMADDR" "(" output-section-spec | address-expr ")" | "SIZEOP" "(" output-section-spec | address-expr ")" | "SIZEOP_ROM" "(" output-section-spec | address-expr ")" )

address-spec = number | "," | operand | (address-spec binary-op operand) | (unary-op address-spec) | (address-spec postfix-unary-op)

memory-spec = memory-area-symbol ":" "origin" | "org" | "o" "+" number "," "length" | "len" | "l" "+" number | [">" file-name]

memory-gaps = "," "+" address-spec

memory = "MEMORY" "{" memory-spec "+" }"

sections = "SECTIONS" "{" (section-spec | memory-gaps | symbol-declaration | group )* "}"

section-spec = output-section-name ":" 
["[" input-type "]"] [address-modifiers ] "{" [ (input-section-spec )* ] "}"

[= fill-shortnumber] [(">"|">">") memory-area-symbol ]

output-section-name = section-name

input-type = [ "TEXT" | "DATA" | "BSS" | "CONST" | "ZTEXT" | "ZCODE" | "VLECODE" ]

address-modifiers = ["BIND" "(" address-spec ")" ] ["ALIGN" "(" address-spec ")" ] ["NEXT" "(" address-spec ")" ] ["LOAD" "(" address-spec ")" ] ["INTERNAL_LOAD" "(" address-spec ")" ]

input-section-spec = { file-name | "(" section-name ")" | "(" section-name ")" | symbol-declaration | data-write }

data-write = ("LONG" | "SHORT" | "BYTE") "(" number ")"

fill-shortnumber = number
13.10.1 Linker Command File Operator Precedence and Support

The table below lists the LCF operators supported and their order of precedence (highest to lowest). Their associativity indicates in what order operators of equal precedence in an expression are applied.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>Parenthesis</td>
<td>left-to-right</td>
</tr>
<tr>
<td>!</td>
<td>Logical negation</td>
<td>left-to-right</td>
</tr>
<tr>
<td>~</td>
<td>Bitwise complement</td>
<td>left-to-right</td>
</tr>
<tr>
<td>* / %</td>
<td>Multiplication/Division/Modulus</td>
<td>left-to-right</td>
</tr>
<tr>
<td>+</td>
<td>Addition</td>
<td>left-to-right</td>
</tr>
<tr>
<td>&lt;&lt; &gt;&gt;</td>
<td>Bitwise shift left, Bitwise shift right</td>
<td>left-to-right</td>
</tr>
<tr>
<td>==</td>
<td>Is equal</td>
<td>left-to-right</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>left-to-right</td>
</tr>
<tr>
<td>!=</td>
<td>Is not equal to</td>
<td>left-to-right</td>
</tr>
<tr>
<td>&lt; &lt;=</td>
<td>Relational less than/Less than or equal to</td>
<td>left-to-right</td>
</tr>
<tr>
<td>&gt; &gt;=</td>
<td>Relational greater than/Greater than or equal to</td>
<td>left-to-right</td>
</tr>
<tr>
<td>&amp;</td>
<td>Bitwise and</td>
<td>left-to-right</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bitwise inclusive or</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Logical and</td>
<td>left-to-right</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>Assignment</td>
<td>right-to-left</td>
</tr>
</tbody>
</table>

13.11 Commands, Directives, and Keywords

The rest of this chapter consists of explanations of all valid LCF functions, keywords, directives, and commands, in alphabetic order.

- . (location counter)
- ADDR
- ALIGN
- ALIGNALL
- BIND
• BIN_FILE_TYPE
• BYTE
• EXCEPTION
• EXCLUDEFILES
• EXPORTSTRTAB
• EXPORTSYMTAB
• EXTERNAL_SYMBOL
• FORCEACTIVE
• GROUP
• IMPORTSTRTAB
• IMPORTSYMTAB
• INCLUDE
• INCLUDEDWARF
• INTERNAL_LOAD
• INTERNAL_SYMBOL
• KEEP
• LOAD
• LONG
• MEMORY
• MAX_BIN_GAP
• NEXT
• NO_INIT_BSS
• NO_TRAILING_BSS_IN_BIN_FILES
• OBJECT
• REF_INCLUDE
• REGISTER
• ROMADDR
• SECTIONS
• SHORT
• SIZEOF
• SIZEOF_ROM
• WRITES0COMMENT

13.11.1 . (location counter)

Denotes the current output location.

Remarks
The period always refers to a location in a sections segment, so is valid only in a sections-section definition. Within such a definition, '. ' may appear anywhere a symbol is valid.

Assigning a new, greater value to '. ' causes the location counter to advance. But it is not possible to decrease the location-counter value, so it is not possible to assign a new, lesser value to '. ' You can use this effect to create empty space in an output section, as the Listing 13-12 on page 226 example does.

Example

The code of Listing 13-12 on page 226 moves the location counter to a position 0x10000 bytes past the symbol __start.

Listing 13-12. Moving the Location Counter

```
..data :
{
    *(data)
    *(bss)
    *(COMMON)
    __start = .;
    . = __start + 0x10000;
    __end = .;
} > DATA
```

13.11.2 ADDR

Returns the address of the named section or memory segment.

ADDR (sectionName | segmentName)

Parameters

sectionName
Identifier for a file section.

segmentName
Identifier for a memory segment

Example
The code of Listing 13-13 on page 227 uses the ADDR function to assign the address of ROOT to the symbol __rootbasecode.

**Listing 13-13. ADDR() Function**

```plaintext
MEMORY{
    ROOT : origin = 0x80000400, length = 0
}

SECTIONS{
    .code :
    {
        __rootbasecode = ADDR(ROOT);
        *(text);
    } > ROOT
}
```

### 13.11.3 ALIGN

Returns the location-counter value, aligned on a specified boundary.

ALIGN(alignValue)

**Parameter**

alignValue

Alignment-boundary specifier; must be a power of two.

**Remarks**

The ALIGN function does not update the location counter; it only performs arithmetic. Updating the location counter requires an assignment such as:

`. = ALIGN(0x10); #update location counter to 16-byte alignment`

### 13.11.4 ALIGNALL
Forces minimum alignment for all objects in the current segment to the specified value.

ALIGNALL(alignValue);

**Parameter**

alignValue

Alignment-value specifier; must be a power of two.

**Remarks**

ALIGNALL is the command version of the ALIGN function. It updates the location counter as each object is written to the output.

**Example**

Listing 13-14 on page 228 is an example use for ALIGNALL() command.

**Listing 13-14. ALIGNALL Example**

```
.code :
{
    ALIGNALL(16); // Align code on 16-byte boundary
    * (.init)
    * (.text)

    ALIGNALL(64); //align data on 64-byte boundary
    * (.rodata)
} > .text
```

### 13.11.5 BIND

Specifies a section modifier for setting an address.

BIND(address)

**Parameter**

address

A memory address, such as 0x80000400.
13.11.6  BIN_FILE_TYPE

Controls the generation of a binary file. By default, the linker does not generate any binary file, even if the s-record generation is enabled.

BIN_FILE_TYPE (NO_BIN_FILE | ONE_BIN_FILE | MULTIPLE_BIN_FILES)

Parameters

NO_BIN_FILE

No binary file will be generated even if s-record generation is on.

ONE_BIN_FILE

Generates a single binary file with all the loadable code and data, even if s-record generation is off.

MULTIPLE_BIN_FILES

A separate binary file is generated for each MEMORY directive. When selected, the hexadecimal address of the location, the file should be loaded is inserted between the filename and extension.

Example

Listing 13-15 on page 229 is an example of a binary file generated using the MULTIPLE_BIN_FILES parameter for a non-ROM Image build.

Listing 13-16 on page 229 is an example of a binary file generated using the MULTIPLE_BIN_FILES parameter for a ROM Image build.
The code of Listing 13-17 on page 230 uses the MULTIPLE_BIN_FILES parameter to change the name of a binary file in the MEMORY directive.

**Listing 13-17. Change Binary File Name in the MEMORY Directive**

```plaintext
MEMORY {
    code : org = 0x00002000 > my_binary_file.xyz
}
```

Any MEMORY directive without a custom name will be given a name like Test.ffe00000.bin, where the binary file reflects the ROM address of the segment.

The code of Listing 13-18 on page 230 uses the MULTIPLE_BIN_FILES parameter to merge some binary files together, assuming that there are no segments between them.

**Listing 13-18. Merge Binary Files Together Example**

```plaintext
MEMORY {
    code : org = 0x00002000 > my_binary_file.xyz
    special : org = 0x00004000 > my_binary_file.xyz
}
```

### 13.11.7 BYTE

Inserts a byte of data at the current address of a section.

```plaintext
BYTE {
    expression
} ;
```

**Parameter**

- **expression**

  Any expression that returns a value **0x00** to **0xFF**.

### 13.11.8 EXCEPTION

Creates the exception table index in the output file.

```plaintext
EXCEPTION
```

**Remarks**
Only C++ code requires exception tables. To create an exception table, add the `EXCEPTION` command, with symbols `__exception_table_start__` and `__exception_table_end__`, to the end of your code section segment, just as Listing 13-19 on page 231 shows. (At runtime, the system knows the values of the two symbols.)

**Example**

Listing 13-19 on page 231 shows the code for creating an exception table.

**Listing 13-19. Creating an Exception Table**

```c
__exception_table_start__ = .;
EXCEPTION

__exception_table_end__ = .;
```

## 13.11.9 EXCLUDEFILES

Ignores object code in files.

**Syntax**

```c
EXCLUDEFILES { 
  file-name 
}
```

**Remarks**

This directive is for partial link projects only. It makes your partial link file smaller. `EXCLUDEFILES` can be used independently of `INCLUDEDWARF`. Unlike `INCLUDEDWARF`, `EXCLUDEFILES` can take any number of executable files.

In this example

```c
EXCLUDEFILES { kernel.elf }
```

`kernel.elf` is added to your project but the linker does not add any section from `kernel.elf` to your project. However, it does delete any weak symbol from your partial link that also exists in `kernel.elf`. Weak symbols can come from templates or out-of-line inline functions.

## 13.11.10 EXPORTSTRTAB
Creates a string table from the names of exported symbols.

**EXPORTSTRTAB**

**Remarks**

Table 13-4 shows the structure of the export string table. As with an ELF string table, the system zero-terminates the library and symbol names.

<table>
<thead>
<tr>
<th>Table 13-4. Export String Table Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
</tr>
<tr>
<td>1 byte</td>
</tr>
<tr>
<td>library name</td>
</tr>
<tr>
<td>varies</td>
</tr>
<tr>
<td>symbol1 name</td>
</tr>
<tr>
<td>varies</td>
</tr>
<tr>
<td>symbol2 name</td>
</tr>
<tr>
<td>varies</td>
</tr>
</tbody>
</table>

**Example**

Listing 13-20 on page 232 shows the code for creating an export string table.

```
Listing 13-20. Creating an Export String Table

.expstr:
{
    EXPORTSTRTAB
} > EXPSTR
```

**13.11.11 EXPORTSYMTAB**

Creates a jump table of the exported symbols.

**EXPORTSYMTAB**

**Remarks**

Table 13-5 shows the structure of the export symbol table. The start of the export symbol table must be aligned on at least a four-byte boundary.

<table>
<thead>
<tr>
<th>Table 13-5. Export Symbol Table Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size (in bytes) of export table</td>
</tr>
<tr>
<td>4 bytes</td>
</tr>
<tr>
<td>Index to library name in export symbol table</td>
</tr>
<tr>
<td>4 bytes</td>
</tr>
<tr>
<td>Index to symbol1 name in export symbol table</td>
</tr>
<tr>
<td>4 bytes</td>
</tr>
<tr>
<td>Address of symbol1</td>
</tr>
<tr>
<td>4 bytes</td>
</tr>
</tbody>
</table>
Table 13-5. Export Symbol Table Structure (continued)

<table>
<thead>
<tr>
<th>A5 value for symbol1</th>
<th>4 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index to symbol2 name in export symbol table</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Address of symbol2</td>
<td>4 bytes</td>
</tr>
<tr>
<td>A5 value for symbol2</td>
<td>4 bytes</td>
</tr>
</tbody>
</table>

Example

Listing 13-21 on page 233 shows the code for creating an export symbol table.

Listing 13-21. Creating an Export Symbol Table

```
.expsym:
{
  EXPORTSYMTAB
}
```

13.11.12 EXTERNAL_SYMBOL

Specifies objects that may be referred to by modules outside of an object code file.

Syntax

```
EXTERNAL_SYMBOL {
  symbol [, symbol ]*
}
```

Remarks

The symbols must be the link time symbol names. In the case of C++ object code, these names must be the mangled.

13.11.13 FORCEACTIVE

Specifies objects that must not be deadstripped.

Syntax

```
FORCEACTIVE {
  symbol [, symbol ]*
}
```
13.11.14  GROUP

Organizes objects listed in a linker command file.

Syntax

GROUP
  address-modifiers  : {  
    section-spec 
    [,  
    section-spec ]* } 

[ >  
  memory-area-name  ]

Remarks

The GROUP directive lets you organize the linker command file.

Listing 13-22 on page 234 shows that each group starts at a specified address. If no address_modifiers are present, it would start following the previous section or group. Although you normally do not have an address_modifier for an output_spec within a group; all sections in a group follow contiguously unless there is an address_modifier for that output_spec.

The optional memory-area-name clause specifies a name defined with the MEMORY directive. Using this clause specifies the memory space in the target platform where the group's object code should be placed.

Listing 13-22. Example of linker GROUP directive

SECTIONS {
  GROUP BIND(0x00010000) : { 
    .text : {}
    .rodata : {*(.rodata) *(extab) *(extabindex)}
  }

  GROUP BIND(0x2000) : {
    .data : {}
    .bss : {}
    .sdata BIND(0x3500) : {}
    .sbss  : {}
  }
}
13.11.15 IMPORTSTRTAB

Creates a string table from the names of imported symbols.

Remarks

Table 13-6 shows the structure of the import string table. As with an ELF string table, the system zero-terminates the library and symbol names.

<table>
<thead>
<tr>
<th>0x00</th>
<th>1 byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>library name</td>
<td>varies</td>
</tr>
<tr>
<td>symbol1 name</td>
<td>varies</td>
</tr>
<tr>
<td>symbol2 name</td>
<td>varies</td>
</tr>
</tbody>
</table>

Example

Listing 13-23 on page 235 shows the code for creating an import string table.

Listing 13-23. Creating an Import String Table

```assembly
.impstr:
{
    IMPORTSTRTAB
} > IMPSTR
```
13.11.16 IMPORTSYMTAB

Creates a jump table of the imported symbols.

Remarks

Table 13-7 shows the structure of the import symbol table. The start of the import symbol table must be aligned on at least a four-byte boundary.

<table>
<thead>
<tr>
<th>Description</th>
<th>Size (in bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size (in bytes) of import table</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Index to library1 name in import string table</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Number of entries in library1</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Index to symbol1 name in import string table</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Address of symbol1 vector in export string table</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Index to symbol2 name in import string table</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Address of symbol2 vector in export string table</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Index to library2 name in import string table</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Number of entries in library2</td>
<td>4 bytes</td>
</tr>
</tbody>
</table>

Example

Listing 13-24 on page 236 shows the code for creating an import symbol table.

Listing 13-24. Creating an Import Symbol Table

```
.exp sym:
{
    IMPORTSYMTAB
}
> EXPSYM
```

13.11.17 INCLUDE

Include a specified binary file in the output file.

INCLUDE filename

Parameter
filename

Name of a binary file in the project. The File Mappings target settings panel must specify resource file for all files that have the same extension as this file.

13.11.18 INCLUDEDWARF

Allows source-level kernel debugging.

Syntax

INCLUDEDWARF { file-name }

Remarks

In this example

INCLUDEDWARF { kernel.elf }

the linker inserts the .debug and .line sections of kernel.elf to your application. These sections let you debug source level code in the kernel while debugging your application.

You are limited to one executable file when using this directive. If you need to process more than one executable, add this directive to another file.

13.11.19 INTERNAL_LOAD

Loads one or several segments at an address not specified at link time.

Syntax

INTERNAL_LOAD(address)

Parameter

address

A memory address, such as 0x80000400.

Remarks

Use INTERNAL_LOAD directive to specify an internal ROM addr_mode.
13.11.20  INTERNAL_SYMBOL

Symbols created in a linker command file are considered external unless they are redefined using INTERNAL_SYMBOL.

Syntax

```
INTERNAL_SYMBOL {
  symbol [,
  symbol ]* }
```

13.11.21  KEEP

Forces the linker to not dead strip the unused symbols in the specified section.

Syntax

```
KEEP(*(sectionType))
```

Parameter

`sectionType`

Identifier for any user-defined or predefined section.

Example

Listing 13-25 on page 238 shows the sample usage.

```
Listing 13-25. KEEP Directive Usage

GROUP : {
  .text (TEXT) : {}
  .mycode (TEXT) : {KEEP(*(.mycode))}
  ...}
```

13.11.22  LOAD

Loads one or several segments at a specific address.
### Syntax

`LOAD(address)`

### Parameter

**address**

A memory address, such as 0x80000400.

### Remarks

- Use `LOAD` directive to specify an external ROM `addr_mode`.
- The `LOAD` directive can be used on all the sections going into ROM.
- The directive is ignored in links if Generate ROM image is not checked in the linker preference panel.
- Inconsistent initialized data sections copied from ROM to RAM may need a reference to a ROM address. For example, to store the `.data` and `.sdata` sections in ROM such that they are immediately after the `.text` section, try the following:

```
.text LOAD(ADDR(.text)) : {} > rom
.data LOAD(ROMADDR(.text) + SIZEOF(.text)) : {} > code
.sdata LOAD(ROMADDR(.data) + SIZEOF(.data)) : {} > code
```

#### 13.11.23 LONG

Inserts a word of data at the current address of a section.

```
LONG {
  expression
}
```

### Parameter

**expression**

Any expression that returns a value 0x00000000 to 0xFFFFFFFF.

#### 13.11.24 MEMORY

Starts the LCF memory segment, which defines segments of target memory.

```
MEMORY { memory_spec[, memory_spec] }
```
Parameters

memory_spec

segmentName: origin = address, length = length [> fileName]

segmentName

Name for a new segment of target memory. Consists of alphanumeric characters; can include the underscore character.

accessFlags

ELF-access permission flags - R = read, W = write, or X = execute.

address

A memory address should be a numeric value, such as 0x80000400.

length

Size of the new memory segment: a value greater than zero. Optionally, the value zero for autolength, in which the linker allocates space for all the data and code of the segment. (Autolength cannot increase the amount of target memory, so the feature can lead to overflow.)

fileName

Optional, binary-file destination. The linker writes the segment to this binary file on disk, instead of to an ELF program header. The linker puts this binary file in the same folder as the ELF output file. This option has two variants:

- > fileName: writes the segment to a new binary file.
- >> fileName: appends the segment to an existing binary file.

Remarks

The LCF contains only one MEMORY directive, but this directive can define as many memory segments as you wish.

For each memory segment, the ORIGIN keyword introduces the starting address, and the LENGTH keyword introduces the length value.

There is no overflow checking for the autolength feature. To prevent overflow, the segment start address should be specified correctly.

Example

Listing 13-26 on page 240 is an example use of the MEMORY directive.

Listing 13-26. MEMORY Directive Example
MEMORY {
    TEXT (RX) : ORIGIN = 0x00003000, LENGTH = 0
    DATA (RW) : ORIGIN = 0x5000, LENGTH = 0
}

13.11.25 MAX_BIN_GAP

Controls the maximum gap size value between two segments.

MAX_BIN_GAP(nnnnn)

Parameters

nnnnn
Size of the maximum gap allowed between segments.

Remarks

The directive can be placed in the LCF anyway except within the MEMORY and SECTIONS directives.

This directive can only be used if you are generating a single binary file.

13.11.26 NEXT

Specifies an expression for setting an address.

NEXT(address)

Parameter

address

A memory address, such as 0x80000400.

13.11.27 NO_INIT_BSS
Prevents initializing a BSS section to zero to avoid uninitialized variable to be initialized to zero by the startup.

**Syntax**

NO_INIT_BSS

**Remarks**

If the variable is initialized to 0 while using the directive NO_INIT_BSS then the variable will be placed in BSS section and will not be initialized to zero (Listing 13-27 on page 242).

**Example**

Following are few examples of using the NO_INIT_BSS directive.

Listing 13-27. NO_INIT_BSS Directive Usage - Example 1

LCF:

```
GROUP : {
    ___uninitialized_intc_handler_table ALIGN(0x10) : {}
    .data : {}
    .user_def_Init (DATA):{}
    .sdata : {}
    .sbss : {}
    .sdata2 : {}
    .sbss2 : {}
    .bss : {}
    .user_def_Uninit (BSS) NO_INIT_BSS :{}
} > internal_ram
```

Test case:

```c
#pragma section RW ".user_def_Init" ".user_def_Uninit"
// first user defined section treated as initialized section and second
// user defined section treated as uninitialized section
__declspec (section ".user_def_Init")  int myvar_a;
// myvar_a goes to user_def_Init section as it is not initialized
__declspec (section ".user_def_Init")  int myvar_b=3;
// myvar_b goes to user_def_Init section as it is initialized
```
__declspec (section ".user_def_Init") int myvar_c=0;

// myvar_c goes to user_def_Init section as it is initialized to zero
// But var myvar_c is not initialized to zero as it comes under
NO_INIT_BSS

int main(void) {
    volatile int i = 0;

    /* Loop forever */
    for (;;) {
        i++;
        myvar_a ++;
        myvar_b ++;
    }
}

Listing 13-28. NO_INIT_BSS Directive Usage - Example 2

LCF:
........
GROUP : {
    __uninitialized_intc_handlertable ALIGN(0x10) : {}
    .data : {}
    .user_def_Init (DATA):{}
    .sdata : {}
    .sbss : {}
    .sdata2 : {}
    .sbss2 : {}
    .bss : {}
    .user_def_Uninit (BSS) NO_INIT_BSS :{}
} > internal_ram

........

Test case:
#pragma push //Save the original pragma context
#pragma section data_type sdata_type ".user_def_Init"
    ".user_def_Uninit"

unsigned int test_var; //Here variable test_var will place in the
section user_def_Uninit and NOT initializes to zero.
#pragma pop

unsigned int test_var2 = 0; // Here variable test_var2 will place in .bss/.sbss section and initializes to zero.

Listing 13-29. NO_INIT_BSS Directive Usage - Example 3

LCF:

......

GROUP : {
    .__uninitialized_intc_handlertable ALIGN(0x10) : {} 
    .data : {} 
    .user_def_Init (DATA):{} 
    .sdata : {} 
    .sbss : {} 
    .sdata2 : {} 
    .sbss2 : {} 
    .bss : {} 
    .user_def_Uninit (BSS) NO_INIT_BSS :{}
} > internal_ram

......

Test case:
#pragma push // Save the original pragma context

#pragma section data_type sdata_type ".user_def_Init"
".user_def_Uninit"

unsigned int test_var; // Variable test_var will place in the section user_def_Uninit

#pragma explicit_zero_data on

unsigned int test_var2 = 0; // Variable test_var2 will place in the section user_def_Init as the pragma // explicit_zero_data is enabled

#pragma explicit_zero_data reset
(OR #pragma explicit_zero_data off)

unsigned int test_var3 = 5; // Variable test_var3 will place in the section user_def_Init

unsigned int test_var4 = 0; // Variable test_var3 will place in the section user_def_Uninit

#pragma pop
13.11.28  NO_TRAILING_BSS_IN_BIN_FILES

Removes uninitialized data contained in a binary file.

Syntax

NO_TRAILING_BSS_IN_BIN_FILES

Remarks

This directive can only be used if the last section or sections of a binary file contains
uninitialized data.

Example

Listing 13-30 on page 245 is an example use of the
NO_TRAILING_BSS_IN_BIN_FILES directive.

Listing 13-30. NO_TRAILING_BSS_IN_BIN_FILES Directive Example

SECTIONS
    {
        GROUP {
            .text: {}
        }
    }

>code

NO_TRAILING_BSS_IN_BIN_FILES

13.11.29  OBJECT

Sections-segment keyword that specifies a function. Multiple OBJECT keywords control the
order of functions in the output file.

OBJECT (function, sourcefile.c)

Parameters

function

Name of a function.

sourcefile.c

Name of the C file that contains the function.
Remarks

If an \texttt{OBJECT} keyword tells the linker to write an object to the output file, the linker does not write the same object again, in response to either the \texttt{GROUP} keyword or the \texttt{"*"} wildcard character.

13.11.30 \textbf{REF\_INCLUDE}

Starts an optional LCF closure segment that specifies sections the linker should \textit{not} deadstrip, if program code references the files that contain these sections.

\texttt{REF\_INCLUDE\{ sectionType[, sectionType] \}}

Parameter

\texttt{sectionType}

Identifier for any user-defined or predefined section.

Remarks

Useful if you want to include version information from your source file components.

13.11.31 \textbf{REGISTER}

Use the \texttt{REGISTER} directive to assign one of the EPPC processor's non-volatile registers to a user-defined small data section.

\texttt{REGISTER(nn [ , limit])}

Parameter

\texttt{nn}

Specifies one of the predefined small data base registers, a non-volatile EPPC register, or any of the following values:

- 0, 2, 13

These registers are for the predefined small data sections:

\begin{itemize}
  \item 0 - .PPC.EMB.sdata0/.PPC.EMB.sbss0
\end{itemize}
2 - .sdata2/sbss2

13 - .sdata/sbss

You do not have to define these sections using REGISTER because they are predefined.

- 14 - 31

Match any value in this range with the register reserved by your global register variable declaration.

- -1

This "register" value instructs the linker to treat relocations that refer to objects in your small data section as non-small data area relocations. These objects are converted to near absolute relocations, which means that the objects referenced must reside within the first 32 KB of memory. If they do not, the linker emits a "relocation out of range" error. To fix this problem, rewrite your code such that the offending objects use large data relocations.

\textbf{limit}

Specifies the maximum size of the small data section to which register \texttt{nn} is bound. This value is the size of the initialized and uninitialized sections of the small data section combined. If \texttt{limit} is not specified, 0x00008000 is used.

\textbf{NOTE}

Each small data section you create makes one less register available to the compiler; it is possible to starve the compiler of registers. As a result, create only the number of small data sections you need.

\section*{13.11.32 ROMADDR}

Equivalent to ADDR. Returns ROM address.

\texttt{ROMADDR \{sectionName | segmentName\}}

\textbf{Parameters}

\texttt{sectionName}

Identifier for a file section.

\texttt{segmentName}
Identifier for a memory segment

Example

The code of Listing 13-31 on page 248 uses the ROMADDR function to assign the address of ROM to the symbol __rootbasecode.

### Listing 13-31. ROMADDR() Function

```
MEMORY{
  ROM : origin = 0x80000400, length = 0
}

SECTIONS{
  .code :
  {
    __rootbasecode = ROMADDR(ROM);
    *(.text);
  } > ROM
}
```

### 13.11.33 SECTIONS

Starts the LCF sections segment, which defines the contents of target-memory sections. Also defines global symbols to be used in the output file.

```c
SECTIONS { section_spec[, section_spec] }
```

**Parameters**

`section_spec`

- `sectionName : [LOAD (loadAddress)] {contents} > segmentName`

  **sectionName**

  Name for the output section. Must start with a period.

  **LOAD**

  Loads one or several segments at a specific address.
Statements that assign a value to a symbol or specify section placement, including input sections.

**segmentName**

Predefined memory-segment destination for the contents of the section. The two variants are:

- \( > \) **segmentName**: puts section contents at the beginning of memory segment **segmentName**.
- \( >> \) **segmentName**: appends section contents to the end of memory segment **segmentName**.

**Example**

**Listing 13-32 on page 249** is an example sections-segment definition.

**Listing 13-32. SECTIONS Directive Example**

```assembly
SECTIONS {
  .text : {
    _textSegmentStart = .;
    alpha.c (.text)
    . = ALIGN (0x10);
    beta.c (.text)
    _textSegmentEnd = .;
  }
  .data : { *(.data) }
  .bss : { *(.bss)
    *(COMMON)
  }
}
```

**13.11.34 SHORT**

Inserts a halfword of data at the current address of a section.

```assembly
SHORT
  { expression }
```
Parameter

expression

Any expression that returns a value 0x0000 to 0xFFFF

13.11.35 SIZEOF

Returns the size (in bytes) of the specified segment or section.

SIZEOF(segmentName | sectionName)

Parameters

segmentName

Name of a segment; must start with a period.

sectionName

Name of a section; must start with a period.

13.11.36 SIZEOF_ROM

Returns the size (in bytes) that a segment occupies in ROM.

SIZEOF_ROM (segmentName)

Parameter

segmentName

Name of a ROM segment; must start with a period.

Remarks

Always returns the value 0 until the ROM is built. Accordingly, you should use SIZEOF_ROM only within an expression inside a BYTE, SHORT, or LONG function.

Furthermore, you need SIZEOF_ROM only if you use the COMPRESS option on the memory segment. Without compression, there is no difference between the return values of SIZEOF_ROM and SIZEOF.
13.11.37  WRITES0COMMENT

Inserts an S0 comment record into an S-record file.

WRITES0COMMENT "comment"

**Parameter**

*comment*

Comment text: a string of alphanumerical characters 0-9, A-Z, and a-z, plus space, underscore, and dash characters. Double quotes *must* enclose the comment string. (If you omit the closing double-quote character, the linker tries to put the entire LCF into the S0 comment.)

**Remarks**

This command, valid only in an LCF sections segment, creates an S0 record of the form:

S0aa0000bbbbbbbbbbbbbbbbbbdd

- **aa** - hexadecimal number of bytes that follow
- **bb** - ASCII equivalent of *comment*
- **dd** - the checksum

This command does not null-terminate the ASCII string.

Within a comment string, do not use these character sequences, which are reserved for LCF comments: # /* */ //</p>

**Example**

This example shows that multi-line S0 comments are valid:

WRITES0COMMENT "Line 1 comment
Line 2 comment"
Chapter 14
Linker for Power Architecture Processors

This chapter describes how to use the features in the CodeWarrior linker that are specific to Power Architecture software development.

• Predefined Sections
• Additional Small Data Sections
• Linker Map File
• Deadstripping
• CodeWarrior Linker Command File (LCF)
• Creating an LCF from Scratch
• Relocating Code in ROM
• Relocating Code and Data in Internal RAM
• Relocating Code and Data in External MRAM
• Unique LCF Examples
• Linker Command File Commands

14.1 Predefined Sections

Table 14-1 describes the sections that the compiler creates.

NOTE
The Compiler-defined section names are case sensitive. For example, using .binary instead of .BINARY will not give expected results.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.bss</td>
<td>uninitialized global data</td>
</tr>
<tr>
<td>.BINARY</td>
<td>Binary files.</td>
</tr>
</tbody>
</table>

Table continues on the next page...
Table 14-1. Compiler-defined sections (continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.ctors</td>
<td>C++ constructors and Altivec vector constructors</td>
</tr>
<tr>
<td>.dtors</td>
<td>C++ destructors</td>
</tr>
<tr>
<td>.data</td>
<td>initialized global data</td>
</tr>
<tr>
<td>extab</td>
<td>C++ exception tables</td>
</tr>
<tr>
<td>extabindex</td>
<td>C++ exception tables</td>
</tr>
<tr>
<td>.init</td>
<td>initialization executable code from the runtime library</td>
</tr>
<tr>
<td>.init_vle</td>
<td>Initialization executable code for VLE compilers</td>
</tr>
<tr>
<td>.PPC.EMB.sdata0</td>
<td>Initialized data with addressing relative to address 0</td>
</tr>
<tr>
<td>.PPC.EMB.sbss0</td>
<td>Uninitialized data with addressing relative to address 0</td>
</tr>
<tr>
<td>.rodata</td>
<td>literal values and initialization values in the application's source code</td>
</tr>
<tr>
<td>.sdata</td>
<td>initialized small global data</td>
</tr>
<tr>
<td>.sdata2</td>
<td>initialized global small data defined with the const keyword</td>
</tr>
<tr>
<td>.sbss</td>
<td>uninitialized global small data</td>
</tr>
<tr>
<td>.sbss2</td>
<td>uninitialized global constant small data defined with the const keyword</td>
</tr>
<tr>
<td>.text</td>
<td>application code</td>
</tr>
<tr>
<td>.text_vle</td>
<td>application code for VLE compilers</td>
</tr>
</tbody>
</table>

14.1.1 Linking Binary Files

You can link external binary files/data (tables, Bitmap graphics, sound records) into the project image. The following sections explain how to link binary files using CodeWarrior IDE and Command line:

- **Using CodeWarrior IDE**
- **Using Command-Line**

14.1.1.1 Using CodeWarrior IDE

To link a binary file using CodeWarrior IDE, perform the following steps:

1. Launch CodeWarrior and open the desired project to add the binary file.
2. Add a binary file (bin_data.bin) to project.
3. Select **Properties > C/C++ Build > Settings > Tool Settings > PowerPC Linker > General** and add the binary file to the **Other Flags** text box.
4. Update linker command file (.lcf) and place .BINARY section into memory. Listing 14-1 on page 255 shows a sample linker command file with .BINARY section.

Listing 14-1. Linker Command File with .BINARY section

MEMORY
{
    resetvector: org = 0x00000000, len = 0x00000008
    init: org = 0x00000020, len = 0x00000FE0
    exception_handlers: org = 0x00001000, len = 0x00001000
    internal_flash: org = 0x00002000, len = 0x001FD000
    my_binary_data: org = 0x001FE000, len = 0x00001000

    ...
}

SECTIONS
{
    .__bam_bootarea LOAD (0x00000000): {} > resetvector
    ...
    .binary1_area:
    {
        binary1Start = .;
        bin_data1.bin
        binary1End = .;

    } > my_binary_data
}

.binary2_area:
{
    binary2Start = .;
    bin_data2.bin
    binary2End = .;

} > my_binary_data
}


14.1.1.2 Using Command-Line

To link a binary file using Command line, perform the following steps:

1. Linker recognizes .bin extension as a binary data input file. If binary file has another extension it may not be recognized correctly by the command line linker.
2. Update linker command file (.lcf) and place .BINARY section into memory. Linker Command File with .BINARY section shows a sample linker command file with .BINARY section.
3. Add a binary file (.bin) as an input file for linker (MWLDEPPC.exe)

   mwldeppc main.o msl.lib bin_data.bin -o myapp.elf -lcf commandfile.lcf

14.2 Additional Small Data Sections

The PowerPC EABI specification mandates that compliant build tools predefine three small data sections. The Properties > C/C++ Build > Settings > Tool Settings > PowerPC Linker > Input panel lets you specify the address at which the CodeWarrior linker puts two of these sections (if the default locations are unsatisfactory).

CodeWarrior Development Studio, lets you create small data sections in addition to those mandated by the PowerPC EABI specification. The CodeWarrior tools let you specify that the contents of a given user-defined section will be accessed by the small data base register selected from the available non-volatile registers. To do this, you use a combination of source code statements and linker command file directives.

To create one additional small data area, follow these steps:

1. Open the CodeWarrior project in which you want to create an additional small data section.
2. Select the build target in which you want to create an additional small data section.
3. Select Project > Properties.

   The IDE displays the Properties for <Project> window.

4. In the left pane of the Properties for <Project>, select PowerPC Compiler > Preprocessor.
5. Click **Browse** to open the prefix file whose name appears in the Prefix File text box in an editor window.

6. Add the statements that define a small data section to the top of the prefix file:
   a. Add a statement that creates a global register variable.
      
      For example, to create a global register variable for register 14, add this statement to the prefix file:
      
      ```cpp
      // _dummy does not have to be defined extern int _dummy asm("r14");
      
      b. Create a user-defined section using the section pragma; include the clause `data_mode = sda_rel` so the section can use small data area addressing.
      
      For example:
      
      ```cpp
      // you do not have to use the names in this example
      // .red is the initialized part of the section
      // .blue is the uninitialized part of the section
      #pragma section RW ".red" ".blue"
      data_mode = sda_rel
      
      **NOTE**
      
      If you want your small data area to be the default section for all small data, use the following form of the section pragma instead of the one above:
      
      ```cpp
      #pragma section 
      sdata_type ".red" "blue" data_mode = sda_rel
      
      7. Save the prefix file and close the editor window.

8. In each header or source file that declares or defines a global variable that you want to put in a small data section, put the storage-class modifier `__declspec(section "initialized_small_sect_nm")` in front of the definition or declaration.

   For example, the statement:
   
   ```cpp
   __declspec(section ".red") int x = 5;
   
   instructs the compiler to put the global variable x into the small data section named .red
   
   **CAUTION**
   
   The section name specified in the `__declspec(section <section_name>)` statement must be the name of an initialized data section. It is an error to use the uninitialized data section name.

   **NOTE**
   
   The semantics of `__declspec(section ".sdata") int x;` is to use the section pair `.sdata` and `.sbss` to store x. The location where x is stored is determined by whether or not x is explicitly initialized.
NOTE

If you want your small data section to be the default section for all small data, use #pragma section sdata_type ".foo" ".bar" data_mode = sda_relUse __declspec(section ".foo") only when the object is greater than the size threshold for small data.

9. On the **Tool Settings** tab, select **PowerPC Linker > Input**.

10. Click **Browse** to specify a Link Command File (.lcf).

11. On the **Tool Settings** tab, select **PowerPC CPU**.

12. From the **Code Model** listbox, select **Absolute Addressing**.
13. From the **ABI** listbox, select **EABI**.
14. Click **OK**.

The IDE saves your settings and closes the **Properties for <Project>** window.

15. Modify the project's linker command file such that it instructs the linker to use the global register declared above as the base register for your new small data section.

To do this, follow these steps:

a. In the linker command file, add two REGISTER directives, one for the initialized part of the small data section and one for uninitialized part.

   For example, to make register 14 the base register, add statements like these:

   ```
   .red REGISTER(14) : {} > ram
   .blue REGISTER(14) : {} > ram
   ```

b. Add the linker command file to each build target in which you want to use the new small data section.

16. Open the CodeWarrior project for the runtime library used by your project.
17. In the build target listbox of the runtime library project window, select the build target of the runtime library that your main project uses.
18. Open this build target's prefix file in a CodeWarrior editor window.
19. Add the same statements to this prefix file that you added to the prefix file of the main project.
20. Save the prefix file and close the editor window.
22. Find the string __init_registers(void) and add statements that initialize the small data section base register you are using near the end of this function (immediately above the terminating blr instruction).

   For example, to initialize register 14, add these statements:
lis r14, _SDA14_BASE_@ha
addi r14, r14, _SDA14_BASE_@l
23. Save __start.c and close the editor window.
25. Find the string _SDA_BASE_[] in this file and add this statement after the block of
statements that follow this string:

```c
    // SDA_BASE is defined by the linker if
    // the REGISTER(nn) directive appears in the .lcf file
    __declspec(section "_.init") extern char _SDA14_BASE_[];
```
26. Save __ppc_eabi_linker.h and close the editor window.
27. Return to your main project.
28. Select **Project > Build Project.**

The IDE builds your project.

You can now use the new small data section in this project.

**NOTE**

You can create more small data segments by following the
procedure above. Remember, however, that for each small
data section created, the compiler loses one non-volatile
register to use for other purposes.

### 14.3 Linker Map File

A linker map file is a text file containing information about a program's global symbols,
source file and source line numbers. The linker names the map file with the base name of
the program and the extension .map. The linker map consists of the following sections:

- **Closure**
- **Section Layout**
- **Memory Map**
- **Linker Generated Symbols**

#### 14.3.1 Closure

The linker lists all the required objects under the closure section with the following
details:
• **Level of closure:** Object B is in Object A's closure if and only if, the level of B is higher than the level of A and one of the following conditions is true:

Condition 1: There is no object in between B and A.

Condition 2: There are objects between B and A, and the level of A is lower than the levels of all the objects between B and A.

• **Object name:** specifies the name of an object.

• **Object characteristics:** specifies the characteristics of an object. They can be one of the following:
  - function, local | global | weak
  - section, local | global | weak
  - object, local | global | weak
  - notype, local | global | weak

• **Object locations:** specifies an object location.

The listing below shows a sample closure section.

**Listing 14-2. Sample closure section**

1) reset (func,global) found in reset.o
2) __reset (func,global) found in 8568mds_init.o
3) __start (func,global) found in Runtime.PPCEABI.E2.UC.a __start.o
4) __init_registers (func,weak) found in Runtime.PPCEABI.E2.UC.a __start.o
5) _stack_addr found as linker generated symbol
5) _SDA2_BASE_ found as linker generated symbol
5) _SDA_BASE_ found as linker generated symbol
4) __init_hardware (func,global) found in __ppc_eabi_init.o
5) _usr_init (func,global) found in 8568mds_init.o
6) gInterruptVectorTable (notype,global) found in eppc_exception.o
7) gInterruptVectorTableEnd (notype,global) found in eppc_exception.o
7) .intvec (section,local) found in eppc_exception.o
8) InterruptHandler (func,global) found in interrupt.o
9) @21 (object,local) found in interrupt.o
9) printf (func,global) found in MSL_C.PPCEABI.bare.E2.UC.a printf.o
9) __mclz_32__ (func,global) found in MSL_C.PPCEABI.bare.E.a math_double.o
9) ____UNREFERENCED DUPLICATE __mclz_32__
9) ____func,weak) found in MSL_C.PPCEABI.bare.E.a math_f64.o
9) ____func,weak) found in MSL_C.PPCEABI.bare.E.a math_longdouble.o
9) ____func,weak) found in MSL_C.PPCEABI.bare.E.a math_ppc.o

In the sample above:

• __reset is in the closure of reset because:
  - __reset is of level 2, reset is of level 1 and
  - there is no object in between __reset and reset

• __SDA_BASE_ is in the closure of __init_registers because:
  - __SDA_BASE_ is of level 5, __init_registers is of level 4; and
  - the objects between __init_registers and __SDA_BASE_ are all of level 5 and are higher than the level of __init_registers

• InterruptHandler is in the closure of __init_hardware because:
InterruptHandler is of level 8, __init_hardware is of level 4; and
the objects between __init_hardware and InterruptHandler are of level 5, 6, 7
respectively and are all higher than the level of __init_hardware.

__init_hardware is NOT in the closure of __init_registers because they both are of level 4.
gInterruptVectorTableEnd is NOT in the closure of __init_registers because:
the objects between gInterruptVectorTableEnd and __init_registers are not all of a
higher level than __init_registers.
__init_hardware is of the same level as __init_registers.

Weak symbols are allowed by the ABI and are global. They can have the same name as
another symbol. The line before the UNREFERENCED DUPLICATE lists the first weak
symbol found by the linker, that appears in the executable.

The line after the UNREFERENCED DUPLICATE lists other versions of a same object
found by the linker. Linker will not copy the duplicate objects to the executable.

14.3.2 Section Layout

The linker lists information of all the objects within a section in a section layout. Listing
14-3 on page 261 shows a sample .text section layout.

Listing 14-3. Sample .text section layout

 Starting Virtual File
 address Size address offset
------------------------- ------------- -------------
00000084 000030 fffc1964 00001ce4 1  .text
00000084 00000c fffc1964 00001ce4 4 __init_user Runtime.PPCEABI.E2.UC.a CPlusLibPPC.o
00000090 000020 fffc1970 00001cf0 4 exit Runtime.PPCEABI.E2.UC.a CPlusLibPPC.o
000000b0 000004 fffc1990 00001d10 4 ExitProcess Runtime.PPCEABI.E2.UC.a CPlusLibPPC.o

The first line of a section layout specifies the name of a section. Starting from the 5th line
(the line after the dotted line separator), objects within section are listed with the
following information:

• Starting address: specifies the starting address of an object. The object is listed as
  UNUSED if it is dead-stripped.
• Size: specifies the size of each object in a section.
• Virtual address: specifies the virtual address of the object.
• File offset: specifies the offset of an object in the section.
• **Alignment**: specifies the alignment of an object. For legacy reasons, the alignment of all section symbols is 1. In reality, a section symbol is the highest alignment of all symbols in its section which in the above listing is 4.

• **Object name**: specifies the name of an object. The names are the C or mangled C++, depending on the language. The name of an object is similar to the one in a disassembled file.

• **Object location**: specifies the location an object. This is usually a name of the object file (.o), when no other column exists. In presence of an other column, the library file information is listed here.

In the sample above, note that the 5th line has a section name and that its Starting address, Virtual address and File offset values are similar to __init_user values. The 5th line is the section symbol for the objects that follow it. Its Size column is the total size of the section in the executable file (after dead stripping) and its alignment (column 5) is 1.

Although **Listing 14-3 on page 261** is from a normal .o file, if this project had an input file which was a partially linked file, then you would see a section symbol between each merged section from each input file used to create the partially linked file. For example, if plf.o is a partially linked file, and it is composed of a.o and b.o and each of those files had a .text section, then plf.o would have one .text section but two .text section symbols. The .text content from a.o would follow the first section symbol and the content from b.o would follow the second section symbol.

### 14.3.3 Memory Map

You can verify segment allocation from the Memory map section in a linker map file. **Listing 14-4 on page 262** shows a sample Memory map section.

<table>
<thead>
<tr>
<th></th>
<th>Starting address</th>
<th>Size</th>
<th>File Offset</th>
<th>ROM Address</th>
<th>RAM Buffer Address</th>
<th>S-Record Line</th>
<th>Offset</th>
<th>Bin File Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>.init</td>
<td>00000000</td>
<td>00000000</td>
<td>00000368</td>
<td>00000010</td>
<td>00000020</td>
<td>3</td>
<td>00000000</td>
<td>Test.00000010.bin</td>
</tr>
<tr>
<td>.init_vle</td>
<td>00000000</td>
<td>00000000</td>
<td>00000418</td>
<td>00000050</td>
<td>00000000</td>
<td>0</td>
<td>00000000</td>
<td>Test.00000000.bin</td>
</tr>
<tr>
<td>.text</td>
<td>00040000</td>
<td>00000000</td>
<td>00000dfe</td>
<td>00000000</td>
<td>00000000</td>
<td>9</td>
<td>00000000</td>
<td>Test.00000000.bin</td>
</tr>
<tr>
<td>.text_vle</td>
<td>00040000</td>
<td>00000000</td>
<td>00000e00</td>
<td>00000000</td>
<td>00000000</td>
<td>140</td>
<td>00000000</td>
<td>Test.00000000.bin</td>
</tr>
</tbody>
</table>

**Listing 14-4. Verifying segment allocation in a .MAP file**

- **Starting address**: specifies the starting address for each section. Constant and executable object code are allocated in ROM space and data object code is allocated in RAM space.
- **Size**: specifies the size of each sections.
- **File offset**: specifies the offset of a section in the file.
- **ROM Address**: specifies the address of the section in the ROM image. For executable code and constants sections, Starting address is equal to ROM Address. For data
sections. **ROM Address** specifies the address in ROM where the initialization values are stored.

- **RAM Buffer Address**: specifies the address in RAM that is to be used as a buffer for the flash image programmer. It is important to note that the RAM buffer is not used when the RAM address equals to the ROM address.
- **S-Record Line**: specifies the line number of a section in the S-Record file in decimal format.
- **Bin File Offset**: specifies the offset of a section in the binary file.
- **Bin File Name**: specifies the binary file name of the section. The file name also reflects the ROM address of the section.

### 14.3.4 Linker Generated Symbols

You can find a complete list of the linker generated symbols and user-defined symbols in either the C include file `__ppc_eabi_linker.h` or the assembly include file `__ppc_eabi_linker.i`. The CodeWarrior linker automatically generates symbols for the start address, the end address (the first byte after the last byte of the section), and the start address for the section if it will be burned into ROM. With a few exceptions, all CodeWarrior linker-generated symbols are immediate 32 bit values. **Listing 14-5 on page 263** shows a sample list of linker-generated symbols.

**Listing 14-5. Sample list of linker-generated symbols**

```plaintext
    _f_init                 000034d8
    _f_init_rom                 000034d8
    _e_init                 000035b0
    _f_init_vle                 000035b0
    _f_init_vle_rom                 000035b0
    _e_init_vle                 00003864
    _f_text                 00003864
    _f_text_rom                 00003864
    _e_text                 00003864
    _f_text_vle                 00003870
    _f_text_vle_rom                 00003870
    _e_text_vle                 00003ad4
```

If addresses are declared in your source file as `unsigned char _f_text[]`; you can treat `_f_text` just as a C variable even though it is a 32-bit immediate value.
unsigned int textsize = _e_text - _f_text;

If you do need linker symbols that are not addresses, you can access them from C.

unsigned int size = (unsigned int)&_text_size;

The linker generates four symbols:

- __ctors - an array of static constructors
- __dtors - an array of destructors
- __rom_copy_info - an array of a structure that contains all of the necessary information about all initialized sections to copy them from ROM to RAM
- __bss_init_info - a similar array that contains all of the information necessary to initialize all of the bss-type sections. Please see __init_data in __start.c.

These four symbols are actually not 32-bit immediate values but are variables with storage. You access them just as C variables. The startup code now automatically handles initializing all bss type sections and moves all necessary sections from ROM to RAM, even for user defined sections.

14.4 Deadstripping

If the Pool Data checkbox is checked in the CodeWarrior IDE's Properties > C/C++ Build > Settings > Tool Settings > PowerPC Compiler > Processor panel, the pooled data is not stripped. However, all small data and code is still subject to deadstripping.

14.5 CodeWarrior Linker Command File (LCF)

Linker command files are an alternative way of specifying segment addresses.

Only one linker command file is supported per target in a project. The linker command filename must end in the .lcf extension.

The LCF along with other compiler directives, places pieces of code and data into ROM and RAM. You can do this by creating specific sections in the LCF and then matching them to the source code using pragma directives.

LCF consists of three kinds of segments, which must be in this order:

- A memory segment, which begins with the MEMORY[] directive,
• An optional closure segment, which begins with the \texttt{FORCE_ACTIVE{}}, \texttt{KEEP_SECTION{}} , or \texttt{REF_INCLUDE{}} directives, and
• A sections segment, which begins with the \texttt{SECTIONS{}} directive.

14.6 Creating an LCF from Scratch

Consider creating a LCF for a sample \texttt{BOOKE ROM} project.

First add the memory area for vectors (interrupts), exception tables, code, data, heap and stack.

\textbf{Listing 14-6. Adding memory area}

\begin{verbatim}
MEMORY {
    resetvector:        org = 0x00000000,   len = 0x00000008
    exception_handlers_p0: org = 0x00001000,   len = 0x00001000
    internal_flash:        org = 0x00003000,   len = 0x001FD000
    internal_ram:          org = 0x40000000,   len = 0x0007C000
    heap :                org = 0x4007C000,   len = 0x00002000 /*  Heap start location */
    stack :                org = 0x4007E000,   len = 0x00002000 /* Start location for Stack */
}
\end{verbatim}

Place the sections to the above memory areas in LCF in the \texttt{SECTIONS { } } block.

\texttt{.\_\_\_bam\_bootarea LOAD (0x00000000): {} > resetvector}

The code to handle exceptions are grouped and placed in memory area \texttt{exception_handlers_p0}.

\textbf{Listing 14-7. Grouping exceptions code}

\begin{verbatim}
GROUP : {
    .ivor\_branch\_table\_p0 LOAD (0x00001000) : {}
    .intc\_hw\_branch\_table\_p0 LOAD (0x00001800): {} 
    .\_\_\_exception\_handlers\_p0 LOAD (0x00001100) : {}
} > exception\_handlers\_p0
\end{verbatim}
The hardware initialization routines, application code, constants, code for constructors/destructors, and C++ exception tables are grouped together and placed in Flash.

**Listing 14-8. Grouping initialization routines**

```assembly
GROUP : {
   .intc_sw_isr_vector_table_p0 ALIGN (2048) : {}
   .init : {}
   .text : {}
   .rodata (CONST) : {
      *(.rdata)
      *(.rodata)
   }
   .ctors : {}
   .dtors : {}
   extab : {}
   extabindex : {}
} > internal_flash
```

The uninitialized and initialized data are placed in RAM.

**Listing 14-9. Placing data in RAM**

```assembly
GROUP : {
   __uninitialized_intc_handler_table ALIGN(2048) : {}
   .data : {}
   .sdata : {}
   .sbss : {}
   .sdata2 : {}
   .sbss2 : {}
   .bss : {}
} > internal_ram
```

The sections are allocated to segments in the order given in `SECTIONS/GROUP` block of lcf file.

For `internal_flash` segment, following is the order of section allocation: .init, .text, .rodata, .ctors, .dtors, extab and extabindex.

Variables are added in LCF and these can be used in application as well as internally in linker tool for computation.

**Listing 14-10. Adding variables in LCF**

```assembly
_stack_addr = ADDR(stack)+SIZEOF(stack);
_stack_end  = ADDR(stack);
_heap_addr  = ADDR(heap);
_heap_end   = ADDR(heap)+SIZEOF(heap);
EXCEPTION_HANDLERS = ADDR(exception_handlers_p0);
L2SRAM_LOCATION = 0x40000000;
```
Let us take a simple example to see how the allocation of variables to the respective sections take place.

Listing 14-11. C Source file

```c
#include "MPC5675K.h"
int sdata_i = 10;
int sbss_i;
const char sdata2_array[] = "Hello";
__declspec(section ".rodata") const char rodata_array[40]="CodeWarior";
__declspec(section ".data") long bss_i;
__declspec(section ".data") long data_i = 10;
int main(void) {
    return sdata_i + sbss_i + sdata2_array[3] + data_i + bss_i + rodata_array[5];
}
```

NOTE

Above is a hypothetical example built to provide clarity on variables and their allocation to sections. __declspec is used to forcefully place the variables into sections.

The table below lists the details of the allocation of objects to the sections.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Section</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>sdata_i</td>
<td>.sdata</td>
<td>0x400004d8</td>
</tr>
<tr>
<td>sbss_i</td>
<td>.sbss</td>
<td>0x400004e8</td>
</tr>
<tr>
<td>sdata2_array</td>
<td>.sdata2</td>
<td>0x40000500</td>
</tr>
<tr>
<td>rodata_array</td>
<td>.rodata</td>
<td>0x00003938</td>
</tr>
<tr>
<td>bss_i</td>
<td>.bss</td>
<td>0x40000508</td>
</tr>
<tr>
<td>data_i</td>
<td>.data</td>
<td>0x400004d0</td>
</tr>
</tbody>
</table>

14.7 Relocating Code in ROM

To place data and code in a specific memory location there are two general steps that must be performed.
Use pragma compiler directives to tell the compiler which part of the code is going to be relocated.

Tell the linker where the code will be placed within the memory map using LCF definitions.

### 14.7.1 Relocating Function in ROM

To put code in a specific memory section it is needed first to create the section using the section pragma directive. In the following listing a new section called `.romsymbols` is created.

All the content in this section is going to be referenced in the LCF with the name `.romsymbols`. After defining a new section you can place code in this section by using the `__declspec()` directive.

In the following listing, `__declspec()` directive is used to tell the compiler that function `funcInROM()` is going to be placed in section `.romsymbols`.

Create a stationary project for any target and add the following code to your `main.c` file before the `main()` function and have a call to this function.

**Listing 14-12. Code to add in the main.c**

```c
#pragma section RX "romsymbols" data_mode=far_abs
__declspec(section "romsymbols") void funcInROM(int flag); //Function Prototype

void funcInROM(int flag){
    if (flag > 0)
    {
        flag ++;
    }
}
```

### 14.7.2 Placing Code in ROM

You have just edited a source file to tell the compiler which code will be relocated. Next, the LCF needs to be edited to tell the linker the memory addresses where these sections are going to be allocated.
First you need to define a new Microcontroller memory segment where new sections will be allocated.

You can have just one memory segment for all the new sections or one segment for each section.

### 14.7.2.1 Create New ROM Segment

Below you can find the memory segment of a LCF. Notice that the segment `internal_flash` has been edited and its length has been reduced by `0x10000` from its original size. This memory space is taken to create the new segment. In the following listing the new segment is called `myrom`, it will be located next to segment `internal_flash` and its length is going to be `0x10000`. You can calculate the address where segment code ends by adding its length plus the origin address.

Edit your LCF as shown in the following listing. Ensure you edit ROM target lcf.

**Listing 14-13. Memory Segment of LCF**

```
MEMORY
{
    resetvector:       org = 0x00000000,  len = 0x00000008
    init:             org = 0x00000010,  len = 0x0000FFF0
    exception_handlers_p0: org = 0x00010000,  len = 0x00010000
    internal_flash:  org = 0x00030000,  len = 0x001C0000
    myrom:           org = 0x00220000,  len = 0x00010000
    internal_ram:    org = 0x40000000,  len = 0x0007C000
    heap :           org = 0x4007C000,  len = 0x00002000 /* z7_0 Heap start location */
    stack :          org = 0x4007E000,  len = 0x00002000 /* z7_0 Start location for Stack */
}
```

### 14.7.2.2 Create New ROM Section

The next step is to add the content of the new section into the Microcontroller memory segment you have reserved. This is done in the sections segment of the LCF.
The code below creates a new section called `.rom_symbols`, then the label `__ROM_SYMBOLS` points to the address where the section begins. Then `*(.romsymbols)` instruction is used to tell the linker that all the code referenced with this word is going to be placed in section `.rom_symbols`.

Finally you close the section telling the linker that this content is going to be located in segment `myrom`.

Edit your LCF as shown below.

**Listing 14-14. Code to add your LCF.**

```assembly
.rom_symbols :
{
  __ROM_SYMBOLS = . ;             #start address of the new symbol area
  . = ALIGN (0x4);
  *(.romsymbols)                  #actual data matching pragma directives.
  . = ALIGN (0x4);
} > myrom
```

Please note that in the MAP file you can notice newly created ROM section.

### 14.8 Relocating Code and Data in Internal RAM

Since it is not possible to write a variable in ROM, data must be relocated in RAM. Code can be also relocated in RAM. Another reason to relocate code in RAM is that it is twice as fast as in Flash.

To relocate code and data in internal RAM, create a new section using section pragma directive and `__declspec` directives as shown in the listing below.

**Listing 14-15. Using pragma Directives to Define a Section**

```assembly
#pragma section  "myCodeInRAM" data_mode=far_abs
__declspec(section "myCodeInRAM")
struct {
  unsigned char data0;
  unsigned char data1;
  unsigned char data2;
  unsigned char data3;
  unsigned char data4;
  unsigned char data5;
```
unsigned char data6;
unsigned char data7;
} CTMData = { 0x82, 0x65, 0x77, 0x32, 0x84, 0x69, 0x83, 0x84 };
__declspec(section "myCodeInRAM") void funcInROM(int flag);
void funcInROM(int flag){
if (flag > 0)
{
flag++;
}
}

14.8.1 Placing Code and Data in RAM

Placing code and data into RAM is more complicated. As the content in RAM cannot be saved when turning power off, you first need to save the code and data in flash and then make a copy to RAM in runtime.

Following are the steps to relocate code and data in a new RAM segment.

1. Create New RAM Segment
2. Create New RAM Section

14.8.1.1 Create New RAM Segment

As it was made for the new ROM segment, a piece of the user ram memory segment is taken to create a new memory segment called myram.

Edit your LCF as shown in Listing 5.

Listing 14-16. Memory Segment of LCF

MEMORY
{
exception_handlers_p0: org = 0x40000000, len = 0x00001000
pseudo_rom: org = 0x40001000, len = 0x00006800
init: org = 0x40007800, len = 0x00000800
internal_ram: org = 0x40008000, len = 0x00070000
}
14.8.1.2 Create New RAM Section

The memory segment specifies the intended location in RAM. The code below shows a new section called .my_ram which is going to be linked in segment .myram but is going to be resident in the Flash memory address calculated by label ___CodeStart. This label is intended to find the first address available in flash.

In the listing section .app_text the linker places in the segment code all the code and then the read only data. After this it sets a label called __ROM_AT. Section .data is allocated in the address pointed by this label.

Add the following code to LCF. You can put this code just after Placing data in RAM. The uninitialized and initialized data are placed in RAM.

Listing 14-17. Add this Code to LCF after Listing D.

___CodeStart = ___RAM_end;
.my_ram :
{
  . = ALIGN (0x4);
  ___myRAMStart = .;
  *(.myCodeInRAM)
  ___myRAMEnd = .;
  . = ALIGN (0x4);
} > myram

14.9 Relocating Code and Data in External MRAM
Many times the internal RAM in the Microcontroller you are using is not enough for the application. For this reason it is needed to use external memories as part of the solution. The process to relocate code and data in external memories is exactly the same as you did for internal RAM. The only difference is that the external device needs to be communicated by an interface controller.

14.10 Unique LCF Examples

This topic describes the following LCF examples.

- Configuring Linker File to Several ROM Blocks
- Place the Library File in the LCF
- Place Symbols in Specific Memory Location
- How to Relocate Code in RAM

14.10.1 Configuring Linker File to Several ROM Blocks

The following listing is an example to configure linker file to several ROM blocks.

```
Listing 14-18. ROM IMAGE address = 0x3000

MEMORY{
  internal_flash:  org = 0x00003000,  len = 0x0010000

MyFlash:  org = 0x00041000,  len = 0x00000008

  //org should match the LOAD address
}

SECTIONS{
  .text {} > internal_flash

  .my_flash  ALIGN(0x08) LOAD(0x00041000) : {} > MyFlash

}
```

14.10.2 Place the Library File in the LCF
The following listing is an example to place the library file in the LCF.

**Listing 14-19. Placing library file in LCF**

```c
GROUP : {
  .libcode (VLECODE) LOAD (0x00004000) : {
    Runtime.PPCEABI.VS.UC.a (.text)
  }
  .libconst:
  {
    Runtime.PPCEABI.VS.UC.a (.rodata)
  }
} > lib_flash
```

**NOTE**

For small data sections, ctors, dtors section it's not allowed to have different output section name.

### 14.10.3 Place Symbols in Specific Memory Location

For placing the symbols in specific memory location, user has to define the memory region (say `Memory_to_store`) in the lcf file and also define a new section (say `.user_defined_section`) then use the same section in the source file to place the symbol.

**Listing 14-20. Example for initialized variable**

```c
In the source file:
#pragma section <section_qualifier(R,RW)> ".user_defined_section"
__declspec(section ".user_defined_section") int temp = 5;
```

In the LCF file:

```c
GROUP : {
  .user_defined_section :{}
  } > Memory_to_store // Memory_to_store is the memory area where user want to store
```

**Listing 14-21. Example for uninitialized variable**

```c
In the source file:
#pragma section ".user_defined_section".data
__declspec(section ".user_defined_section") /* We cannot have an uninitialized section name in The //uninitialized section must be paired with initialized section. */
__declspec(section ".user_defined_section") int temp;
```
In the LCF file:

GROUP : {
  .user_defined_section :{}
} > Memory_to_store

### 14.10.4 How to Relocate Code in RAM

The following listings provide sample code examples to relocate the code in RAM.

**Listing 14-22. Sample code in the source file**

```c
#pragma section "myCodeInRAM" code_mode=far_abs
__declspec(section "myCodeInRAM") int _add(int a , int b);
int main(void) {
    volatile int i = 0;
    volatile int total = 0;
    /* Loop forever */
    for (;;) {
        total = _add(i , i);
        i++;
    }
}
__declspec(section "myCodeInRAM") int _add(int a , int b)
{
    return a + b;
}
```

**Listing 14-23. Sample code in the lcf file**

```c
MEMORY
{
    ............
    /* SRAM: 0x40000000 - 0x4000FFFF */
    internal_ram: org = 0x40000000, len = 0x0000D000
    myram: org = 0x4000D000, len = 0x00001000
    ............
}
```
14.11 Linker Command File Commands

The CodeWarrior Power Architecture linker supports these additional commands listed below:

- AGGRESSIVE_MERGE
- AGGRESSIVE_MERGE_FILES
- AGGRESSIVE_MERGE_SECTIONS
- DO_NOT_MERGE
- DO_NOT_MERGE_FILES
- DO_NOT_MERGE_SECTIONS
- INIT
- FORCEFILES
- SHORTEN_NAMES_FOR_TOR_101

14.11.1 AGGRESSIVE_MERGE

Specifies functions that should be considered for aggressive merging when applying the code merging optimization.

Syntax

```
DO_NOT_MERGE {
  symbol [, symbol ]*
}
```

14.11.2 AGGRESSIVE_MERGE_FILES
Specifies that all functions in object code files should be considered for aggressive merging when applying the code merging optimization.

**Syntax**

```markdown
DO_NOT_MERGE_FILES {
  file-name [, file-name ]*
}
```

### 14.11.3 AGGRESSIVE_MERGE_SECTIONS

Specifies that all functions in object code sections should be considered for aggressive merging when applying the code merging optimization.

**Syntax**

```markdown
AGGRESSIVE_MERGE_SECTIONS {
  section-name [, section-name ]*
}
```

### 14.11.4 DO_NOT_MERGE

Specifies functions that should not be removed when applying the code merging optimization.

**Syntax**

```markdown
DO_NOT_MERGE {
  symbol [, symbol ]*
}
```

**Remarks**

This directive specifies functions that the linker should keep in the output file when applying the code merging optimization even if other functions with identical object code exist.

### 14.11.5 DO_NOT_MERGE_FILES
Specifies that all functions in a file should not be removed when applying the code merging optimization.

Syntax

```
DO_NOT_MERGE_FILES {
  file-name [,
  file-name ]*
}
```

14.11.6  DO_NOT_MERGE_SECTIONS

Specifies that all functions in an object code section should not be removed when applying the code merging optimization.

Syntax

```
DO_NOT_MERGE_SECTIONS {
  section-name [,
  section-name ]*
}
```

14.11.7  INIT

Defines the initialization entry point for the application.

Syntax

```
INIT (FunctionName)
```

Remarks

This command is mandatory for assembly application and optional otherwise. It cannot be specified more than once in the prm file. When you specify the INIT command in the prm file, the linker uses the specified function as application entry point. This is either the main routine or a startup routine calling the main routine.

When INIT is not specified in the prm file, the linker looks for a function named __start and uses it as the application entry point.

Example

```
INIT (MyGlobStart) /* Specify a global variable as application entry point.*/
```
14.11.8  FORCEFILES

Specifies that the contents of object code files must not be deadstripped.

Syntax

FORCEFILES { file-name
    [,
    file-name ]* }

Remarks

Use FORCEFILES to list source files, archives, or archive members that you do not want
dead-stripped. All objects in each of the files are included in the linker's output file even
if the linker has determined that those objects are not referenced by other objects.

If you only have a few symbols that you do not want deadstripped, use FORCEACTIVE.

14.11.9  SHORTEN_NAMES_FOR_TOR_101

The directive SHORTEN_NAMES_FOR_TOR_101 instructs the linker to shorten long template names
for the benefit of the WindRiver® Systems Target Server. To use this directive, simply
add it to the linker command file on a line by itself.

SHORTEN_NAMES_FOR_TOR_101

WindRiver Systems Tornado Version 1.0.1 (and earlier) does not support long template
names as generated for the MSL C++ library. Therefore, the template names must be
shortened if you want to use them with these versions of the WindRiver Systems Target
Server.
Chapter 15
C Compiler

This chapter explains the CodeWarrior implementation of the C programming language:

- Extensions to Standard C
- C99 Extensions
- GCC Extensions

15.1 Extensions to Standard C

The CodeWarrior C compiler adds extra features to the C programming language. These extensions make it easier to port source code from other compilers and offer some programming conveniences. Note that some of these extensions do not conform to the ISO/IEC 9899-1990 C standard ("C90").

- Controlling Standard C Conformance
- C++-style Comments
- Unnamed Arguments
- Extensions to the Preprocessor
- Non-Standard Keywords
- Declaring Variables by Address

15.1.1 Controlling Standard C Conformance

The compiler offers settings that verify how closely your source code conforms to the ISO/IEC 9899-1990 C standard ("C90"). Enable these settings to check for possible errors or improve source code portability.
Some source code is too difficult or time-consuming to change so that it conforms to the ISO/IEC standard. In this case, disable some or all of these settings.

**Table 15-1** shows how to control the compiler's features for ISO conformance.

<table>
<thead>
<tr>
<th>To control this option from here...</th>
<th>use this setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeWarrior IDE</td>
<td>ANSI Strict and ANSI Keywords Only in the C/C++ Language Settings panel</td>
</tr>
<tr>
<td>source code</td>
<td>#pragma ANSI_strict#pragma only_std_keywords</td>
</tr>
<tr>
<td>command line</td>
<td>-ansi</td>
</tr>
</tbody>
</table>

### 15.1.2 C++-style Comments

When ANSI strictness is off, the C compiler allows C++-style comments. **Listing 15-1 on page 282** shows an example.

#### Listing 15-1. C++ Comments

```c
a = b;    // This is a C++-style comment.
c = d;    // This is a regular C-style comment. */
```

### 15.1.3 Unnamed Arguments

When ANSI strictness is off, the C compiler allows unnamed arguments in function definitions. **Listing 15-2 on page 282** shows an example.

#### Listing 15-2. Unnamed Arguments

```c
void f(int) {} /* OK if ANSI Strict is disabled. */
void f(int i) {} /* Always OK. */
```

### 15.1.4 Extensions to the Preprocessor
When ANSI strictness is off, the C compiler allows a # to prefix an item that is not a macro argument. It also allows an identifier after an #endif directive. Listing 15-3 on page 283 and Listing 15-4 on page 283 show examples.

### Listing 15-3. Using # in Macro Definitions

```
#define add1(x) #x #1
   /* OK, if ANSI_strict is disabled,
       but probably not what you wanted:
       add1(abc) creates "abc="#1
   */
```

```
#define add2(x) #x "2"
   /* Always OK: add2(abc) creates "abc2". */
```

### Listing 15-4. Identifiers After #endif

```
#ifdef __CWCC__
   /* . . . */
#endif __CWCC__ /* OK if ANSI_strict is disabled. */
```

```
#ifdef __CWCC__
   /* . . . */
#endif /*__CWCC__*/ /* Always OK. */
```

### 15.1.5 Non-Standard Keywords

When the ANSI keywords setting is off, the C compiler recognizes non-standard keywords that extend the language.

### 15.1.6 Declaring Variables by Address

The C compiler lets you explicitly specify the address that contains the value of a variable. For example, the following definition states that the variable `MemErr` contains the contents of the address `0x220`:

```
short MemErr:0x220;
```

You cannot disable this extension, and it has no corresponding pragma or setting in a panel.
15.2 C99 Extensions

The CodeWarrior C compiler accepts the enhancements to the C language specified by the ISO/IEC 9899-1999 standard, commonly referred to as "C99."

• Controlling C99 Extensions
• Trailing Commas in Enumerations
• Compound Literal Values
• Designated Initializers
• Predefined Symbol __func__
• Implicit Return From main()
• Non-constant Static Data Initialization
• Variable Argument Macros
• Extra C99 Keywords
• C++-Style Comments
• C++-Style Digraphs
• Empty Arrays in Structures
• Hexadecimal Floating-Point Constants
• Variable-Length Arrays
• Unsuffixed Decimal Literal Values
• C99 Complex Data Types

15.2.1 Controlling C99 Extensions

Table 15-2 shows how to control C99 extensions.

<table>
<thead>
<tr>
<th>To control this option from here...</th>
<th>use this setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeWarrior IDE</td>
<td>Enable C99 Extensions in the C/C++ Language Settings panel</td>
</tr>
<tr>
<td>source code</td>
<td>#pragma c99</td>
</tr>
<tr>
<td>command line</td>
<td>-c99</td>
</tr>
</tbody>
</table>

15.2.2 Trailing Commas in Enumerations
When the C99 extensions setting is on, the compiler allows a comma after the final item in a list of enumerations. Listing 15-5 on page 285 shows an example.

**Listing 15-5. Trailing comma in enumeration example**

```c
enum
{
    violet,
    blue
    green,
    yellow,
    orange,
    red, /* OK: accepted if C99 extensions setting is on. */
};
```

### 15.2.3 Compound Literal Values

When the C99 extensions setting is on, the compiler allows literal values of structures and arrays. Listing 15-6 on page 285 shows an example.

**Listing 15-6. Example of a Compound Literal**

```c
#pragma c99 on
struct my_struct {
    int i;
    char c[2];
} my_var;

my_var = ((struct my_struct) {x + y, 'a', 0});
```

### 15.2.4 Designated Initializers

When the C99 extensions setting is on, the compiler allows an extended syntax for specifying which structure or array members to initialize. Listing 15-7 on page 285 shows an example.

**Listing 15-7. Example of Designated Initializers**

```c
#pragma c99 on
struct X {
    int a,b,c;
} x = { .c = 3, .a = 1, 2 };
union U {
    char a;
    `t`
15.2.5 Predefined Symbol __func__

When the C99 extensions setting is on, the compiler offers the __func__ predefined variable. Listing 15-8 on page 286 shows an example.

Listing 15-8. Predefined symbol __func__

```c
void abc(void)
{
    puts(__func__); /* Output: "abc" */
}
```

15.2.6 Implicit Return From main()

When the C99 extensions setting is on, the compiler inserts this statement at the end of a program's main() function if the function does not return a value:

```c
return 0;
```

15.2.7 Non-constant Static Data Initialization

When the C99 extensions setting is on, the compiler allows static variables to be initialized with non-constant expressions.

15.2.8 Variable Argument Macros

When the C99 extensions setting is on, the compiler allows macros to have a variable number of arguments. Listing 15-9 on page 286 shows an example.

Listing 15-9. Variable argument macros example
#define MYLOG(...) fprintf(myfile, __VA_ARGS__)
#define MYVERSION 1
#define MYNAME "SockSorter"

int main(void)
{
    MYLOG("%d %s\n", MYVERSION, MYNAME);
    /* Expands to: fprintf(myfile, "%d %s\n", 1, "SockSorter"); */

    return 0;
}

15.2.9 Extra C99 Keywords

When the C99 extensions setting is on, the compiler recognizes extra keywords and the language features they represent. Table 15-3 lists these keywords.

<table>
<thead>
<tr>
<th>This keyword or combination of keywords...</th>
<th>represents this language feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>_Bool</td>
<td>boolean data type</td>
</tr>
<tr>
<td>long long</td>
<td>integer data type</td>
</tr>
<tr>
<td>restrict</td>
<td>type qualifier</td>
</tr>
<tr>
<td>inline</td>
<td>function qualifier</td>
</tr>
<tr>
<td>_Complex</td>
<td>complex number data type</td>
</tr>
<tr>
<td>_Imaginary</td>
<td>imaginary number data type</td>
</tr>
</tbody>
</table>

15.2.10 C++-Style Comments

When the C99 extensions setting is on, the compiler allows C++-style comments as well as regular C comments. A C++-style comment begins with

```
//
```

and continues until the end of a source code line.

A C-style comment begins with

```
/*
```

ends with
and may span more than one line.

### 15.2.11 C++-Style Digraphs

When the C99 extensions setting is on, the compiler recognizes C++-style two-character combinations that represent single-character punctuation. Table 15-4 lists these digraphs.

<table>
<thead>
<tr>
<th>This digraph</th>
<th>is equivalent to this character</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;:</td>
<td>[</td>
</tr>
<tr>
<td>::</td>
<td>]</td>
</tr>
<tr>
<td>&lt;%%</td>
<td>{</td>
</tr>
<tr>
<td>%&gt;</td>
<td>}</td>
</tr>
<tr>
<td>%:</td>
<td>#</td>
</tr>
<tr>
<td>%:%:</td>
<td>##</td>
</tr>
</tbody>
</table>

### 15.2.12 Empty Arrays in Structures

When the C99 extensions setting is on, the compiler allows an empty array to be the last member in a structure definition. Listing 15-10 on page 288 shows an example.

Listing 15-10. Example of an Empty Array as the Last struct Member

```c
struct {
  int r;
  char arr[];
} s;
```

### 15.2.13 Hexadecimal Floating-Point Constants
Precise representations of constants specified in hexadecimal notation to ensure an accurate constant is generated across compilers and on different hosts. The compiler generates a warning message when the mantissa is more precise than the host floating point format. The compiler generates an error message if the exponent is too wide for the host float format.

Examples:

0x2f.3a2p3

0xEp1f

0x1.8p0L

The standard library supports printing values of type `float` in this format using the "%a" and "%A" specifiers.

15.2.14  Variable-Length Arrays

Variable length arrays are supported within local or function prototype scope, as required by the ISO/IEC 9899-1999 ("C99") standard. Listing 15-11 on page 289 shows an example.

Listing 15-11. Example of C99 Variable Length Array usage

```c
#pragma c99 on

void f(int n) {
    int arr[n];
    /* ... */
}
```

While the example shown in Listing 15-12 on page 289 generates an error message.

Listing 15-12. Bad Example of C99 Variable Length Array usage

```c
#pragma c99 on
int n;
int arr[n];
// ERROR: variable length array
// types can only be used in local or
// function prototype scope.
```
A variable length array cannot be used in a function template's prototype scope or in a local template typedef, as shown in Listing 15-13 on page 290.

Listing 15-13. Bad Example of C99 usage in Function Prototype

```c
#pragma c99 on

template<typename T> int f(int n, int A[n][n]);
{
};
// ERROR: variable length arrays
// cannot be used in function template prototypes
// or local template variables
```

15.2.15 Unsuffixed Decimal Literal Values

Listing 15-14 on page 290 shows an example of specifying decimal literal values without a suffix to specify the literal's type.

Listing 15-14. Examples of C99 Unsuffixed Constants

```c
#pragma c99 on  // Note: ULONG_MAX == 4294967295
sizeof(4294967295) == sizeof(long long)
sizeof(4294967295u) == sizeof(unsigned long)
#pragma c99 off
sizeof(4294967295) == sizeof(unsigned long)
sizeof(4294967295u) == sizeof(unsigned long)
```

15.2.16 C99 Complex Data Types

The compiler supports the C99 complex and imaginary data types when the C99 extensions option is enabled. Listing 15-15 on page 290 shows an example.

Listing 15-15. C99 Complex Data Type

```c
#include <complex.h>
complex double cd = 1 + 2*I;
```

NOTE

This feature is currently not available for all targets. Use #if __has_feature(C99_COMPLEX) to check if this feature is available for your target.
15.3 GCC Extensions

The CodeWarrior compiler accepts many of the extensions to the C language that the GCC (Gnu Compiler Collection) tools allow. Source code that uses these extensions does not conform to the ISO/IEC 9899-1990 C ("C90") standard.

- Controlling GCC Extensions
- Initializing Automatic Arrays and Structures
- The sizeof() Operator
- Statements in Expressions
- Redefining Macros
- The typeof() Operator
- Void and Function Pointer Arithmetic
- The __builtin_constant_p() Operator
- Forward Declarations of Static Arrays
- Omitted Operands in Conditional Expressions
- The __builtin_expect() Operator
- Void Return Statements
- Minimum and Maximum Operators
- Local Labels

15.3.1 Controlling GCC Extensions

Table 15-5 shows how to turn GCC extensions on or off.

<table>
<thead>
<tr>
<th>To control this option from here...</th>
<th>use this setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeWarrior IDE</td>
<td>Enable GCC Extensions in the C/C++ Language Settings panel</td>
</tr>
<tr>
<td>source code</td>
<td>#pragma gcc_extensions</td>
</tr>
<tr>
<td>command line</td>
<td>-gcc_extensions</td>
</tr>
</tbody>
</table>

15.3.2 Initializing Automatic Arrays and Structures
GCC Extensions

When the GCC extensions setting is on, array and structure variables that are local to a function and have the automatic storage class may be initialized with values that do not need to be constant. Listing 15-16 on page 292 shows an example.

**Listing 15-16. Initializing arrays and structures with non-constant values**

```c
void f(int i)
{
    int j = i * 10; /* Always OK. */
    /* These initializations are only accepted when GCC extensions are on. */
    struct { int x, y; } s = { i + 1, i + 2 };
    int a[2] = { i, i + 2 };
}
```

15.3.3 The sizeof() Operator

When the GCC extensions setting is on, the `sizeof()` operator computes the size of function and void types. In both cases, the `sizeof()` operator evaluates to 1. The ISO/IEC 9899-1990 C Standard ("C90") does not specify the size of the `void` type and functions. Listing 15-17 on page 292 shows an example.

**Listing 15-17. Using the sizeof() operator with void and function types**

```c
int f(int a)
{
    return a * 10;
}

void g(void)
{
    size_t voidsize = sizeof(void); /* voidsize contains 1 */
    size_t funcsize = sizeof(f); /* funcsize contains 1 */
}
```

15.3.4 Statements in Expressions

When the GCC extensions setting is on, expressions in function bodies may contain statements and definitions. To use a statement or declaration in an expression, enclose it within braces. The last item in the brace-enclosed expression gives the expression its value. Listing 15-18 on page 292 shows an example.

**Listing 15-18. Using statements and definitions in expressions**

```c
```
#define POW2(n) ({ int i,r; for(r=1,i=n; i>0; --i) r *= 2; r;})

int main()
{
    return POW2(4);
}

15.3.5 Redefining Macros

When the GCC extensions setting is on, macros may be redefined with the \#define directive without first undefining them with the \#undef directive. Listing 15-19 on page 293 shows an example.

Listing 15-19. Redefining a macro without undefining first

#define SOCK_MAXCOLOR 100
#undef SOCK_MAXCOLOR
#define SOCK_MAXCOLOR 200 /* OK: this macro is previously undefined. */

#define SOCK_MAXCOLOR 300

15.3.6 The typeof() Operator

When the GCC extensions setting is on, the compiler recognizes the typeof() operator. This compile-time operator returns the type of an expression. You may use the value returned by this operator in any statement or expression where the compiler expects you to specify a type. The compiler evaluates this operator at compile time. The \_typeof()\_ operator is the same as this operator. Listing 15-20 on page 293 shows an example.

Listing 15-20. Using the typeof() operator

int *ip;
/* Variables iptr and jptr have the same type. */
typeof(ip) iptr;
int *jptr;

/* Variables i and j have the same type. */
typeof(*ip) i;
int j;
15.3.7 Void and Function Pointer Arithmetic

The ISO/IEC 9899-1990 C Standard does not accept arithmetic expressions that use pointers to `void` or functions. With GCC extensions on, the compiler accepts arithmetic manipulation of pointers to `void` and functions.

15.3.8 The `__builtin_constant_p()` Operator

When the GCC extensions setting is on, the compiler recognizes the `__builtin_constant_p()` operator. This compile-time operator takes a single argument and returns 1 if the argument is a constant expression or 0 if it is not.

15.3.9 Forward Declarations of Static Arrays

When the GCC extensions setting is on, the compiler will not issue an error when you declare a static array without specifying the number of elements in the array if you later declare the array completely. Listing 15-21 on page 294 shows an example.

Listing 15-21. Forward declaration of an empty array

```c
static int a[]; /* Allowed only when GCC extensions are on. */
/* ... */
static int a[10]; /* Complete declaration. */
```

15.3.10 Omitted Operands in Conditional Expressions

When the GCC extensions setting is on, you may skip the second expression in a conditional expression. The default value for this expression is the first expression. Listing 15-22 on page 294 shows an example.

Listing 15-22. Using the shorter form of the conditional expression

```c
void f(int i, int j)
{
    int a = i ? i : j;
    int b = i ?: j; /* Equivalent to int b = i ? i : j; */
    /* Variables a and b are both assigned the same value. */
```
15.3.11 The __builtin_expect() Operator

When the GCC extensions setting is on, the compiler recognizes the __builtin_expect() operator. Use this compile-time operator in an if or while statement to specify to the compiler how to generate instructions for branch prediction.

This compile-time operator takes two arguments:

- the first argument must be an integral expression
- the second argument must be a literal value

The second argument is the most likely result of the first argument. Listing 15-23 on page 295 shows an example.

**Listing 15-23. Example for __builtin_expect() operator**

```c
void search(int *array, int size, int key)
{
    int i;
    for (i = 0; i < size; ++i)
    {
        /* We expect to find the key rarely. */
        if (__builtin_expect(array[i] == key, 0))
        {
            rescue(i);
        }
    }
}
```

15.3.12 Void Return Statements

When the GCC extensions setting is on, the compiler allows you to place expressions of type void in a return statement. Listing 15-24 on page 295 shows an example.

**Listing 15-24. Returning void**

```c
void f(int a)
{
    /* ... */
    return; /* Always OK. */
}

void g(int b)
```
15.3.13 Minimum and Maximum Operators

When the GCC extensions setting is on, the compiler recognizes built-in minimum (<?) and maximum (?>) operators.

Listing 15-25. Example of minimum and maximum operators

```c
int a = 1 <? 2; // 1 is assigned to a.
int b = 1 >? 2; // 2 is assigned to b.
```

15.3.14 Local Labels

When the GCC extensions setting is on, the compiler allows labels limited to a block's scope. A label declared with the __label__ keyword is visible only within the scope of its enclosing block. Listing 15-26 on page 296 shows an example.

Listing 15-26. Example of using local labels

```c
void f(int i)
{
    if (i >= 0)
    {
        __label__ again; /* First again. */
        if (--i > 0)
            goto again; /* Jumps to first again. */
    }
    else
    {
        __label__ again; /* Second again. */
        if (++i < 0)
            goto again; /* Jumps to second again. */
    }
}
```
Chapter 16
C++ Compiler

This chapter explains the CodeWarrior implementation of the C++ programming language:

• C++ Compiler Performance
• Extensions to Standard C++
• Implementation-Defined Behavior
• GCC Extensions

16.1 C++ Compiler Performance

Some options affect the C++ compiler's performance. This section explains how to improve compile times when translating C++ source code:

• Precompiling C++ Source Code
• Using the Instance Manager

16.1.1 Precompiling C++ Source Code

The CodeWarrior C++ compiler has these requirements for precompiling source code:

• C source code may not include precompiled C++ header files and C++ source code may not include precompiled C header files.
• C++ source code can contain inline functions
• C++ source code may contain constant variable declarations
• A C++ source code file that will be automatically precompiled must have a .pch++ file name extension.
16.1.2 Using the Instance Manager

The instance manager reduces compile time by generating a single instance of some kinds of functions:

- template functions
- functions declared with the `inline` qualifier that the compiler was not able to insert in line

The instance manager reduces the size of object code and debug information but does not affect the linker's output file size, though, since the compiler is effectively doing the same task as the linker in this mode.

Table 16-1 shows how to control the C++ instance manager.

**Table 16-1. Controlling the C++ instance manager**

<table>
<thead>
<tr>
<th></th>
<th>To control this option from here...</th>
<th>use this setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>source code</td>
<td>#pragma instmgr_file</td>
<td></td>
</tr>
<tr>
<td>command line</td>
<td>-instmgr</td>
<td></td>
</tr>
</tbody>
</table>

16.2 Extensions to Standard C++

The CodeWarrior C++ compiler has features and capabilities that are not described in the ISO/IEC 14882:2003 C++ standard:

- `__PRETTY_FUNCTION__` Identifier
- Standard and Non-Standard Template Parsing

16.2.1 `__PRETTY_FUNCTION__` Identifier

The `__PRETTY_FUNCTION__` predefined identifier represents the qualified (unmangled) C++ name of the function being compiled.
16.2.2 Standard and Non-Standard Template Parsing

CodeWarrior C++ has options to specify how strictly template declarations and instantiations are translated. When using its strict template parser, the compiler expects the `typename` and `template` keywords to qualify names, preventing the same name in different scopes or overloaded declarations from being inadvertently used. When using its regular template parser, the compiler makes guesses about names in templates, but may guess incorrectly about which name to use.

A qualified name that refers to a type and that depends on a template parameter must begin with `typename` (ISO/IEC 14882:2003 C++, §14.6). Listing 16-1 on page 299 shows an example.

### Listing 16-1. Using the `typename` keyword

```cpp
template <typename T> void f()
{
    T::name *ptr; // ERROR: an attempt to multiply T::name by ptr
    typename T::name *ptr; // OK
}
```

The compiler requires the `template` keyword at the end of "." and "->" operators, and for qualified identifiers that depend on a template parameter. Listing 16-2 on page 299 shows an example.

### Listing 16-2. Using the `template` keyword

```cpp
template <typename T> void f(T* ptr)
{
    ptr->f<int>(); // ERROR: f is less than int
    ptr->template f<int>();// OK
}
```

Names referred to inside a template declaration that are not dependent on the template declaration (that do not rely on template arguments) must be declared before the template's declaration. These names are bound to the template declaration at the point where the template is defined. Bindings are not affected by definitions that are in scope at the point of instantiation. Listing 16-3 on page 299 shows an example.

### Listing 16-3. Binding non-dependent identifiers

```cpp
void f(char);

template <typename T> void tmpl_func()
{
    f(1); // Uses f(char); f(int), below, is not defined yet.
    g(); // ERROR: g() is not defined yet.
}
void g();
void f(int);
```
Names of template arguments that are dependent in base classes must be explicitly qualified (ISO/IEC 14882:2003 C++, §14.6.2). See Listing 16-4 on page 300.

Listing 16-4. Qualifying template arguments in base classes

```cpp
template <typename T> struct Base
{
    void f();
}
template <typename T> struct Derive: Base<T>
{
    void g()
    {
        f(); // ERROR: Base<T>::f() is not visible.
        Base<T>::f(); // OK
    }
}
```

When a template contains a function call in which at least one of the function's arguments is type-dependent, the compiler uses the name of the function in the context of the template definition (ISO/IEC 14882:2003 C++, §14.6.2.2) and the context of its instantiation (ISO/IEC 14882:2003 C++, §14.6.4.2). Listing 16-5 on page 300 shows an example.

Listing 16-5. Function call with type-dependent argument

```cpp
void f(char);
template <typename T> void type_dep_func()
{
    f(1); // Uses f(char), above; f(int) is not declared yet.
    f(T()); // f() called with a type-dependent argument.
}
void f(int);
struct A{};
void f(A);
int main()
{
    type_dep_func<int>(); // Calls f(char) twice.
    type_dep_func<A>(); // Calls f(char) and f(A);
    return 0;
}
```

The compiler only uses external names to look up type-dependent arguments in function calls. See Listing 16-6 on page 300.

Listing 16-6. Function call with type-dependent argument and external names

```cpp
static void f(int); // f() is internal.
template <typename T> void type_dep_fun_ext()
{
    f(T()); // f() called with a type-dependent argument.
}
int main()
{
    type_dep_fun_ext<int>(); // ERROR: f(int) must be external.
}
```

The compiler only uses external names to look up type-dependent arguments in function calls. See Listing 16-6 on page 300.
The compiler does not allow expressions in inline assembly statements that depend on template parameters. See Listing 16-7 on page 301.

**Listing 16-7. Assembly statements cannot depend on template arguments**

```c
template <typename T> void asm_tmpl()
{
    asm { move #sizeof(T), D0 }; // ERROR: Not supported.
}
```

The compiler also supports the address of template-id rules. See Listing 16-8 on page 301.

**Listing 16-8. Address of Template-id Supported**

```c
template <typename T> void funcA(T) {}
template <typename T> void funcB(T) {}
...
funcA( &funcB<int> ); // now accepted
```

### 16.3 Implementation-Defined Behavior

Annex A of the ISO/IEC 14882:2003 C++ Standard lists compiler behaviors that are beyond the scope of the standard, but which must be documented for a compiler implementation. This annex also lists minimum guidelines for these behaviors, although a conforming compiler is not required to meet these minimums.

The CodeWarrior C++ compiler has these implementation quantities listed in Table 16-2, based on the ISO/IEC 14882:2003 C++ Standard, Annex A.

**NOTE**

The term *unlimited* in Table 16-2 means that a behavior is limited only by the processing speed or memory capacity of the computer on which the CodeWarrior C++ compiler is running.


<table>
<thead>
<tr>
<th>Behavior</th>
<th>Standard Minimum Guideline</th>
<th>CodeWarrior Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nesting levels of compound statements, iteration control structures, and selection control structures</td>
<td>256</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Nesting levels of conditional inclusion</td>
<td>256</td>
<td>256</td>
</tr>
<tr>
<td>Pointer, array, and function declarators (in any combination) modifying an arithmetic, structure, union, or incomplete type in a declaration</td>
<td>256</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Nesting levels of parenthesized expressions within a full expression</td>
<td>256</td>
<td>Unlimited</td>
</tr>
</tbody>
</table>

*Table continues on the next page...*
### Table 16-2. Implementation Quantities for C/C++ Compiler (ISO/IEC 14882:2003 C++, §A) (continued)

<table>
<thead>
<tr>
<th>Behavior</th>
<th>Standard Minimum Guideline</th>
<th>CodeWarrior Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of initial characters in an internal identifier or macro name</td>
<td>1024</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Number of initial characters in an external identifier</td>
<td>1024</td>
<td>Unlimited</td>
</tr>
<tr>
<td>External identifiers in one translation unit</td>
<td>65536</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Identifiers with block scope declared in one block</td>
<td>1024</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Macro identifiers simultaneously defined in one translation unit</td>
<td>65536</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Parameters in one function definition</td>
<td>256</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Arguments in one function call</td>
<td>256</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Parameters in one macro definition</td>
<td>256</td>
<td>256</td>
</tr>
<tr>
<td>Arguments in one macro invocation</td>
<td>256</td>
<td>256</td>
</tr>
<tr>
<td>Characters in one logical source line</td>
<td>65536</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Characters in a character string literal or wide string literal (after concatenation)</td>
<td>65536</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Size of an object</td>
<td>262144</td>
<td>2 GB</td>
</tr>
<tr>
<td>Nesting levels for # include files</td>
<td>256</td>
<td>256</td>
</tr>
<tr>
<td>Case labels for a switch statement (excluding those for any nested switch statements)</td>
<td>16384</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Data members in a single class, structure, or union</td>
<td>16384</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Enumeration constants in a single enumeration</td>
<td>4096</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Levels of nested class, structure, or union definitions in a single struct-declaration-list</td>
<td>256</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Functions registered by atexit()</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>Direct and indirect base classes</td>
<td>16384</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Direct base classes for a single class</td>
<td>1024</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Members declared in a single class</td>
<td>4096</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Final overriding virtual functions in a class, accessible or not</td>
<td>16384</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Direct and indirect virtual bases of a class</td>
<td>1024</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Static members of a class</td>
<td>1024</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Friend declarations in a class</td>
<td>4096</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Access control declarations in a class</td>
<td>4096</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Member initializers in a constructor definition</td>
<td>6144</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Scope qualifications of one identifier</td>
<td>256</td>
<td>Unlimited</td>
</tr>
</tbody>
</table>

*Table continues on the next page...*
### Table 16-2. Implementation Quantities for C/C++ Compiler (ISO/IEC 14882:2003 C++, §A) (continued)

<table>
<thead>
<tr>
<th>Behavior</th>
<th>Standard Minimum Guideline</th>
<th>CodeWarrior Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nested external specifications</td>
<td>1024</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Template arguments in a template declaration</td>
<td>1024</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Recursively nested template instantiations</td>
<td>17</td>
<td>64 (adjustable up to 30000 using #pragma template_depth(&lt;n&gt;))</td>
</tr>
<tr>
<td>Handlers per try block</td>
<td>256</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Throw specifications on a single function declaration</td>
<td>256</td>
<td>Unlimited</td>
</tr>
</tbody>
</table>

### 16.4 GCC Extensions

The CodeWarrior C++ compiler recognizes some extensions to the ISO/IEC 14882:2003 C++ standard that are also recognized by the GCC (GNU Compiler Collection) C++ compiler.

The compiler allows the use of the `::` operator, of the form `class::member`, in a class declaration.

**Listing 16-9. Using the `::` operator in class declarations**

```cpp
class MyClass {
    int MyClass::getval();
};
```
Chapter 17
Precompiling

Each time you invoke the CodeWarrior compiler to translate a source code file, it preprocesses the file to prepare its contents for translation. Preprocessing tasks include expanding macros, removing comments, and including header files. If many source code files include the same large or complicated header file, the compiler must preprocess it each time it is included. Repeatedly preprocessing this header file can take up a large portion of the time that the compiler operates.

To shorten the time spent compiling a project, CodeWarrior compilers can precompile a file once instead of preprocessing it every time it is included in project source files. When it precompiles a header file, the compiler converts the file's contents into internal data structures, then writes this internal data to a precompiled file. Conceptually, precompiling records the compiler's state after the preprocessing step and before the translation step of the compilation process.

This section shows you how to use and create precompiled files:

- What Can be Precompiled
- Using a Precompiled File
- Creating a Precompiled File

17.1 What Can be Precompiled

A file to be precompiled does not have to be a header file (.h or .hpp files, for example), but it must meet these requirements:

- The file must be a source code file in text format.
  You cannot precompile libraries or other binary files.
- Precompiled files must have a .mch filename extension.
- The file must not contain any statements that generate data or executable code.
However, the file may define static data.

• Precompiled header files for different IDE build targets are not interchangeable.

17.2 Using a Precompiled File

To use a precompiled file, simply include it in your source code files like you would any other header file:

• A source file may include only one precompiled file.
• A file may not define any functions, variables or types before including a precompiled file.
• Typically, a source code file includes a precompiled file before anything else (except comments).

Listing 17-1 on page 306 shows an example.

Listing 17-1. Using a precompiled file

/* sock_main.c */
#include "sock.mch" /* Precompiled header file. */
#include "wool.h /* Regular header file. */
/* ... */

17.3 Creating a Precompiled File

This section shows how to create and manage precompiled files:

• Precompiling a File in the CodeWarrior IDE
• Precompiling a File on the Command Line
• Updating a Precompiled File Automatically
• Preprocessor Scope in Precompiled Files

17.3.1 Precompiling a File in the CodeWarrior IDE

To precompile a file in the CodeWarrior IDE, follow these steps:

1. Start the CodeWarrior IDE.
2. Open or create a project.
3. Select the source code file to precompile.
4. Right-click to activate the context menu, select **Preprocess**. A precompiled file is created and displayed in the Editor view.

You may now include the new precompiled file in source code files.

### 17.3.2 Precompiling a File on the Command Line

To precompile a file on the command line, follow these steps:

1. Start a command line shell.
2. Issue this command

```
  mwcc  h_file  -precompile  p_file
```

where `mwcc` is the name of the CodeWarrior compiler tool, `h_file` is the name of the header to precompile, and `p_file` is the name of the resulting precompiled file.

### 17.3.3 Updating a Precompiled File Automatically

Use the CodeWarrior IDE's project manager to update a precompiled header automatically. The IDE creates a precompiled file from a source code file during a compile, update, or make operation if the source code file meets these criteria:

- The text file name ends with `.pch`.
- The file is in a project's build target.
- The file uses the `precompile_target` pragma.
- The file, or files it depends on, have been modified.

The IDE uses the build target's settings to preprocess and precompile files.

### 17.3.4 Preprocessor Scope in Precompiled Files
When precompiling a header file, the compiler preprocesses the file too. In other words, a precompiled file is preprocessed in the context of its precompilation, not in the context of its later compilation.

The preprocessor also tracks macros used to guard #include files to reduce parsing time. If a file's contents are surrounded with

```c
#ifndef MYHEADER_H
    #define MYHEADER_H
    /* file contents */
#endif
```

the compiler will not load the file twice, saving some time in the process.

Pragma settings inside a precompiled file affect only the source code within that file. The pragma settings for an item declared in a precompiled file (such as data or a function) are saved then restored when the precompiled header file is included.

For example, the source code in Listing 17-2 on page 308 specifies that the variable `xxx` is a far variable.

### Listing 17-2. Pragma Settings in a Precompiled Header

```c
/* my_pch.pch */

/* Generate a precompiled header named pch.mch. */
#pragma precompile_target "my_pch.mch"

#pragma far_data on
extern int xxx;
```

The source code in Listing 17-3 on page 308 includes the precompiled version of Listing 17-2 on page 308.

### Listing 17-3. Pragma Settings in an Included Precompiled File

```c
/* test.c */

/* Far data is disabled. */
#pragma far_data off

/* This precompiled file sets far_data on. */
#include "my_pch.mch"

/* far_data is still off but xxx is still a far variable. */
```

The pragma setting in the precompiled file is active within the precompiled file, even though the source file including the precompiled file has a different setting.
Chapter 18
Intermediate Optimizations

After it translates a program's source code into its intermediate representation, the compiler optionally applies optimizations that reduce the program's size, improve its execution speed, or both. The topics in this chapter explains these optimizations and how to apply them:

- Interprocedural Analysis
- Intermediate Optimizations
- Inlining

18.1 Interprocedural Analysis

Most compiler optimizations are applied only within a function. The compiler analyzes a function's flow of execution and how the function uses variables. It uses this information to find shortcuts in execution and reduce the number of registers and memory that the function uses. These optimizations are useful and effective but are limited to the scope of a function.

The CodeWarrior compiler has a special optimization that it applies at a greater scope. Widening the scope of an optimization offers the potential to greatly improve performance and reduce memory use. Interprocedural analysis examines the flow of execution and data within entire files and programs to improve performance and reduce size.

- Invoking Interprocedural Analysis
- File-Level Optimization
- Program-Level Optimization
- Program-Level Requirements
18.1.1 Invoking Interprocedural Analysis

Table 18-1 explains how to control interprocedural analysis.

<table>
<thead>
<tr>
<th>Turn control this option from here...</th>
<th>use this setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeWarrior IDE</td>
<td>Choose an item from the IPA option drop down list box on the Properties &gt; C/C++ Build &gt; Settings &gt; Tool Settings &gt; PowerPC Compiler &gt; Optimization panel.</td>
</tr>
<tr>
<td>source code</td>
<td>#pragma ipaprogram file</td>
</tr>
<tr>
<td>command line</td>
<td>-ipa file</td>
</tr>
</tbody>
</table>

18.1.2 Function-Level Optimization

Interprocedural analysis may be disabled by setting it to either off or function. If IPA is disabled, the compiler generates instructions and data as it reads and analyzes each function. This setting is equivalent to the "no deferred codegen" mode of older compilers.

18.1.3 File-Level Optimization

When interprocedural analysis is set to optimize at the file level, the compiler reads and analyzes an entire file before generating instructions and data.

At this level, the compiler generates more efficient code for inline function calls and C++ exception handling than when interprocedural analysis is off. The compiler is also able to increase character string reuse and pooling, reducing the size of object code. This is equivalent to the " deferred inlining" and "deferred codegen" options of older compilers.

The compiler also safely removes static functions and variables that are not referred to within the file, which reduces the amount of object code that the linker must process, resulting in better linker performance.

18.1.4 Program-Level Optimization
When interprocedural analysis is set to optimize at the program level, the compiler reads and analyzes all files in a program before generating instructions and data.

At this level of interprocedural analysis, the compiler generates the most efficient instructions and data for inline function calls and C++ exception handling compared to other levels. The compiler is also able to increase character string reuse and pooling, reducing the size of object code.

### 18.1.5 Program-Level Requirements

Program-level interprocedural analysis imposes some requirements and limitations on the source code files that the compiler translates:

- Dependencies Among Source Files
- Function and Top-level Variable Declarations
- Type Definitions
- Unnamed Structures and Enumerations in C

#### 18.1.5.1 Dependencies Among Source Files

A change to even a single source file in a program still requires that the compiler read and analyze all files in the program, even those files that are not dependent on the changed file. This requirement significantly increases compile time.

#### 18.1.5.2 Function and Top-level Variable Declarations

Because the compiler treats all files that compose a program as if they were a single, large source file, you must make sure all non-static declarations for variables or functions with the same name are identical. See Listing 18-1 on page 311 for an example of declarations that prevent the compiler from applying program-level analysis. Listing 18-2 on page 312 fixes this problem by renaming the conflicting symbols.

Listing 18-1. Declaration conflicts in program-level interprocedural analysis

```c
/* file1.c */
extern int i;
extern int f();
int main(void)
{
```
Listing 18-2. Fixing declaration conflicts for program-level interprocedural analysis

```c
/* file1.c */
extern int i1;
extern int f1();
int main(void)
{
    return i1 + f1();
}
/* file2.c */
short i2;
extern void f2();
```

18.1.5.3 Type Definitions

Because the compiler examines all source files for a program, make sure all definitions for a type are the same. See Listing 18-3 on page 312 for an example of conflicting type definitions. Listing 18-4 on page 312 and Listing 18-5 on page 312 show suggested solutions.

Listing 18-3. Type definitions conflicts in program-level interprocedural analysis

```c
/* fileA.c */
struct a_rec { int i, j; };
a_rec a;
/* fileB.c */
struct a_rec { char c; }; /* Conflict with a_rec in fileA.c */
a_rec b;
```

Listing 18-4. Fixing type definitions conflicts in C

```c
/* fileA.c */
struct a1_rec { int i, j; };
a1_rec a;
/* fileB.c */
struct a2_rec { char c; };
a2_rec b;
```

Listing 18-5. Fixing type definitions conflicts in C++

```c
/* fileA.c */
namespace { struct a_rec { int i, j; } };
a_rec a;
/* fileB.c */
namespace { struct a_rec { char c; } };
a_rec b;
```
18.1.5.4 Unnamed Structures and Enumerations in C

The C language allows anonymous `struct` and `enum` definitions in type definitions. Using such definitions prevents the compiler from properly applying program-level interprocedural analysis. Make sure to give names to structures and enumerations in type definitions. Listing 18-6 on page 313 shows an example of unnamed structures and enumerations and Listing 18-7 on page 313 shows a suggested solution.

```
Listing 18-6. Unnamed structures and enumerations in C

/* In C, the types x_rec and y_enum each represent a structure and an enumeration with no name.

   In C++ these same statements define a type x_rec and y_enum, a structure named x_rec and an enumeration named y_enum.
*/
typedef struct { int a, b, c; } x_rec;
typedef enum { Y_FIRST, Y_SECOND, Y_THIRD } y_enum;
```

```
Listing 18-7. Naming structures and enumerations in C

typedef struct x_rec { int a, b, c; } x_rec;
typedef enum y_enum { Y_FIRST, Y_SECOND, Y_THIRD } y_enum;
```

18.2 Intermediate Optimizations

After it translates a function into its intermediate representation, the compiler may optionally apply some optimizations. The result of these optimizations on the intermediate representation will either reduce the size of the executable code, improve the executable code's execution speed, or both.

- Dead Code Elimination
- Expression Simplification
- Common Subexpression Elimination
- Copy Propagation
- Dead Store Elimination
- Live Range Splitting
- Loop-Invariant Code Motion
- Strength Reduction
- Loop Unrolling
18.2.1 Dead Code Elimination

The dead code elimination optimization removes expressions that are not accessible or are not referred to. This optimization reduces size and increases execution speed. Table 18-2 explains how to control the optimization for dead code elimination.

<table>
<thead>
<tr>
<th>Turn control this option from here...</th>
<th>use this setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeWarrior IDE</td>
<td>Choose an Optimization Level value (1, 2, 3, 4) from the Properties &gt; C/C++ Build &gt; Settings &gt; Tool Settings &gt; PowerPC Compiler &gt; Optimization panel.</td>
</tr>
<tr>
<td>source code</td>
<td>#pragma opt_dead_code on</td>
</tr>
<tr>
<td>command line</td>
<td>-opt [no]deadcode</td>
</tr>
</tbody>
</table>

In Figure 18-1, the call to `func1()` will never execute because the `if` statement that it is associated with will never be true. Consequently, the compiler can safely eliminate the call to `func1()`, as shown in Figure 18-2.

**Figure 18-1. Before dead code elimination**

```c
void func_from(void)
{
    if (0)
    {
        func1();
    }
    func2();
}
```

**Figure 18-2. After dead code elimination**

```c
void func_to(void)
{
    func2();
}
```

18.2.2 Expression Simplification

The expression simplification optimization attempts to replace arithmetic expressions with simpler expressions. Additionally, the compiler also looks for operations in expressions that can be avoided completely without affecting the final outcome of the expression. This optimization reduces size and increases speed.
Table 18-3 explains how to control the optimization for expression simplification.

### Table 18-3. Controlling expression simplification

<table>
<thead>
<tr>
<th>Turn control this option from here...</th>
<th>use this setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeWarrior IDE</td>
<td>Choose an <strong>Optimization Level</strong> value (1, 2, 3, 4) from the Properties &gt; C/C++ Build &gt; Settings &gt; Tool Settings &gt; PowerPC Compiler &gt; Optimization panel.</td>
</tr>
<tr>
<td>source code</td>
<td>There is no pragma to control this optimization.</td>
</tr>
<tr>
<td>command line</td>
<td><code>-opt level=1, -opt level=2, -opt level=3, -opt level=4</code></td>
</tr>
</tbody>
</table>

For example, **Figure 18-3** contains a few assignments to some arithmetic expressions:

- addition to zero
- multiplication by a power of 2
- subtraction of a value from itself
- arithmetic expression with two or more literal values

**Figure 18-3. Before expression simplification**

```c
void func_from(int* result1, int* result2, int* result3, int* result4, int x) {
    *result1 = x + 0;
    *result2 = x * 2;
    *result3 = x - x;
    *result4 = 1 + x + 4;
}
```

**Figure 18-4** shows source code that is equivalent to expression simplification. The compiler has modified these assignments to:

- remove the addition to zero
- replace the multiplication of a power of 2 with bit-shift operation
- replace a subtraction of x from itself with 0
- consolidate the additions of 1 and 4 into 5

**Figure 18-4. After expression simplification**

```c
void func_to(int* result1, int* result2, int* result3, int* result4, int x) {
    *result1 = x;
    *result2 = x << 1;
    *result3 = 0;
    *result4 = 5 + x;
}
```

### 18.2.3 Common Subexpression Elimination
Common subexpression elimination replaces multiple instances of the same expression with a single instance. This optimization reduces size and increases execution speed.

Table 18-4 explains how to control the optimization for common subexpression elimination.

<table>
<thead>
<tr>
<th>Turn control this option from here...</th>
<th>use this setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeWarrior IDE</td>
<td>Choose an <strong>Optimization Level</strong> value (2, 3, 4) from the Properties &gt; C/C++ Build &gt; Settings &gt; Tool Settings &gt; PowerPC Compiler &gt; Optimization panel.</td>
</tr>
<tr>
<td>source code</td>
<td>#pragma opt_common_subs on</td>
</tr>
<tr>
<td>command line</td>
<td>-opt [no]cse</td>
</tr>
</tbody>
</table>

For example, in Figure 18-5, the subexpression $x \times y$ occurs twice.

**Figure 18-5. Before common subexpression elimination**

```c
void func_from(int* vec, int size, int x, int y, int value)
{
    if (x * y < size)
    {
        vec[x * y - 1] = value;
    }
}
```

**Figure 18-6. After common subexpression elimination**

```c
void func_to(int* vec, int size, int x, int y, int value)
{
    int temp = x * y;
    if (temp < size)
    {
        vec[temp - 1] = value;
    }
}
```

18.2.4 Copy Propagation

Copy propagation replaces variables with their original values if the variables do not change. This optimization reduces runtime stack size and improves execution speed.
Table 18-5 explains how to control the optimization for copy propagation.

**Table 18-5. Controlling copy propagation**

<table>
<thead>
<tr>
<th>Turn control this option from here...</th>
<th>use this setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeWarrior IDE</td>
<td>Choose an <strong>Optimization Level</strong> value (2, 3, 4) from the Properties &gt; C/C++ Build &gt; Settings &gt; Tool Settings &gt; PowerPC Compiler &gt; Optimization panel.</td>
</tr>
<tr>
<td>source code</td>
<td>#pragma opt_propagation on</td>
</tr>
<tr>
<td>command line</td>
<td>-opt [no]prop[agation]</td>
</tr>
</tbody>
</table>

For example, in Figure 18-7, the variable \( j \) is assigned the value of \( x \). But \( j \)'s value is never changed, so the compiler replaces later instances of \( j \) with \( x \), as shown in Figure 18-8.

By propagating \( x \), the compiler is able to reduce the number of registers it uses to hold variable values, allowing more variables to be stored in registers instead of slower memory. Also, this optimization reduces the amount of stack memory used during function calls.

**Figure 18-7. Before copy propagation**

```c
void func_from(int* a, int x)
{
    int i;
    int j;
    j = x;
    for (i = 0; i < j; i++)
    {
        a[i] = j;
    }
}
```

**Figure 18-8. After copy propagation**

```c
void func_to(int* a, int x)
{
    int i;
    int j;
    j = x;
    for (i = 0; i < x; i++)
    {
        a[i] = x;
    }
}
```

### 18.2.5 Dead Store Elimination

Dead store elimination removes unused assignment statements. This optimization reduces size and improves speed.
Table 18-6 explains how to control the optimization for dead store elimination.

### Table 18-6. Controlling dead store elimination

<table>
<thead>
<tr>
<th>Turn control this option from here...</th>
<th>use this setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeWarrior IDE</td>
<td>Choose an Optimization Level value ((3, 4)) from the Properties &gt; C/C++ Build &gt; Settings &gt; Tool Settings &gt; PowerPC Compiler &gt; Optimization panel.</td>
</tr>
<tr>
<td>source code</td>
<td>#pragma opt_dead_assignments on</td>
</tr>
<tr>
<td>command line</td>
<td>-opt [no]deadstore</td>
</tr>
</tbody>
</table>

For example, in Figure 18-9 the variable \(x\) is first assigned the value of \(y \times y\). However, this result is not used before \(x\) is assigned the result returned by a call to `getresult()`.

In Figure 18-10 the compiler can safely remove the first assignment to \(x\) since the result of this assignment is never used.

**Figure 18-9. Before dead store elimination**

```c
void func_from(int x, int y)
{
    x = y * y;
    otherfunc1(y);
    x = getresult();
    otherfunc2(y);
}
```

**Figure 18-10. After dead store elimination**

```c
void func_to(int x, int y)
{
    otherfunc1(y);
    x = getresult();
    otherfunc2(y);
}
```

### 18.2.6 Live Range Splitting

Live range splitting attempts to reduce the number of variables used in a function. This optimization reduces a function's runtime stack size, requiring fewer instructions to invoke the function. This optimization potentially improves execution speed.
Table 18-7 explains how to control the optimization for live range splitting.

### Table 18-7. Controlling live range splitting

<table>
<thead>
<tr>
<th>Turn control this option from here...</th>
<th>use this setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeWarrior IDE</td>
<td>Choose an <strong>Optimization Level</strong> value (3, 4) from the <strong>Properties &gt; C/C++ Build &gt; Settings &gt; Tool Settings &gt; PowerPC Compiler &gt; Optimization</strong> panel.</td>
</tr>
<tr>
<td>source code</td>
<td>There is no pragma to control this optimization.</td>
</tr>
<tr>
<td>command line</td>
<td><code>-opt level=3 , -opt level=4</code></td>
</tr>
</tbody>
</table>

For example, in **Figure 18-11** three variables, `a`, `b`, and `c`, are defined. Although each variable is eventually used, each of their uses is exclusive to the others. In other words, `a` is not referred to in the same expressions as `b` or `c`, `b` is not referred to with `a` or `c`, and `c` is not used with `a` or `b`.

In **Figure 18-12**, the compiler has replaced `a`, `b`, and `c`, with a single variable. This optimization reduces the number of registers that the object code uses to store variables, allowing more variables to be stored in registers instead of slower memory. This optimization also reduces a function's stack memory.

**Figure 18-11. Before live range splitting**

```c
void func_from(int x, int y)
{
    int a;
    int b;
    int c;
    a = x * y;
    otherfunc(a);
    b = x + y;
    otherfunc(b);
    c = x - y;
    otherfunc(c);
}
```

**Figure 18-12. After live range splitting**

```c
void func_to(int x, int y)
{
    int a_b_or_c;
    a_b_or_c = x * y;
    otherfunc(temp);
    a_b_or_c = x + y;
    otherfunc(temp);
    a_b_or_c = x - y;
    otherfunc(temp);
}
```
18.2.7 Loop-Invariant Code Motion

Loop-invariant code motion moves expressions out of a loop if the expressions are not affected by the loop or the loop does not affect the expression. This optimization improves execution speed.

Table 18-8 explains how to control the optimization for loop-invariant code motion.

<table>
<thead>
<tr>
<th>Turn control this option from here...</th>
<th>use this setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeWarrior IDE</td>
<td>Choose an <strong>Optimization Level</strong> value (3, 4) from the Properties &gt; C/C++ Build &gt; Settings &gt; Tool Settings &gt; PowerPC Compiler &gt; Optimization panel.</td>
</tr>
<tr>
<td>source code</td>
<td>#pragma opt_loop_invariants on</td>
</tr>
<tr>
<td>command line</td>
<td>-opt [no]loop[invariants]</td>
</tr>
</tbody>
</table>

For example, in Figure 18-13, the assignment to the variable \( \text{circ} \) does not refer to the counter variable of the \textbf{for} loop, \( i \). But the assignment to \( \text{circ} \) will be executed at each loop iteration.

Figure 18-14 shows source code that is equivalent to how the compiler would rearrange instructions after applying this optimization. The compiler has moved the assignment to \( \text{circ} \) outside the \textbf{for} loop so that it is only executed once instead of each time the \textbf{for} loop iterates.

**Figure 18-13. Before loop-invariant code motion**

```c
void func_from(float* vec, int max, float val)
{
    float circ;
    int i;
    for (i = 0; i < max; ++i)
    {
        circ = val * 2 * PI;
        vec[i] = circ;
    }
}
```

**Figure 18-14. After loop-invariant code motion**

```c
void func_to(float* vec, int max, float val)
{
    float circ;
    int i;
    circ = val * 2 * PI;
    for (i = 0; i < max; ++i)
    {
        vec[i] = circ;
    }
}
```
18.2.8 Strength Reduction

Strength reduction attempts to replace slower multiplication operations with faster addition operations. This optimization improves execution speed but increases code size. Table 18-9 explains how to control the optimization for strength reduction.

**Table 18-9. Controlling strength reduction**

<table>
<thead>
<tr>
<th>Turn control this option from here...</th>
<th>use this setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeWarrior IDE</td>
<td>Choose an <strong>Optimization Level</strong> value (3,4) from the <code>Properties &gt; C/C++ Build &gt; Settings &gt; Tool Settings &gt; PowerPC Compiler &gt; Optimization</code> panel.</td>
</tr>
<tr>
<td>source code</td>
<td><code>#pragma opt_strength_reduction</code> on</td>
</tr>
<tr>
<td>command line</td>
<td><code>-opt [no]strength</code></td>
</tr>
</tbody>
</table>

For example, in **Figure 18-15**, the assignment to elements of the `vec` array use a multiplication operation that refers to the `for` loop's counter variable, `i`.

In **Figure 18-16**, the compiler has replaced the multiplication operation with a hidden variable that is increased by an equivalent addition operation. Processors execute addition operations faster than multiplication operations.

**Figure 18-15. Before strength reduction**

```c
void func_from(int* vec, int max, int fac)
{
    int i;
    for (i = 0; i < max; ++i)
    {
        vec[i] = fac * i;
    }
}
```

**Figure 18-16. After strength reduction**

```c
void func_to(int* vec, int max, int fac)
{
    int i;
    int hidden_strength_red;
    hidden_strength_red = 0;
    for (i = 0; i < max; ++i)
    {
        vec[i] = hidden_strength_red;
        hidden_strength_red = hidden_strength_red + fac;
    }
}
```
18.2.9 Loop Unrolling

Loop unrolling inserts extra copies of a loop's body in a loop to reduce processor time executing a loop's overhead instructions for each iteration of the loop body. In other words, this optimization attempts to reduce the ratio of time that the processor executes a loop's completion test and branching instructions compared to the time the processor executes the loop's body. This optimization improves execution speed but increases code size.

Table 18-10 explains how to control the optimization for loop unrolling.

**Table 18-10. Controlling loop unrolling**

<table>
<thead>
<tr>
<th>Turn control this option from here...</th>
<th>use this setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeWarrior IDE</td>
<td>Choose an Optimization Level value (3,4) from the Properties &gt; C/C++ Build &gt; Settings &gt; Tool Settings &gt; PowerPC Compiler &gt; Optimization panel.</td>
</tr>
<tr>
<td>source code</td>
<td>#pragma opt_unroll_loops on</td>
</tr>
<tr>
<td>command line</td>
<td>-opt level=3,-opt level=4</td>
</tr>
</tbody>
</table>

For example, in Figure 18-17, the for loop's body is a single call to a function, otherfunc(). For each time the loop's completion test executes

for (i = 0; i < MAX; ++i)

the function executes the loop body only once.

In Figure 18-18, the compiler has inserted another copy of the loop body and rearranged the loop to ensure that variable i is incremented properly. With this arrangement, the loop's completion test executes once for every 2 times that the loop body executes.

**Figure 18-17. Before loop unrolling**

```c
const int MAX = 100;
void func_from(int* vec)
{
    int i;
    for (i = 0; i < MAX; ++i)
    {
        otherfunc(vec[i]);
    }
}
```

**Figure 18-18. After loop unrolling**

```c
const int MAX = 100;
void func_to(int* vec)
{
```
int i;
for (i = 0; i < MAX;)
{
    otherfunc(vec[i]);
    ++i;
    otherfunc(vec[i]);
    ++i;
}

18.3 Inlining

Inlining replaces instructions that call a function and return from it with the actual instructions of the function being called. Inlining functions makes your program faster because it executes the function code immediately without the overhead of a function call and return. However, inlining can also make your program larger because the compiler may insert the function's instructions many times throughout your program.

The rest of this section explains how to specify which functions to inline and how the compiler performs the inlining:

- Choosing Which Functions to Inline
- Inlining Techniques

18.3.1 Choosing Which Functions to Inline

The compiler offers several methods to specify which functions are eligible for inlining.

To specify that a function is eligible to be inlined, precede its definition with the _inline_, __inline__, or __inline keyword. To allow these keywords in C source code, turn off ANSI Keywords Only in the CodeWarrior IDE’s Properties > C/C++ Build > Settings > Tool Settings > PowerPC Compiler > C/C++ Language panel or turn off the only_std_keywords pragma in your source code.

To verify that an eligible function has been inlined or not, use the Non-Inlined Functions option in the IDE's Warnings panel or the warn_notinlined pragma. Figure 18-19 shows an example.

Figure 18-19. Specifying to the compiler that a function may be inlined

#pragma only_std_keywords off
inline int attempt_to_inline(void)
{
    return 10;
}
To specify that a function must never be inlined, follow its definition's specifier with __attribute__((never_inline)). Figure 18-20 shows an example.

Figure 18-20. Specifying to the compiler that a function must never be inlined

```c
int never_inline(void) __attribute__((never_inline))
{
    return 20;
}
```

To specify that no functions in a file may be inlined, including those that are defined with the inline, __inline__, or __inline keywords, use the dont_inline pragma. Figure 18-21 shows an example.

Figure 18-21. Specifying that no functions may be inlined

```c
#pragma dont_inline on

/* Will not be inlined. */
inline int attempt_to_inline(void)
{
    return 10;
}

/* Will not be inlined. */
int never_inline(void) __attribute__((never_inline))
{
    return 20;
}

#pragma dont_inline off

/* Will be inlined, if possible. */
inline int also_attempt_to_inline(void)
{
    return 10;
}
```

Some kinds of functions are never inlined:

- functions with variable argument lists
- functions defined with __attribute__((never_inline))
- functions compiled with #pragma optimize_for_size on or the Optimize For Size setting in the IDE's Global Optimizations panel
- functions which have their addresses stored in variables

**NOTE**
The compiler will not inline these functions, even if they are defined with the inline, __inline__, or __inline keywords.

- functions that return class objects that need destruction
- functions with class arguments that need destruction
The compiler will inline functions that need destruction, without any dependency on the ISO C++ templates, if the class has a trivial empty constructor. Figure 18-22 shows an example.

**Figure 18-22. Inlining function with an empty destructor**

```c
struct X {
    int n;
    X(int a) { n = a; }
    ~X() {}
};

inline X f(X x) { return X(x.n + 1); }

int main()
{
    return f(X(1)).n;
}
```

18.3.2 Inlining Techniques

The depth of inlining explains how many levels of function calls the compiler will inline. The Inline Depth setting in the IDE’s Properties > C/C++ Build > Settings > Tool Settings > PowerPC Compiler > C/C++ Language panel and the inline_depth pragma control inlining depth.

Normally, the compiler only inlines an eligible function if it has already translated the function's definition. In other words, if an eligible function has not yet been compiled, the compiler has no object code to insert. To overcome this limitation, the compiler can perform interprocedural analysis (IPA) either in file or program mode. This lets the compiler evaluate all the functions in a file or even the entire program before inlining the code. The IPA setting in the IDE's C/C++ Language panel and the ipa pragma control this capability.

The compiler normally inlines functions from the first function in a chain of function calls to the last function called. Alternately, the compiler may inline functions from the last function called to the first function in a chain of function calls. The Bottom-up Inlining option in the IDE's C/C++ Language panel and the inline_bottom_up and inline_bottom_up_once pragmas control this reverse method of inlining.
Some functions that have not been defined with the `inline`, `__inline__`, or `__inline` keywords may still be good candidates to be inlined. Automatic inlining allows the compiler to inline these functions in addition to the functions that you explicitly specify as eligible for inlining. The Auto-Inline option in the IDE's C/C++ Language panel and the `auto_inline` pragma control this capability.

When inlining, the compiler calculates the complexity of a function by counting the number of statements, operands, and operations in a function to determine whether or not to inline an eligible function. The compiler does not inline functions that exceed a maximum complexity. The compiler uses three settings to control the extent of inlined functions:

- maximum auto-inlining complexity: the threshold for which a function may be auto-inlined
- maximum complexity: the threshold for which any eligible function may be inlined
- maximum total complexity: the threshold for all inlining in a function

The `inline_max_auto_size`, `inline_max_size`, and `inline_max_total_size` pragmas control these thresholds, respectively.
Chapter 19
Power Architecture Optimizations

This chapter describes optimizations specific to Power Architecture platforms that the CodeWarrior compiler applies to your object code.

- Code Merging

19.1 Code Merging

Code merging reduces the size of object code by removing identical functions. Two or more functions are identical when their executable code is identical.

The CodeWarrior build tools can only apply this optimization to object files generated by the CodeWarrior compilers. The CodeWarrior build tools can only apply this optimization to object code translated from C and C++ source code; the tools cannot apply this optimization to object code generated from assembly files.

Tip
For example, the C++ compiler often generates several copies of the same function when it instantiates template functions. These functions have different names, and these names are considered weak. Under normal circumstances, the linker will issue an error message if it encounters duplicate names. But the linker ignores duplicate names that are marked as weak.
The code merging optimization removes all but one of a group of identical functions. **Table 19-1** shows how to invoke this optimization for all functions. **Table 19-2** shows how to invoke this optimization for weak functions.

**Table 19-1. Controlling code merging for all identical functions**

<table>
<thead>
<tr>
<th>Control this option from here...</th>
<th>use this setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeWarrior IDE</td>
<td>Choose <strong>All Functions</strong> from the <strong>Code Merging</strong> option of the <strong>PowerPC Linker &gt; General</strong> panel.</td>
</tr>
<tr>
<td>command line</td>
<td>-code_merging all</td>
</tr>
</tbody>
</table>

**Table 19-2. Controlling code merging for weak functions only**

<table>
<thead>
<tr>
<th>Control this option from here...</th>
<th>use this setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeWarrior IDE</td>
<td>Choose <strong>Safe Functions</strong> from the <strong>Code Merging</strong> option of the <strong>PowerPC Linker &gt; General</strong> panel.</td>
</tr>
<tr>
<td>command line</td>
<td>-code_merging safe</td>
</tr>
</tbody>
</table>

The code merging optimization will not remove an identical copy of a function if your program refers to its address. In this case, the compiler keeps this copied function but replaces its executable code with a branch instruction to the original function.

To specify that the compiler and linker must not apply code merging to a function, use this directive in your source code:

```c
__declspec(do_not_merge)
fname;
```

where `fname` is the name of a function.

To specify to the linker how to apply code merging to functions, object files, or sections, use these directives in linker command file:

```c
DO_NOT_MERGE
DO_NOT_MERGE_FILES
DO_NOT_MERGE_SECTIONS
AGGRESSIVE_MERGE
AGGRESSIVE_MERGE_FILES
AGGRESSIVE_MERGE_SECTIONS
```
Chapter 20
Inline-Assembly for Power Architecture Build Tools

This chapter explains how to use the inline assembler built into the CodeWarrior™ C and C++ compilers for Power Architecture processors. The compiler's inline assembler allows you to embed assembly language statements in C and C++ functions.

The chapter does not describe the standalone CodeWarrior assembler. For information about this tool, refer to the chapter titled Assembler.

This chapter does not document all the instructions in the Power Architecture instruction set. For complete documentation of this instruction set, see Programming Environments Manual for 32-Bit Implementations of the PowerPC™ Architecture, published by Freescale.

The sections in this chapter are:

- Assembly Syntax
- Referring to Assembly, C, and C++ Variables
- Assembler Directives
- Intrinsic Functions

20.1 Assembly Syntax

The compiler's inline assembler allows a variety of ways to insert assembly language statements in your C or C++ source code:

- Specifying Inline Assembly Statements
- Function-Level Inline Assembly
- Statement-Level Inline Assembly
- GCC-Style Inline Assembly
- Branch Prediction
- PC-Relative Addressing
20.1.1 Specifying Inline Assembly Statements

To specify that a block of C or C++ source code should be interpreted as assembly language, use the `asm` keyword.

**NOTE**
To ensure that the C/C++ compiler recognizes the `asm` keyword, you must clear the ANSI Keywords Only checkbox in the C/C++ Language panel.

As an alternative, the compiler also recognizes the keyword `__asm` even if the ANSI Keywords Only checkbox is checked.

There are a few ways to use assembly language with the CodeWarrior compilers.

- Function-level assembly language: an entire function is in assembly language.
- Statement-level assembly language: mix assembly language with regular C or C++ statements.
- Intrinsic functions: the compiler makes some assembly instructions available as functions that your program calls as regular C or C++ functions.

Keep these tips in mind as you write inline assembly statements:

- All statements must follow this syntax:

  ```
  [label :] (instruction | directive) [operands]
  ```

- Each inline assembly statement must end with a newline or a semicolon (`;`).
- Hexadecimal constants must be in C-style.

  For example: `li r3, 0xABCDEF`

- Assembler directives, instructions, and registers are case-sensitive and must be in lowercase.

20.1.2 Function-Level Inline Assembly
The compiler accepts function definitions that are composed entirely of assembly statements. Function-level assembly code uses this syntax:

```asm
function-definition
```

A function that uses function-level assembly must end with a `blr` instruction.

**Listing 20-1. Example Assembly Language Function**

```asm
asm void mystrcpy(char *tostr, char *fromstr)
{
    addi  tostr, tostr, -1
    addi  fromstr, fromstr, -1
    @1    lbzu  r5, 1(fromstr)
    cmpwi r5, 0
    stbu  r5, 1(tostr)
    bne   @1
    blr
}
```

### 20.1.3 Statement-Level Inline Assembly

The compiler accepts functions that mix regular C/C++ statements with inline assembly. Statement-level assembly language acts as a block of assembly language that may appear anywhere that the compiler allows a regular C or C++ statement. It has this syntax:

```asm
{ one or more instructions }
```

**Listing 20-2. Example of statement-level inline assembly**

```asm
void g(void)
{
    asm { add r2, r3, r4 ; }
}
```

**NOTE**

If you check the Inlined Assembler is Volatile checkbox in the **EPPC Processor** panel, functions that contain an `asm` block are only partially optimized. The optimizer optimizes the function, but skips any `asm` blocks of code. If the Inlined Assembler is Volatile checkbox is clear, the compiler also optimizes `asm` statements.

### 20.1.4 GCC - Style Inline Assembly
The CodeWarrior compiler accepts GCC (Gnu Compiler Collection) syntax for inline assembly statements:

```c
asm("assembly-statements")
```

where assembly-statements represents inline assembly statements that follow the syntax recognized by the GCC C/C++ compiler.

**Figure 20-1. Example of GCC-style inline assembly**

```c
void g(void)
{
    asm("add r2,r3,r4\n\t");
}
```

## 20.1.5 Branch Prediction

To set the branch prediction \( y \) bit for those branch instructions that can use it, use plus (+) or minus (-). For example:

```c
@1 bne+ @2
@2 bne- @1
```

## 20.1.6 PC-Relative Addressing

The compiler does not accept references to addresses that are relative to the program counter. For example, the following is not supported:

```c
asm(b *+8);
```

Instead, use one of the following:

1. Use labels to specify an address in executable code.

   **Listing 20-3. Using a label instead if PC-relative addressing**

   ```c
   asm(b next);
   asm(next:);
   /* OR */
   asm{
       b next1
       next1:
   }
   ```
2. Use relative branch in the function-level assembly instead of statement level.

Listing 20-4. Using relative branching in the function-level assembly

```assembly
asm void functionLevel();
asm void functionLevel()
  b *+8
  nop
  blr
```

### 20.1.7 Normal, Record, and Overflow Forms

Most integer instructions have four forms:

- **normal form** - `add r3,r4,r5`
- **record form** - `add. r3,r4,r5`
  
  This form ends in a period. This form sets register `cr0` to whether the result is less, than, equal to, or greater than zero.
- **overflow** - `addo r3,r4,r5`
  
  This form ends in the letter `o`. This form sets the `so` and `ov` bits in the `xer` if the result overflows.
- **overflow and record** - `addo. r3,r4,r5`
  
  This form ends in `o.`. This form sets both registers.

Some instructions only have a record form (with a period). Always make sure to include the period. For example:

```assembly
andi. r3,r4,7
andis. r3,r4,7
stwcx. r3,r4,r5
```

### 20.1.8 Creating Statement Labels

The name of an inline assembly language statement label must follow these rules:

- A label name cannot be the same as the identifier of any local variables of the function in which the label name appears.
• A label name does not have to start in the first column of the function in which it appears; a label name can be preceded by white space.
• A label name can begin with an "at-sign" character (@) unless the label immediately follows a local variable declaration.
• A label name must end with a colon character (:) unless it begins with an at-sign character (@).

For example, red: and @red are valid, but red is not valid.

• A label name can be the same as an assembly language statement mnemonic.

For example, this statement is valid:

    add: add r3, r4, r5

Examples:

    asm void func1()
    {
        int i;
        @x: li r0,1 //Invalid !!!
    }

    asm void func2()
    {
        int i;
        x: li r0,1 //OK
        @y: add r3, r4, r5 //OK
    }

This is an example of a complete inline assembly language function:

    asm void red(void){ x1: add r3,r4,r5 @x2: add r6,r7,r8 }

20.1.9 Using Comments

You cannot begin comments with a pound sign (#) because the preprocessor uses the pound sign. For example, this format is invalid:

    add r3,r4,r5 #
    Comment

Use C and C++ comments in this format:

    add r3,r4,r5 //
    Comment
    add r3,r4,r5 /*
20.1.10 Using the Preprocessor

You can use all preprocessor features, such as comments and macros, in the assembler. In multi-line macros, you must end each assembly statement with a semicolon ( ; ) because the ( \ ) operator removes newlines. For example:

```assembly
#define remainder(x,y,z) \
divw z,x,y; \
mullw z,z,y; \
subf z,z,x

asm void newPointlessMath(void) 
{ 
    remainder(r3,r4,r5) 
    blr 
}
```

20.2 Referring to Assembly, C, and C++ Variables

The compiler's inline assembler has access to the local and global variables declared in C or C++ source code. The assembler also allows access to members of array, struct, and class objects:

- Using Local Variables and Arguments
- Creating a Stack Frame
- Referring to Variables in Instruction Operands
- Referring to Variables in Memory
- Referring to Registers
- Referring to Labels
- Using Variable Names as Memory Locations
- Using Immediate Operands

20.2.1 Using Local Variables and Arguments
To refer to a memory location, you can use the name of a local variable or argument. The rule for assigning arguments to registers or memory depends on whether the function has a stack frame.

If function has a stack frame, the inline assembler assigns:

- scalar arguments declared as `register` to general purpose registers `r14` to `r31`
- floating-point arguments declared as `register` to floating point `fp14` to `fp31`
- other arguments to memory locations

If a function has no stack frame, the inline assembler assigns arguments that are declared `register` and kept in registers. If you have variable or non-register arguments, the compiler will warn you that you should use `frfree`

**NOTE**

Some op-codes require registers, and others require objects. For example, if you use `nofralloc` with function arguments, you may run into difficulties.

### 20.2.2 Creating a Stack Frame

You need to create a stack frame for a function if the function:

- calls other functions.
- declares non-register arguments or local variables.

To create a stack frame, use the `fralloc` directive at the beginning of your function and the `frfree` directive just before the `blr` statement. The directive `fralloc` automatically allocates (while `frfree` automatically de-allocates) memory for local variables, and saves and restores the register contents.

**Listing 20-5. Example of creating a stack frame**

```assembly
asm void red ()
{
  fralloc
  // Your code here
  frfree
  blr
}
```

The `fralloc` directive has an optional argument, number, that lets you specify the size, in bytes, of the parameter area of the stack frame. The stack frame is an area for storing parameters used by the assembly code. The compiler creates a 0-byte parameter area for you to pass variables into your assembly language functions.
Function arguments are passed using registers. If your assembly-language routine calls any function that requires more parameters than will fit into registers $r_3$ to $r_{10}$ and $fp_1$ to $fp_8$, you need to pass that size to \texttt{fralloc}. In the case of integer values, registers $r_3$ - $r_{10}$ are used. For floating-point values, registers $fp_1$ - $fp_8$ are used.

As an example, if you pass 12 values of type \texttt{long integer} to your assembly function, this would consume 16 bytes of the parameter area. Registers $r_3$ - $r_{10}$ will hold eight integers, leaving 4 byte integers in the parameter area.

### 20.2.3 Referring to Variables in Instruction Operands

For instructions that require register operands, (such as the \texttt{add} instruction), global variables, function parameters, and local variables must be declared with the keyword \texttt{register}.

Listing 20-6 on page 337 shows inline assembly language statements that correctly use C-language variables as operands in instructions that require register operands.

**Listing 20-6. Using C Variables with Instructions that Require Register Operands**

```
int my_global asm("r14")= 25; /* global variable */
int my_loc asm("r15")= 1;
asm void red(register int *my_param)
{
    register int result;
    fralloc
    add result, r4, my_global /* line 10 */
    add result, my_global, my_param /* line 11 */
    add result, my_param, my_loc /* line 12 */
    frfree
    blr
}
```

In Listing 20-6 on page 337, the statement on line 10, 11, and 12 are all correct because their operands are all declared with the \texttt{register} keyword.

### 20.2.4 Referring to Variables in Memory

For instructions that take a memory operand (such as the \texttt{lwz} instruction), follow these rules when using a C-language variable as an operand:

- Global variables and function parameters must:
  
  Be declared/assigned with the register.
Referring to Assembly, C, and C++ Variables

- Adhere to the syntax below when used as operands, so they are treated as an offset from zero.

\[
\text{instrName regName , 0(globaNameVarName )}
\]

or

\[
\text{instrName regName , 0(parameterName )}
\]

Listing 20-7 on page 338 shows inline assembly language statements that correctly use C-language variables as operands in instructions that take a memory operand.

**Listing 20-7. Using C Variables with Instructions that Take a Memory Operand**

```c
int my_global asm("r14")= 25; /* global variable */
int my_loc = 1;   /* my_loc is on memory, not in a register */
asm void red(register int *my_param)
{
    lwz r4, 0(my_global) /* line 9 */
    lwz r4, 0(my_param) /* line 10 */
    lwz r4, my_loc /* line 11 */
    lwz r4, my_loc(SP)
    blr
}
```

In Listing 20-7 on page 338:

- The statement on line 9 is correct.
  - The operand is fully expressed (because it is an offset from zero).
  - The argument `my_global` is in a register.
- The statement on line 10 is correct for the same reasons as stated above.
- The statement on line 11 is correct.

The CodeWarrior inline assembler automatically adds the contents of the `SP` register to local variable `my_loc`.

- The statement on line 12 is correct.

Note that statements 11 and 12 are equivalent.

As mentioned above, the inline assembler automatically adds the `SP` register to local variable `my_loc`, so explicitly including `(SP)` is redundant.

### 20.2.5 Referring to Registers

For a register operand, you must use one of the register names of the appropriate kind for the instruction. The register names are case-sensitive. You also can use a symbolic name for an argument or local variable that was assigned to a register.
The general registers are `SP, r0 to r31`, and `gpr0 to gpr31`. The floating-point registers are `fp0 to fp31` and `f0 to f31`. The condition registers are `cr0 to cr7`.

### 20.2.6 Referring to Labels

For a label operand, you can use the name of a label. For long branches (such as `b` and `bl` instructions) you can also use function names. For `bla` and `la` instructions, use absolute addresses.

For other branches, you must use the name of a label. For example,

- `b @3` - correct syntax for branching to a local label
- `b red` - correct syntax for branching to external function `red`
- `bl @3` - correct syntax for calling a local label
- `bl red` - correct syntax for calling external function `red`
- `bne red` - incorrect syntax; short branch outside function `red`

**NOTE**

You cannot use local labels that have already been declared in other functions.

### 20.2.7 Using Variable Names as Memory Locations

Whenever an instruction, such as a load instruction, a store instruction, or `la`, requires a memory location, you can use a local or global variable name. You can modify local variable names with struct member references, class member references, array subscripts, or constant displacements. For example, all the local variable references in Listing 20-8 on page 339 are valid.

**Listing 20-8. Example of referring to variables stored in memory locations**

```c
asm void red(void){
    long myVar;
    long myArray[1];
    Rect myRectArray[3];
    fralloc
    lwz r3,myVar(SP)
    la r3,myVar(SP)
    lwz r3,myRect.top
    lwz r3,myArray[2](SP)
    lwz r3,myRectArray[2].top
    lbz r3,myRectArray[2].top+1(SP)
    frfree
```
You can also use a register variable that is a pointer to a *struct* or *class* to access a member of the object, shown in **Listing 20-9 on page 340**.

**Listing 20-9. Example of referring to a struct or class member**

```c
void red(void){
    Rect q;
    register Rect *p = &q;
    asm {
        lwz r3,p->top;
    }
}
```

You can use the `@hiword` and `@loword` directives to access the high and low four bytes of 8 byte long longs and software floating-point doubles (**Listing 20-10 on page 340**).

**Listing 20-10. Example of referring to high and low words**

```c
long long gTheLongLong = 5;
asm void Red(void);
asm void Red(void)
{
    fralloc
    lwz r5, gTheLongLong@hiword
    lwz r6, gTheLongLong@loword
    frfree
    blr
}
```

### 20.2.8 Using Immediate Operands

For an immediate operand, you can use an integer or enum constant, `sizeof` expression, and any constant expression using any of the C dyadic and monadic arithmetic operators.

These expressions follow the same precedence and associativity rules as normal C expressions. The inline assembler carries out all arithmetic with 32-bit signed integers.

An immediate operand can also be a reference to a member of a struct or class type. You can use any struct or class name from a `typedef` statement, followed by any number of member references. This evaluates to the offset of the member from the start of the struct. For example:

```c
lwz   r4,Rect.top(r3)
addi  r6,r6,Rect.left
```

As a side note, `la rD,d(rA)` is the same as `addi rD,rA,d`.
You also can use the top or bottom half-word of an immediate word value as an immediate operand by using one of the @ modifiers (Listing 20-11 on page 341).

**Listing 20-11. Example of referring to immediate operands**

```c
long gTheLong;
asm void red(void)
{
  fralloc
  lis r6, gTheLong@ha
  addi r6, r6, gTheLong@h
  lis r7, gTheLong@h
  ori r7, br7, gTheLong@l
  frfree
  blr
}
```

The access patterns are:

- `lis x, var@ha`
- `la x, var@l(x)`

Or

- `lis x, var@h`
- `ori x, x, var@l`

In this example, `la` is the simplified form of `addi` to load an address. The instruction `las` is similar to `la` but shifted. Refer to the Freescale Power Architecture manuals for more information.

Using `@ha` is preferred since you can write:

- `lis x, var@ha`
- `lwz v, var@l(x)`

You cannot do this with `@h` because it requires that you use the `ori` instruction.

### 20.3 Assembler Directives

This section describes some special assembler directives that the PowerPC built-in assembler accepts. These directives are:

- `entry`
- `fralloc`
- `frfree`
- `machine`
20.3.1 entry

Defines an entry point into the current function.

```
entry [ extern | static ]
_name
```

Use the `extern` qualifier to declare a global entry point; use the `static` qualifier to declare a local entry point. If you leave out the qualifier, `extern` is assumed.

**NOTE**
Inline-assembly directive `entry` can be used only with Function-level assembly code.

Listing 20-12 on page 342 shows how to use the `entry` directive.

Listing 20-12. Using the `entry` directive

```
void __save_fpr_15(void);
void __save_fpr_16(void);
asm void __save_fpr_14(void)
{
  stfd fp14,-144(SP)
  entry __save_fpr_15
  stfd fp15,-136(SP)
  entry __save_fpr_16
  stfd fp16,-128(SP)
  // ...
  blr
}
```

20.3.2 fralloc

Creates a stack frame for a function and reserves registers for local register variables.

```
fralloc [ number ]
```

You need to create a stack frame for a function if the function:

- calls other functions.
- uses more arguments than will fit in the designated parameters (`r3 - r10`, `fp1 - fp8`).
- declares local registers.
- declares non-registered parameters.
The `fralloc` directive has an optional argument number that lets you specify the size in bytes of the parameter area of the stack frame. The compiler creates a 0-byte parameter area. If your assembly language routine calls any function that requires more parameters than will fit in `r3 - r10` and `fp1 - fp8`, you must specify a larger amount.

### 20.3.3 frfree

Frees a function's stack frame and restores local register variables.

```assembly
frfree
```

This directive frees the stack frame and restores the registers that `fralloc` reserved.

The `frfree` directive does not generate a `blr` instruction. If your function uses function-level inline assembly, you must explicitly terminate it with this instruction.

### 20.3.4 machine

Specifies the processor that the assembly language targets.

```assembly
machine
number
```

The value of `number` must be one of those listed in the table below.

<table>
<thead>
<tr>
<th>505</th>
<th>509</th>
<th>555</th>
<th>56x</th>
</tr>
</thead>
<tbody>
<tr>
<td>all</td>
<td>generic</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If you use `generic`, the compiler supports the core instructions for the 603, 604, 740, and 750 processors. In addition, the compiler supports all optional instructions.

If you use `all`, the compiler recognizes assembly instructions for all core and optional instructions for all Power Architecture processors.

If you do not use the `machine` directive, the compiler uses the settings you selected from the Processor listbox of the EPPC Processor settings panel.
20.3.5 nofralloc

Specifies that the function will build a stack frame explicitly.

nofralloc

Use the nofralloc directive so that an inline assembly function does not build a stack frame. When you use nofralloc, if you have local variables, parameters or make function calls, you are responsible for creating and deleting your own stack frame. For an example of nofralloc, see the file __start.c in the directory:

InstallDir\PowerPC_EABI_Support\Runtime\Src

where InstallDir is the name of the directory on your host computer where you installed your CodeWarrior development tools.

20.3.6 opword

Inserts raw bytes into the object code.

opword value

This directive inserts value into the object code. For example

opword 0x7C0802A6

is equivalent to

mflr r0

The compiler does not check the validity of value; the compiler simply copies it into the object code that it generates.

20.3.7 equ

Sets the value of symbol to expression.

equ symbol, expression

For example
Intrinsic functions are a mechanism you can use to get assembly language into your source code without using the `asm` keyword. Intrinsic functions are not part of the ISO/IEC C or C++ standards. They are an extension provided by the CodeWarrior compilers.

There is an intrinsic function for several common processor op-codes (instructions). Rather than using inline assembly syntax and specifying the op-code in an `asm` block, you call the intrinsic function that matches the op-code.

When the compiler encounters the intrinsic function call in your source code, it does not actually make a function call. The compiler substitutes the assembly instruction that matches your function call. As a result, no function call occurs in the final object code. The final code is the assembly language instructions that correspond to the intrinsic functions.

### 20.4.1 Low-Level Processor Synchronization

These functions perform low-level processor synchronization.

- `void __eieio(void)` - Enforce in-order execution of I/O
- `void __sync(void)` - Synchronize
- `void __isync(void)` - Instruction synchronize

For more information on these functions, see the instructions `eieio`, `sync`, and `isync` in PowerPC Microprocessor Family: The Programming Environments by Freescale.

### 20.4.2 Absolute Value Functions
These functions generate inline instructions that take the absolute value of a number.

- `int __abs(int)` - Absolute value of an integer
- `float __fabs(float)` - Absolute value of a float
- `float __fnabs(float)` - Negative absolute value of a float
- `long __labs(long)` - Absolute value of a long int

`__fabs(float)` and `__fnabs(float)` are not available if the Hardware option button is cleared in the EPPC Processor settings panel.

### 20.4.3 Byte-Reversing Functions

These functions generate inline instructions that can dramatically speed up certain code sequences, especially byte-reversal operations.

- `int __lhbrx(const void *, int)` - Load halfword byte; reverse indexed
- `int __lwbrx(const void *, int)` - Load word byte; reverse indexed
- `void __sthbrx(unsigned short, const void *, int)` - Store halfword byte; reverse indexed
- `void __stwbrx(unsigned int, const void *, int)` - Store word byte; reverse indexed

### 20.4.4 Setting the Floating-Point Environment

This function lets you change the Floating Point Status and Control Register (FPSCR). It sets the FPSCR to its argument and returns the original value of the FPSCR.

This function is not available if you select the None option button in the EPPC Processor settings panel.

```c
float
__setflm(float);
```

shows how to set and restore the FPSCR.

#### Listing 20-13. Example of setting the FPSCR

```c
double old_fpscr;
/* Clear flag/exception/mode bits, save original settings */
old_fpscr = __setflm(0.0);
/* Perform some floating-point operations */
__setflm(old_fpscr); /* Restores the FPSCR */
```
20.4.5 Manipulating the Contents of a Variable or Register

These functions rotate the contents of a variable to the left:

- int __rlwinm(int, int, int, int) - Rotate left word (immediate), then AND with mask
- int __rlwnm(int, int, int, int) - Rotate left word, then AND with mask
- int __rlwimi(int, int, int, int, int) - Rotate Left word (immediate), then mask insert

The first argument to __rlwimi is overwritten. However, if the first parameter is a local variable allocated to a register, it is both an input and output parameter. For this reason, this intrinsic should always be written to put the result in the same variable as the first parameter as shown here:

```c
ra = __rlwimi( ra, rs, sh, mb, me );
```

You can count the leading zeros in a register using this intrinsic:

```c
int
__cntlzw(int);
```

You can use inline assembly for a complete assembly language function, as well as individual assembly language statements.

20.4.6 Data Cache Manipulation

The intrinsics shown in the table below, map directly to Power Architecture assembly instructions

<table>
<thead>
<tr>
<th>Intrinsic Prototype</th>
<th>Power Architecture Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>void __dcbf(const void *, int);</td>
<td>dcbf</td>
</tr>
<tr>
<td>void __dcbt(const void *, int);</td>
<td>dcbt</td>
</tr>
<tr>
<td>void __dcbst(const void *, int);</td>
<td>dcbst</td>
</tr>
<tr>
<td>void __dcbtst(const void *, int);</td>
<td>dcbtst</td>
</tr>
<tr>
<td>void __dcbz(const void *, int);</td>
<td>dcbz</td>
</tr>
<tr>
<td>void __dcba(const void *, int);</td>
<td>dcba</td>
</tr>
</tbody>
</table>
20.4.7 Math Functions

The table below lists intrinsic functions for mathematical operations.

<table>
<thead>
<tr>
<th>Intrinsic Prototype</th>
<th>Power Architecture Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>int __mulhw(int, int);</td>
<td>mulhw</td>
</tr>
<tr>
<td>uint __mulhwu(uint, uint);</td>
<td>mulhwu</td>
</tr>
<tr>
<td>double __fmadd(double, double, double);</td>
<td>fmadd</td>
</tr>
<tr>
<td>double __fmsub(double, double, double);</td>
<td>fmsub</td>
</tr>
<tr>
<td>double __fnmadd(double, double, double);</td>
<td>fnmadd</td>
</tr>
<tr>
<td>double __fnmsub(double, double, double);</td>
<td>fnmsub</td>
</tr>
<tr>
<td>float __fmaids(float, float, float);</td>
<td>fnmadds</td>
</tr>
<tr>
<td>float __fmsubs(float, float, float);</td>
<td>fmsubs</td>
</tr>
<tr>
<td>float __fnmadds(float, float, float);</td>
<td>fnmadds</td>
</tr>
<tr>
<td>float __fnmsubs(float, float, float);</td>
<td>fnmsubs</td>
</tr>
<tr>
<td>double __mffs(void);</td>
<td>mffs</td>
</tr>
<tr>
<td>float __fabsf(float);</td>
<td>fabsf</td>
</tr>
<tr>
<td>float __fnabsf(float);</td>
<td>fnabsf</td>
</tr>
</tbody>
</table>

20.4.8 Buffer Manipulation

Some intrinsics allow control over areas of memory, so you can manipulate memory blocks.

void *__alloca(ulong);

__alloca implements alloca() in the compiler. char *__strcpy(char *, const char *);

__strcpy() detects copies of constant size and calls __memcpy(). This intrinsic requires that a __strcpy function be implemented because if the string is not a constant it will call __strcpy to do the copy.

void *__memcpy(void *, const void *, size_t);

__memcpy() provides access to the block move in the code generator to do the block move inline.
Chapter 21
Power Architecture Code Generation

This chapter describes the conventions that the C/C++ compiler and linker follow to generate object code for Power Architecture processors, the data types that the compiler recognizes, and how to specify to the compiler the byte-alignment of data in object code.

- ABI Conformance
- Data Representation
- Data Addressing
- Aligning Data
- Small Data Area PIC/PID Support
- Variable Length Encoding
- Building a ROM Image
- Embedded C++

21.1 ABI Conformance

The CodeWarrior compiler for Power Architecture processors follows the application binary interface (ABI) specified by *PowerPC Embedded Binary Interface, 32-Bit Implementation*.

21.2 Data Representation

The compiler recognizes ISO standard data types and some Power Architecture-specific types:

- Boolean Type
- Character Types
21.2.1 Boolean Type

Table 21-1 lists the name, size, and range of the boolean data type. The compiler recognizes this data type when compiling C99 (ISO/IEC 9899-1999) source code.

<table>
<thead>
<tr>
<th>This type</th>
<th>has this size</th>
<th>and holds this range of values</th>
</tr>
</thead>
<tbody>
<tr>
<td>_Bool</td>
<td>8 bits when pragma uchar_bool is on, 32 bits when pragma uchar_bool is off</td>
<td>0 (&quot;false&quot;) and 1 (&quot;true&quot;)</td>
</tr>
</tbody>
</table>

Table 21-2 lists the name, size, and range of the C++ boolean data type. The C++ compiler does not recognize the C99 _Bool type.

<table>
<thead>
<tr>
<th>This type</th>
<th>has this size</th>
<th>and holds this range of values</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>8 bits when pragma uchar_bool is on, 32 bits when pragma uchar_bool is off</td>
<td>true, false</td>
</tr>
</tbody>
</table>

21.2.2 Character Types

Table 21-3 lists the name, size, and range of the character data types.

<table>
<thead>
<tr>
<th>This type</th>
<th>has this size</th>
<th>and holds this range of values</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>8 bits</td>
<td>either -128 to 127 or 0 to 255</td>
</tr>
<tr>
<td>unsigned char</td>
<td>8 bits</td>
<td>0 to 255</td>
</tr>
<tr>
<td>signed char</td>
<td>8 bits</td>
<td>-128 to 127</td>
</tr>
</tbody>
</table>
Table 21-4 lists the name, size, and range of the C++ `wchar_t` data types.

### Table 21-4. Character data types

<table>
<thead>
<tr>
<th>This type</th>
<th>has this size</th>
<th>and holds this range of values</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>wchar_t</code></td>
<td>16 bits</td>
<td>either -32768 to 32767 or 0 to 65535</td>
</tr>
<tr>
<td><code>unsigned wchar_t</code></td>
<td>16 bits</td>
<td>0 to 65535</td>
</tr>
<tr>
<td><code>signed wchar_t</code></td>
<td>16 bits</td>
<td>-32768 to 32767</td>
</tr>
</tbody>
</table>

The pragma `unsigned_char` controls whether or not the compiler treats the `wchar_t` and `char` types as signed or unsigned.

### 21.2.3 Integer Types

Table 21-5 lists the name, size, and range of the integer data types.

### Table 21-5. Integer data type

<table>
<thead>
<tr>
<th>This type</th>
<th>has this size</th>
<th>and holds this range of values</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>short</code></td>
<td>16 bits</td>
<td>-32,768 to 32,767</td>
</tr>
<tr>
<td><code>unsigned short</code></td>
<td>16 bits</td>
<td>0 to 65,535</td>
</tr>
<tr>
<td><code>int</code></td>
<td>32 bits</td>
<td>-2,147,483,648 to 2,147,483,647</td>
</tr>
<tr>
<td><code>unsigned int</code></td>
<td>32 bits</td>
<td>0 to 4,294,967,295</td>
</tr>
<tr>
<td><code>long</code></td>
<td>32 bits</td>
<td>-2,147,483,648 to 2,147,483,647</td>
</tr>
<tr>
<td><code>unsigned long</code></td>
<td>32 bits</td>
<td>0 to 4,294,967,295</td>
</tr>
<tr>
<td><code>long long</code></td>
<td>64 bits</td>
<td>-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807</td>
</tr>
<tr>
<td><code>unsigned long long</code></td>
<td>64 bits</td>
<td>0 to 18,446,744,073,709,551,615</td>
</tr>
</tbody>
</table>

The compiler recognizes the `long long` data type when pragma `longlong` is on or when compiling C99 source code (ISO/IEC 9899-1999 standard).

### 21.2.4 Floating-Point
Table 21-6 lists the floating point data types

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>32 bits</td>
<td>1.17549e-38 to 3.40282e+38</td>
</tr>
<tr>
<td>double</td>
<td>64 bits</td>
<td>2.22507e-308 to 1.79769e+308</td>
</tr>
<tr>
<td>long double</td>
<td>64 bits</td>
<td>2.22507e-308 to 1.79769e+308</td>
</tr>
</tbody>
</table>

21.2.5 AltiVec™ Data Types

There are vector data types for use in writing AltiVec-specific code. (See Table 21-7). All the types are a constant size, 16 bytes (128 bits). This is due to the AltiVec programming model which is optimized for quantities of this size.

<table>
<thead>
<tr>
<th>Vector Data Type</th>
<th>Contents</th>
<th>PossibleValues</th>
</tr>
</thead>
<tbody>
<tr>
<td>vector unsigned char</td>
<td>16 unsigned char</td>
<td>0 to 255</td>
</tr>
<tr>
<td>vector signed char</td>
<td>16 signed char</td>
<td>-128 to 127</td>
</tr>
<tr>
<td>vector bool char</td>
<td>16 unsigned char</td>
<td>0 (&quot;false&quot;), 1 (&quot;true&quot;)</td>
</tr>
<tr>
<td>vector unsigned short [int]</td>
<td>8 unsigned short</td>
<td>0 to 65535</td>
</tr>
<tr>
<td>vector signed short [int]</td>
<td>8 signed short</td>
<td>-32768 to 32767</td>
</tr>
<tr>
<td>vector bool short [int]</td>
<td>8 unsigned short</td>
<td>0 (&quot;false&quot;), 1 (&quot;true&quot;)</td>
</tr>
<tr>
<td>vector unsigned long [int]</td>
<td>4 unsigned int</td>
<td>0 to 232 - 1</td>
</tr>
<tr>
<td>vector signed long [int]</td>
<td>4 signed int</td>
<td>-231 to 231-1</td>
</tr>
<tr>
<td>vector bool long [int]</td>
<td>4 unsigned int</td>
<td>0 (&quot;false&quot;), 1 (&quot;true&quot;)</td>
</tr>
<tr>
<td>vector float</td>
<td>4 float</td>
<td>any IEEE-754 value</td>
</tr>
<tr>
<td>vector pixel</td>
<td>8 unsigned short</td>
<td>1/5/5/5 pixel</td>
</tr>
</tbody>
</table>

In Table 21-7, the [int] portion of the Vector Data Type is optional.

There are two additional keywords besides pixel and vector, __pixel and __vector.

The bool keyword is not a reserved word in C unless it is used as an AltiVec vector data type.

21.3 Data Addressing
In absolute addressing, the compiler generates two instructions to fetch the address of a variable. For example the compiler translates Listing 21-1 on page 353 into the instructions in Listing 21-2 on page 353.

Listing 21-1. Source Code

```c
int red;
int redsky;
void sky()
{
    red = 1;
    redsky = 2;
}
```

Listing 21-2. Generated Code

```
li   r3,1
lis  r4,red@ha
addi r4,r4,red@l
stw  r3,0(r4)
li   r5,2
lis  r6,redsky@ha
addi r6,r6,redsky@l
stw  r5,0(r6)
```

Each variable access takes two instructions and a total of four bytes to make a simple assignment. If you set the small data threshold to be at least the size of an `int` data type, the compiler generates instructions to fetch variables with one instruction (Listing 21-3 on page 353).

Listing 21-3. Fetching variables with one instruction

```
li   r3,1
stw  r3,red
li   r4,2
stw  r4,redsky
```

Because small data sections are limited in size you might not be able to put all of your application data into the small data and small data2 sections. We recommend that you make the threshold as high as possible until the linker reports that you have exceeded the size of the section.

If you do exceed the available small data space, consider using pooled data.

Because the linker can not deadstrip unused pooled data, you should:

1. Check the Generate Link Map and List Unused Objects checkboxes in the CodeWarrior IDE’s EPPC Linker Properties > C/C++ Build > Settings > Tool Settings > PowerPC Linker -> Output panel.
2. Link and examine the map for data objects that are reported unused.
3. Delete or comment out those used definitions in your source.
4. Select Properties > C/C++ Build > Settings > Tool Settings > PowerPC Compiler > Processor, check Check the Pool Data checkbox.
The code in Listing 21-4 on page 354 has a zero small data threshold.

### Listing 21-4. Zero Small Data Threshold

```
lis   r3,...bss.0@ha
addi  r3,r3,...bss.0@l
li    r0,1
stw   r0,0(r3)
li    r0,2
stw   r0,4(r3)
```

When pooled data is implemented, the first used variable of either the `.data`, `.bss` or `.rodata` section gets a two-instruction fetch of the first variable in that section. Subsequent fetches in that function use the register containing the already-loaded section address with a calculated offset.

**NOTE**

You can access small data in assembly files with the two-instruction fetch used with large data, because any data on your board can be accessed as if it were large data. The opposite is not true; large data can never be accessed with small data relocations (the linker issues an error if you try to do so). External declarations of empty arrays (for example, `extern int red=[];`) are always treated as if they were large data. If you know that the size of the array fits into a small data section, specify the size in the brackets.

## 21.4  Aligning Data

This section contains these topics:

- Alignment Attribute Syntax
- Aligning a Variable Declaration
- Alignment in a Structure Definition
- Typedef Declaration
- Structure Member
- Bitfields

### 21.4.1  Alignment Attribute Syntax
Use `__attribute__((aligned(...)))` directive to specify to the compiler on what memory boundary to store data objects. This directive specifies which multiple of bytes to store an object.

The format of this directive is

```c
__attribute__((aligned(x)))
```

where `x` is a decimal number of a power of 2 from 1 to 8192.

### 21.4.2 Aligning a Variable Declaration

Use the alignment attribute to specify a variable's alignment. For example, the following variable declaration aligns `v1` on a 16-byte boundary.

```c
int V1[4] __attribute__((aligned(16)));
```

The following variable declaration aligns `v2` on a 2-byte boundary.

```c
int V2[4] __attribute__((aligned(2)));
```

### 21.4.3 Alignment in a Structure Definition

Use the alignment attribute to specify how instances of a structure should be aligned. You must specify a minimum alignment of at least 4 bytes for structures. Specifying a lower number might cause alignment exceptions at runtime.

For example, this definition aligns all definitions of `struct S1` on an 8-byte boundary.

```c
struct S1 { short f[3]; } __attribute__((aligned(8)));
struct S1 s1;
```

The following definition aligns all definitions of `struct S2` on a 4-byte boundary.

```c
struct S2 { short f[3]; } __attribute__((aligned(1)));
struct S2 s2;
```
21.4.4 Structure Member

Use the alignment attribute to specify how to align a member in a structure.

For example, the following structure member definition aligns all definitions of `struct S3` on an 8-byte boundary, where `a` is at offset 0 and `b` is at offset 8.

```c
struct S3 {  
    char a;
    int b __attribute__((aligned (8)));
};  
struct S3 s3;
```

The following struct member definition aligns all definitions of `struct S4` on a 4-byte boundary, where `a` is at offset 0 and `b` is at offset 4.

```c
struct S4 {  
    char a;
    int b __attribute__((aligned (2)));
};  
struct S4 s4;
```

**NOTE**

Specifying `__attribute__((aligned (2)))` does not affect the alignment of `S4` because 2 is less than the natural alignment of `int`.

21.4.5 Typedef Declaration

Use the alignment attribute to specify how objects of a specific type should be aligned.

For example, the following typedef declaration aligns all definitions of `T1` on an 8-byte boundary.

```c
typedef int T1 __attribute__((aligned (8)))
            T1 t1;
```

The following typedef declaration aligns all definitions of `T2` on an 1-byte boundary.

```c
typedef int T2 __attribute__((aligned (1)))
            T2 t2;
```
21.4.6 Bitfields

If your program's structure has bitfields and the Power Architecture alignment does not give you as small a structure as you desire, double-check that you are specifying the smallest integer size for your bitfields.

For example, Listing 21-5 on page 357 would be smaller if it were written as shown in Listing 21-6 on page 357.

Listing 21-5. Before

```c
typedef struct red {
    unsigned a: 1;
    unsigned b: 1;
    unsigned c: 1;
} red;
```

Listing 21-6. After

```c
typedef struct red {
    unsigned char a: 1;
    unsigned char b: 1;
    unsigned char c: 1;
} red;
```

21.5 Small Data Area PIC/PID Support

The basic requirement for position independent code and data in the small data area is, at runtime, maintaining the link time address relationships between the startup code (.init) and the .sdata and .sdata2 segments. For example, if the link time addresses are:

```
.init = 0x00002000
.sdata2 = 0x00003000
.sdata = 0x00004000
```

but .init somehow is executed at 0x00002500, then those link time addresses must all increment by 0x00000500 for their runtime addresses.

Any segment that does not maintain the address relationship at runtime is considered external and must be addressed with absolute addresses. Segments that do maintain their link time address relationship at runtime are considered internal and must be addressed with PC-relative and SDA-relative addressing.

- Internal and External Segments and References
21.5.1 Internal and External Segments and References

The linker determines at link time whether code and data segments are external or internal. Internal segments reference their data as far or near offsets of the small data registers r2 and r13. Their code references are normally PC-relative, but if far code references are needed, they also use offsets of the small data registers.

Internal segments can also reference code and data in other internal segments with the same addressing that they would use for their own code and data.

By default, the linker considers all segments in your application to be internal with the exception of segments that are at absolute addresses. Segments with names such as .abs.xxxxxxxxx, where xxxxxxxx is a hex address, are considered external.

External segments reference their data with absolute addressing and code references within the segment may be either PC-relative or absolute. Any other segment must use absolute references to reference code or data in external segments. External segments must reference an internal segment with small data registers for code and data.

Related to external segments are external symbol references. These are symbols, usually linker-generated, that are determined not to be within any segment in your application. They are referenced with absolute addressing. All symbols in an external segment are considered to be external symbol references.

21.5.2 PIC/PID Linker Command File Directives

A few linker command file directives override PIC/PID related linker default settings:

- MEMORY
- INTERNAL_SYMBOL
- EXTERNAL_SYMBOL
21.5.3 Linker-defined Symbols

The linker-generated start and end symbols that are automatically generated for loadable segments are internal if they are addresses into internal segments, and external if they are for external segments. All other linker defined symbols you create in a LCF are considered external unless you redefine them with `INTERNAL_SYMBOL` . The linker also defines some linker defined symbols for its own use (Table 21-8).

### Table 21-8. Linker-defined Symbols

<table>
<thead>
<tr>
<th>Symbol Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_stack_addr</td>
<td>top of the stack -</td>
<td>External. Comes from settings panel settings.</td>
</tr>
<tr>
<td>_stack_end</td>
<td>bottom of the stack</td>
<td>External. Comes from settings panel settings.</td>
</tr>
<tr>
<td>_heap_addr</td>
<td>bottom of the heap</td>
<td>External. Comes from settings panel settings.</td>
</tr>
<tr>
<td>_heap_end</td>
<td>top of the heap</td>
<td>External. Comes from settings panel settings.</td>
</tr>
<tr>
<td><em>SDA_BASE</em></td>
<td>.sdata + 0x0000000</td>
<td>Internal per EABI requirement. May not be redefined.</td>
</tr>
<tr>
<td><em>SDA2_BASE</em></td>
<td>.sdata2 + 0x0000000</td>
<td>Internal per EABI requirement. May not be redefined.</td>
</tr>
<tr>
<td><em>ABS_SDA_BASE</em></td>
<td>.sdata + 0x0000000</td>
<td>External version of <em>SDA_BASE</em> that can be used as an absolute. May not be redefined.</td>
</tr>
<tr>
<td><em>ABS_SDA2_BASE</em></td>
<td>.sdata2 + 0x0000000</td>
<td>External version of <em>SDA2_BASE</em> that can be used as an absolute. May not be redefined.</td>
</tr>
<tr>
<td>_nbfunctions</td>
<td>number of functions in program</td>
<td>Deprecated. External. This is a number, not an address. May not be redefined.</td>
</tr>
<tr>
<td>SIZEOF_HEADERS</td>
<td>size of the segment headers</td>
<td>External. This is a number, not an address. May not be redefined.</td>
</tr>
</tbody>
</table>

**NOTE**

The symbols _SDA_BASE_ and _SDA2_BASE_ are not accessible until the small data registers are properly initialized before being accessible. The symbols _ABS_SDA_BASE_ and _ABS_SDA2_BASE_ allow you to access those pointers as absolutes addresses, as it is difficult to initialize those pointers without accessing them as absolute addresses.
NOTE
The stack and heap linker generated symbols are external. It may be more practical in a SDA PIC/PID application to make the heap and stack be contiguous with an internal segment and define them as internal.

21.5.4 Uses for SDA PIC/PID

The PIC/PID runtime can be used for different scenarios:

1. All code and data segments are internal. The simplest case would be for all segments to use the same MEMORY directive and to have all of the .bss type segments at the end. In such a simple case, the application could be converted to a binary file and linked into another application which could copy it to RAM and jump to its entry point.
2. All of the essential segments are internal and therefore moveable. But, there may be some external segments which are absolute. This situation is probably difficult to test but we can download the entire application to the chip and at least debug it at its link time addresses.
3. There are internal and external segments, but the application is linked as a ROM image (the application does not need to be flashed to ROM, however). It is possible to change the ROM Image Address to be an address into RAM and have the debugger download the image to the RAM address. Alternatively, we could have the ROM image converted to a binary file and linked into another application as in 1, above. The structures used in __init_data(), __rom_copy_info and __bss_init_info, have been modified for SDA PIC/PID to have an extra field which tells the runtime where the segment is internal or external so that the internal segments are copied to position-relative addresses and the external segments copied to absolute addresses.

21.5.5 Building an SDA PIC/PID Application

To build a SDA PIC/PID application, select SDA PIC/PID in the ABI list box in the CodeWarrior IDE's EPPC Target target preferences Properties > C/C++ Build > Settings > Tool Settings > PowerPC CPU panel. The compiler defines a simple variable that we can use to guard PIC/PID source.

```c
#if __option(sda_pic_pid) // is true if we have chosen SDA PIC/PID ABI
```
At link-time, the linker generates a table used for the runtime files \_ppc\_eabi\_init.cpp and \_ppc\_eabi\_init.c.

If our application contains absolute addressing relocations, we will receive linker warnings telling us that those relocations may cause a problem. To resolve these warnings, either:

- change the Code Model listbox in the CodeWarrior IDE’s EPPC Target target preferences Properties > C/C++ Build > Settings > Tool Settings > PowerPC CPU panel to be SDA Based PIC/PID Addressing for all of our sources and libraries
- check the Tune Relocations checkbox in the EPPC Target target preferences Properties > C/C++ Build > Settings > Tool Settings > PowerPC CPU panel. This new option is only available for the EABI and SDA PIC/PID ABIs. For EABI, it changes 14-bit branch relocations to 24-bit branch relocations, but only if they can not reach the calling site from the original relocation.

For SDA PIC/PID, this option changes absolute-addressed references of data from code to use a small data register instead of \_r0 and changes absolute code-to-code references to use the PC-relative relocations.

21.5.5.1 Linking Assembly Files

It is always possible to link in an assembly file that does not behave in a standard way. For example, taking the address of a variable with:

\texttt{addis \ rx,r0,object@h}
\texttt{ori \ rx,rx,objec@l}

generally can not be converted by the linker to SDA PIC/PID Addressing and the linker will warn us if it finds an occurrence.

The following will work with Absolute Addressing as well as allow the linker to convert the instructions to SDA PIC/PID Addressing:

\texttt{addis \ rx,r0,object@ha}
\texttt{addi \ rx,rx,objc@l}

Another possible problem may arise if we put constant initialized pointers into a read-only section, thereby not letting the runtime convert the addresses.
21.5.5.2 Modifications to the SectionPragma

The pragma `#pragma section` has been modified to accept `far_sda_rel` for the `data_mode` and `code_mode` options, even if we are not using Code Model SDA Based PIC/PID Addressing. If we omit these options, the compiler uses the Code Model to determine the appropriate modes.

- Absolute Addressing

```c
data_mode = far_abs
code_mode = pc_rel
```

- SDA Based PIC/PID Addressing

```c
data_mode = far_sda_rel
code_mode = pc_rel
```

21.5.6 Internal and External Addressing Modes

An address mode is applied to a memory segment as a part of the ROM image or at the executing (or logical) address of the segment. Following address modes can be applied to a memory segment:

- Internal-the segment executes from an address not specified at link time.
- External- the segment must execute from the address specified at the link time.

Consider an example where the segment `.foo` is a part of ROM Image and will be copied to a RAM location. The link time addresses are:

- ROM = 0x00100000
- RAM = 0x00002000

**NOTE**

Both the link time addresses can be external or internal.
Also assume that the real time (physical) ROM address is 0x00200000 instead of the link time specified address 0x00100000. Table 21-9 lists the possible address mode scenarios.

Table 21-9. Possible addr_mode Scenarios

<table>
<thead>
<tr>
<th>Scenario</th>
<th>ROM addr_mode</th>
<th>RAM addr_mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>internal</td>
<td>external</td>
<td>Runtime correctly figures out that the ROM address is 0x00200000 and copies it to 0x00002000</td>
</tr>
<tr>
<td>B</td>
<td>internal</td>
<td>internal</td>
<td>Runtime correctly figures out that the ROM address is 0x00200000 and copies it to 0x00102000</td>
</tr>
<tr>
<td>C</td>
<td>external</td>
<td>external</td>
<td>Runtime incorrectly assumes that the ROM address is 0x00100000 and copies it to 0x00002000</td>
</tr>
<tr>
<td>D</td>
<td>external</td>
<td>internal</td>
<td>Runtime incorrectly assumes that the ROM address is 0x00100000 and copies it to 0x00102000</td>
</tr>
</tbody>
</table>

In the above possible scenarios only A and B are correct. The difference between scenario A and B is that in A, the executing (logical) address of .foo is absolute and that in B, the executing (logical) address of .foo is relative.

Scenario C and D are possible if .foo is flashed to ROM at its correct ROM address and all other segments are at an offset from their link time ROM addresses.

NOTE
.
.init segment determines the correct address of an application. If .init is at its link time ROM address, then all the segments in the application will be treated as external.

21.5.6.1 Specifying ROM addr_mode

Use the following directives to specify ROM addr_mode:

- LOAD- To specify an external ROM addr_mode.
- INTERNAL_LOAD- To specify an internal ROM addr_mode.

By default the ROM addresses are external.
21.5.6.2 Specifying RAM addr_mode

Use MEMORY directive and any of the following parameters to specify the RAM addr_mode.

addr_mode = external-To specify an external RAM addr_mode.
addr_mode = internal-To specify an internal RAM addr_mode.

By default the RAM addresses are internal.

For example, RAM : org = 0x000e0000, addr_mode = external will make sections defined in the RAM external.

NOTE
addr_mode is ignored if SDA PIC/PID in the ABI list box in the CodeWarrior IDE’s EPPC Target target preferences Properties > C/C++ Build > Settings > Tool Settings > PowerPC CPU panel is not selected.

21.6 Variable Length Encoding

The Variable Length Encoding (VLE) instruction set architecture is an extension to the instruction set specified in Freescale Semiconductor's Book E Implementation Standard (EIS) for Power Architecture processors. This instruction set adds a few identically operating counterparts to the regular EIS instruction set. But where regular EIS instructions occupy 32 bits and must be aligned to 32-bit boundaries, VLE instructions are either 16 or 32 bits long and can be aligned to 16-bit boundaries. This extra flexibility in instruction encoding and alignment allows the compiler and linker to greatly compress the size of runtime object code with only a small penalty in execution performance.

These topics describe how and when to configure the build tools to generate VLE object code:

- Processors With VLE Capability
- Compiling VLE Instructions
- Assembling VLE Instructions
- Linking VLE Object Code
21.6.1 Processors With VLE Capability

The VLE (Variable Length Encoding) instruction set is an extension to the instruction set specified in the Freescale Book E Implementation Standard (EIS). Not all Power Architecture processors have VLE capability. Refer to the manufacturer's documentation for the processor you are targeting. For information on the Book E and VLE programming models, see *EREF: A Programmer's Reference Manual for Freescale Book E Processors*, published by Freescale Semiconductor.

21.6.2 Compiling VLE Instructions

Table 21-10 shows how to control VLE (Variable Length Encoding) code generation.

**Table 21-10. Controlling VLE code generation**

<table>
<thead>
<tr>
<th>To control this option from here...</th>
<th>use this setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeWarrior IDE</td>
<td>Select Zen in the Processor drop-down list box of the Properties &gt; C/C++ Build &gt; Settings &gt; Tool Settings &gt; PowerPC CPU panel, then check Generate VLE ISEL Instructions in Properties &gt; C/C++ Build &gt; Settings &gt; Tool Settings &gt; PowerPC Compiler &gt; Processor panel.</td>
</tr>
<tr>
<td>C/C++ source code</td>
<td>__declspec(vle_on) __declspec(vle_off)</td>
</tr>
<tr>
<td>command line</td>
<td>-vle</td>
</tr>
</tbody>
</table>

21.6.3 Assembling VLE Instructions

The mnemonics for VLE (Variable Length Encoding) instructions begin with "se_" or "e_". The compiler's inline assembler recognizes these mnemonics when the compiler is configured to generate VLE object code.
Only a subset of EIS instructions have equivalent VLE instructions. To save you time and effort, the inline assembler can convert regular EIS instructions to equivalent VLE instructions automatically. In other words, the inline assembler can generate VLE object code from inline assembly statements that use only regular mnemonics. Table 21-11 shows how to control VLE code generation for inline assembly statements.

<table>
<thead>
<tr>
<th>To control this option from here...</th>
<th>use this setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeWarrior IDE</td>
<td>Select Zen in the Processor drop-down list box of the Properties &gt; C/C++ Build &gt; Settings &gt; Tool Settings &gt; PowerPC CPU panel, then check Translate PPC Asm to VLE ASM (Zen) in Properties &gt; C/C++ Build &gt; Settings &gt; Tool Settings &gt; PowerPC Compiler &gt; Processor panel.</td>
</tr>
<tr>
<td>command line</td>
<td>-ppc_asm_to_vle</td>
</tr>
</tbody>
</table>

The stand-alone assembler also recognizes and generates VLE instructions. Table 21-12 shows how to control VLE code generation with the standalone assembler.

<table>
<thead>
<tr>
<th>To control this option from here...</th>
<th>use this setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeWarrior IDE</td>
<td>Select Zen in the Processor drop-down list box of the Properties &gt; C/C++ Build &gt; Settings &gt; Tool Settings &gt; PowerPC CPU panel, then check Generate VLE ISEL Instructions in Properties &gt; C/C++ Build &gt; Settings &gt; Tool Settings &gt; PowerPC Compiler &gt; Processor panel.</td>
</tr>
<tr>
<td>command line</td>
<td>-vle</td>
</tr>
</tbody>
</table>

To specify that a section containing executable code should be executed in the processor's VLE mode, use the text_vle identifier with the .section directive. Figure 21-1 shows examples.

**Figure 21-1. Examples of specifying VLE sections in standalone assembly**

```
.sect .text_vle # Section name is .text_vle
.sect .text,.text_vle # Section name is .text
.sect .littletext,.text_vle # Section name is .littletext
```

Unlike the inline assembler, the standalone assembler does not offer the option to convert regular instruction mnemonics to VLE instructions. To perform this conversion automatically, copy and paste standalone assembly source code into a C or C++ source file, shown in Figure 21-2.

**Figure 21-2. Using automatic VLE instruction conversion**

```c
extern asm void my_func(void)
{
    nofralloc /* No stack frame. */
```
21.6.4 Linking VLE Object Code

A processor capable of executing VLE (Variable Length Encoding) instructions must use separate memory pages for VLE and regular instructions. The compiler and linker ensure this separation by placing executable code that uses VLE instructions and regular instructions in separate object code sections.

To maintain this separation in your own linker command file, specify output sections for VLE and regular instructions. Listing 21-7 on page 367 shows an example. This linker control file specifies that output sections named `.init_vle` and `.text_vle` should only contain object code that the compiler has tagged with `VLECODE`.

Listing 21-7. Separating VLE and regular object code in the linker’s output file

```assembly
.init : { } > code
.init_vle (VLECODE) : {
  *(.init)
  *(.init_vle)
} > code

.text : { } > code
.text_vle (VLECODE) : {
  *(.text)
  *(.text_vle)
} > code
```

To save memory space, the linker compresses VLE object code by shortening the gaps between functions. A VLE function must meet these criteria to be re-aligned:

- The VLE function is referred to only by other VLE functions.
  - The linker will not re-align a function if it is referred to by a non-VLE function.
- The VLE function's alignment is 4 bytes.
  - The linker will not re-align a function if the compiler's function alignment settings specify an explicit alignment value.
- The object code was generated by the CodeWarrior compiler.

21.7 Building a ROM Image
The CodeWarrior compiler and linker can generate a program image that may be stored in and started from ROM (read-only memory). This section uses the term ROM to mean any kind of persistent main storage, including ROM and flash memory.

To create an image for read-only memory, you must configure the compiler and linker:

- Linking a ROM Image
- ROM Image Addresses
- Specifying A Single ROM Block
- Specifying Several ROM Blocks
- Specifying Jump Table Location
- Specifying Constant Data Location

### 21.7.1 Linking a ROM Image

The table below compares the differences between the linker's default RAM image layout and how you should configure your program for loading into and running from ROM.

<table>
<thead>
<tr>
<th>RAM image properties</th>
<th>ROM image properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>The S record file contains executable code, constants, and initialization values for data.</td>
<td>One or more memory areas defined in the linker's .lcf file specifies where store the program image in the target system's memory map.</td>
</tr>
<tr>
<td>Executable code, constants, and data are loaded for execution by the debugger or the program loader.</td>
<td>Initialization values for data are copied from the ROM image to RAM at program startup. Executable code and constant data may also be copied to RAM to improve performance (while requiring more RAM space).</td>
</tr>
</tbody>
</table>

The linker's output ELF file for a ROM image contains a segment named .PPC.EMB.seginfo. This segment describes which segments in the image will be copied from ROM to RAM at runtime. The linker uses this non-loadable segment to generate a data structure named _rom_copy_info. At startup, the program uses the _rom_copy_info structure to determine which segments to move from ROM to RAM.

**Figure 21-3** shows the part of an example disassembly that lists the contents of segment .PPC.EMB.seginfo. When is _rom_copy is set to 1, the corresponding segment is copied from ROM to its final destination during startup. In this example, these sections will be copied from ROM to RAM at startup:

`.bss, .data, .sdata, .sbss, .sdata2.`

**Figure 21-3. Example of segments to copy to RAM at startup**
21.7.2 ROM Image Addresses

The program ROM image is usually the whole image of the program. The ROM image allocates RAM space for its variables, and optionally for its executable code, at application startup. A ROM image is defined by these addresses:

- ROM image address: is the address where you want the ROM image to be allocated. Usually it is the start address of one of the memory blocks defined in the linker .lcf file.

- RAM buffer address: specifies the address in RAM that is to be used as a buffer for the flash image programmer.

**NOTE**

ROM Image address needs to be synchronized with the LCF specified ROM address to avoid conflict between execution address (start address) and load address (ROM address). The linker will pick LCF specified address as the execution address (start address).

To specify these addresses in the CodeWarrior IDE, turn on Generate ROM Image in the EPPC Linker settings panel, then enter addresses in the RAM Buffer Address of ROM Image and ROM Image Address fields on the Properties > C/C++ Build > Settings > Tool Settings > PowerPC Linker > Output panel.

21.7.3 Specifying A Single ROM Block
When specifying a single ROM memory block in a .lcf file, the start address of this memory block can be used as ROM image address. All executable code and constant sections will be allocated in ROM and all variables initialization values will be copied from ROM to RAM during startup.

Listing 21-8 on page 370 shows an example .lcf file for a single ROM block.

Listing 21-8. Configuring a linker file for a ROM image

MEMORY {
    ram : org = 0x00c02000
    rom : org = 0x00000000 // desired ROM address (boot address for 555)
}

SECTIONS {
    .reset : {} > rom
    .init : {} > rom
    GROUP : {
        .text (TEXT) ALIGN(0x1000) : {}
        .rodata (CONST) : {
            *(.rdata)
            *(.rodata)
        }
        .ctors : {}
        .dtors : {}
        extab : {}
        extabindex : {}
    } > rom // for ROM images, this can be 'rom' if you want
    // to execute in ROM or 'ram' if you want to
    // execute in RAM
    GROUP : {
        .data : {}
        .sdata : {}
        .sbss : {}
        .sdata2 : {}
        .sbss2 : {}
        .bss : {}
        .PPC.EMB.sdata0 : {}
        .PPC.EMB.sbss0 : {}
    } > ram
}

21.7.4 Specifying Several ROM Blocks

To specify several ROM blocks in a .lcf file, the start address of the main memory block must be the ROM image address.

To prevent all executable code or constants allocated in other ROM blocks to be copied during startup, use the LOAD linker directive. To prevent a specific executable code or constant section from being copied to its runtime RAM destination, specify the final destination address in the LOAD directive.
Listing 21-9 on page 371 shows an example .lcf file that specifies several ROM blocks.

**Listing 21-9. Configuring linker file for an image with several ROM blocks**

```c
MEMORY {
    APPL_INT_VECT : org= 0x00010000, len= 0x000000FF
    // If org is changed, make sure to adjust start address in
    // .applexctbl LOAD (0x00010000): {} > APPL_INT_VECT
    // accordingly
    CST_DATA : org= 0x00010100, len= 0x000000FF
    APPL_CODE_FLASH : org= 0x00010200, len= 0x000EFE00
    // APPL_CODE_FLASH= int. flash area for application
    // external RAM
    EXT_RAM_A : org= 0x00080000, len= 0x00100000
}

SECTIONS {
    .applexctbl LOAD (0x00010000): {} > APPL_INT_VECT
    .syscall: {} > APPL_CODE_FLASH
    .reset : {} > APPL_CODE_FLASH
    .init: {} > APPL_CODE_FLASH
    GROUP :
        .text (TEXT) : {}
        .rodata (CONST) :
            *(.rdata)
            *(.rodata)
    .ctors : {}
    .dtors : {}
    extab : {}
    extabindex : {}
} > APPL_CODE_FLASH

GROUP :

GROUP :

GROUP :

GROUP:
    .CstData LOAD (0x00010100): {}
} > CST_DATA

If several sections must be allocated in one of the secondary memory areas, use the linker's `ROMADDR` directive to evaluate the final destination address of the sections. Listing 21-10 on page 371 shows an example.

**Listing 21-10. Placing several sections in a secondary memory area**

```c
.applexctbl LOAD (0x00010000): {} > APPL_INT_VECT
.syscall LOAD (ROMADDR (.applexctbl) + SIZEOF (.applexctbl)): {}
> APPL_INT_VECT
```

If the program contains an absolute code section, a section which contains object code that must not be copied at startup, the section must also be specified in the `.lcf` file with the `LOAD` directive. Listing 21-11 on page 372 shows example C source code that
generates an interrupt service routine that must be placed at a specific address at runtime. **Listing 21-12 on page 372** shows the linker directives that ensure that this routine's object code will be loaded at a specific address at runtime.

**Listing 21-11. Absolute code example**

```c
#pragma push
#pragma section code_type ".abs.00010000" code_mode=pc_rel
asm void _ISRVectorTable(void)
{
    b InterruptHandler
    nop
    nop
    b InterruptHandler
}
#pragma pop
```

**Listing 21-12. Linker commands for absolute code in ROM**

```c
MEMORY
{
//internal Flash
APPL_INT_VECT      : org= 0x00010000, len= 0x000000FF;
    // If org is changed, make sure to adjust start
    // address in .abs.00010000 LOAD (0x00010000): {} >
    // APPL_INT_VECT accordingly
    // ...
}
SECTIONS {
    .abs.00010000 LOAD (0x00010000): {} > APPL_INT_VECT
}<...>
```

### 21.7.5 Specifying Jump Table Location

By default the CodeWarrior compiler and linker allocate jump tables for `switch` statements in RAM. When the application executes from ROM, it is sometimes better to have the switch table allocated in ROM to reduce RAM requirements.

To tell the compiler and linker to place jump tables in an object code section that will be placed in ROM, use this directive in your C or C++ source code:

```c
#pragma read_only_switch_tables on
```

Alternately, to tell the compiler to generate a branch tree in executable code instead of a jump table in a data section, use this directive in C or C++:

```c
#pragma switch_tables off
```
21.7.6 Specifying Constant Data Location

By default, the CodeWarrior compiler allocates all constant values of a size greater than 8 bytes in the .rodata section. There are two solutions for storing constants with sizes smaller than 8 bytes to be allocated in this section:

Solution 1: Define the variable in section .rodata using the __declspec(section) directive in C or C++ source code. Listing 21-13 on page 373 shows an example.

Listing 21-13. Using __declspec(section) to store small constants in .rodata

```c
#define SMALL_ROM_CONST __declspec(section "\".rodata\")
SMALL_ROM_CONST const unsigned int MyInt2 = 0x4534
```

Solution 2: Enter 0 in the Small Data2 option in the CodeWarrior IDE’s Properties > C/C++ Build > Settings > Tool Settings > PowerPC CPU panel.

21.8 Embedded C++

Embedded C++ (EC++) is a subset of the ISO/IEC 14882:2003 C++ language that is intended to compile into smaller, faster executable code suitable for embedded systems. Embedded C++ source code is upwardly compatible with ISO/IEC C++ source code.

- Activating EC++
- Differences Between ISO C++ and EC++
- EC++ Specifications

21.8.1 Activating EC++

The table below describes the settings used to control Embedded C++ conformance.

<table>
<thead>
<tr>
<th>To control this option from here...</th>
<th>use this setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeWarrior IDE</td>
<td>EC++ Compatibility Mode in the C/C++ Language Settings panel</td>
</tr>
<tr>
<td>source code</td>
<td>#pragma ecplusplus</td>
</tr>
<tr>
<td>command line</td>
<td>-dialect ec++</td>
</tr>
</tbody>
</table>

Table 21-14. Controlling Embedded C++ conformance
To test for EC++ compatibility mode at compile time, use the __embedded_cplusplus predefined symbol.

21.8.2 Differences Between ISO C++ and EC++

The EC++ proposal does not support the following ISO/IEC 14882:2003 C++ features:

- Templates
- Libraries
- File Operations
- Localization
- Exception Handling
- Unsupported Language Features

21.8.2.1 Templates

ISO/IEC C++ specifies templates. The EC++ proposal does not include template support for class or functions.

21.8.2.2 Libraries

The EC++ proposal supports the <string>, <complex>, <ios>, <streambuf>, <istream>, and <ostream> classes, but only in a non-template form. The EC++ specifications do not support any other ISO/IEC C++ libraries, including the STL-type algorithm libraries.

21.8.2.3 File Operations

The EC++ proposal does not support any file operations except simple console input and output file types.

21.8.2.4 Localization
The EC++ proposal does not contain localization libraries because of the excessive memory requirements.

### 21.8.2.5 Exception Handling

The EC++ proposal does not support exception handling.

### 21.8.2.6 Unsupported Language Features

The EC++ proposal does not support the following language features:

- mutable specified
- RTTI
- namespace
- multiple inheritance
- virtual inheritance

### 21.8.3 EC++ Specifications

Topics in this section describe how to design software that adhere to the EC++ proposal:

- Language Related Issues
- Library-Related Issues

#### 21.8.3.1 Language Related Issues

To make sure your source code complies with both ISO/IEC 14882:2003 C++ and EC++ standards, follow these guidelines:

- Do not use RTTI (Run Time Type Identification).
- Do not use exception handling, namespaces, or other unsupported features.
- Do not use multiple or virtual inheritance.
21.8.3.2 Library-Related Issues

Do not refer to routines, data structures, and classes in the Main Standard Library (MSL) for C++.
Chapter 22
Libraries and Support Code

CodeWarrior software includes libraries and support files you can add to your project. This chapter describes these libraries and how to choose among them:

- runtime libraries: support for higher-level C and C++ language features
- board initialization: low-level startup routines

The sections of this chapter are:

- Main Standard Libraries
- Third Party Standard Libraries
- Embedded Warrior Library
- Runtime Libraries
- Board Initialization Code

22.1 Main Standard Libraries

This section explains how to use the Power Architecture version of the Main Standard Libraries (MSL).

- Using the Main Standard Libraries
- Choosing an MSL Library
- Using Console I/O
- Allocating Additional Heap Space

For more information refer to the MSL C Reference and the MSL C++ Reference.
22.1.1 Using the Main Standard Libraries

The Main Standard Libraries (MSL) are a complete, configurable set of C and C++ standard libraries. These libraries also include MSL Extras, which extends the standard library and adds compatibility with common UNIX libraries. All of the source files required to build MSL are included in your CodeWarrior product, along with project files for different MSL configurations.

To use the MSL library, you must also use a runtime library. To support custom hardware capabilities, such as a new memory configuration, make changes to the runtime libraries instead of the MSL library's source files. Then, if necessary, reconfigure and recompile the MSL library. Refer to the MSL C Reference or MSL C++ Reference for more information.

22.1.2 Choosing an MSL Library

If your program uses features in the MSL libraries, you must choose a configuration that matches your software and hardware requirements.

The filenames of the configurations of MSL libraries follow a naming convention to describe each library's capabilities and features. The table below lists the types of MSL configurations.

<table>
<thead>
<tr>
<th>These characters in a filename...</th>
<th>Applicable?</th>
<th>mean that the library has these features...</th>
</tr>
</thead>
<tbody>
<tr>
<td>fdlibm</td>
<td>No</td>
<td>High-level math functions, including as the trigonometric functions.</td>
</tr>
<tr>
<td>MSL_C</td>
<td>Yes</td>
<td>C standard library.</td>
</tr>
<tr>
<td>MSL_C++</td>
<td>No</td>
<td>C++ standard library.</td>
</tr>
<tr>
<td>MSL_EC++</td>
<td>No</td>
<td>Embedded C++ standard library.</td>
</tr>
<tr>
<td>MSL_SUPP_TRK</td>
<td>No</td>
<td>CodeWarrior TRK support.</td>
</tr>
<tr>
<td>MSL_SUPP_UART</td>
<td>No</td>
<td>UART (serial communications).</td>
</tr>
<tr>
<td>.bare</td>
<td>No</td>
<td>Boards with no operating system.</td>
</tr>
<tr>
<td>PPCEABI</td>
<td>Yes</td>
<td>Conforms to the PowerPC Embedded Application Binary Interface (EABI) standard.</td>
</tr>
<tr>
<td>SZ</td>
<td>No</td>
<td>Optimized for size.</td>
</tr>
<tr>
<td>SP</td>
<td>Yes</td>
<td>Single Precision Floating Point only.</td>
</tr>
<tr>
<td>A</td>
<td>No</td>
<td>Altivec™ support.</td>
</tr>
</tbody>
</table>

Table continues on the next page...
### Table 22-1. MSL Library Naming Conventions (continued)

<table>
<thead>
<tr>
<th>These characters in a filename...</th>
<th>Applicable?</th>
<th>mean that the library has these features...</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>No</td>
<td>Code compression.</td>
</tr>
<tr>
<td>E</td>
<td>Yes</td>
<td>e500 and e200z (formerly Zen) targets.</td>
</tr>
<tr>
<td>E 2</td>
<td>No</td>
<td>e500v2 targets, with double-precision floating-point operations.</td>
</tr>
<tr>
<td>H</td>
<td>No</td>
<td>Hardware floating-point operations.</td>
</tr>
<tr>
<td>HC</td>
<td>No</td>
<td>Hardware floating-point operations and code compression.</td>
</tr>
<tr>
<td>S</td>
<td>No</td>
<td>Software emulation of floating-point operations.</td>
</tr>
<tr>
<td>N</td>
<td>Yes</td>
<td>No floating-point support.</td>
</tr>
<tr>
<td>NC</td>
<td>No</td>
<td>No floating-point support, but with code compression.</td>
</tr>
<tr>
<td>LE</td>
<td>No</td>
<td>Little-endian mode.</td>
</tr>
<tr>
<td>UC</td>
<td>Yes</td>
<td>Function parameters declared <code>char</code> are treated as if they were declared <code>unsigned char</code>. Use a UC library in build targets for which the Use Unsigned Chars option is enabled. Use a non-UC library in build targets for which this option is disabled. If the option used by the build target is different from the option used to generate the build target's runtime library, the linker issue a warning.</td>
</tr>
<tr>
<td>V</td>
<td>No</td>
<td>Uses VLE instructions SPFP/SPE floating point operations in software routines. Use only with processors that have an e200z (formerly Zen) core.</td>
</tr>
<tr>
<td>VS</td>
<td>No</td>
<td>Uses VLE instructions, single-precision floating point operations using native processor instructions, and double-precision floating point operations using software routines.</td>
</tr>
<tr>
<td>SC</td>
<td>Yes</td>
<td>Function parameters declared <code>char</code> are treated as if they were declared <code>signed char</code>.</td>
</tr>
</tbody>
</table>

### 22.1.3 Using Console I/O

The default MSL configuration for Power Architecture processors provides the `stdout`, `stderr`, and `stdin` file streams through serial I/O on most evaluation boards. The C++ standard library assigns `cin`, `cout`, and `cerr` to the target board's serial port. Also, I/O functions that refer to the standard streams implicitly, such as `printf()`, are available.
This configuration does not provide disk I/O, so functions such as printf() are not available.

To use the MSL console I/O functions, you must include a special serial I/O library in your project. Your hardware must be initialized properly to work with this library.

22.1.4 Allocating Additional Heap Space

The heap you define using the Heap Address option of the EPPC Linker panel is the default heap. The default heap needs no initialization. The CodeWarrior linker will only link the object code for memory management if your program calls malloc() or new().

You may find that you do not have enough contiguous memory available for your needs. In this case, you can initialize multiple memory pools to form a large heap.

You create each memory pool by calling init_alloc(). You can find an example of this call in __ppc_eabi_init.c and __ppc_eabi_init.cpp. You do not need to initialize the memory pool for the default heap.

22.2 Third Party Standard Libraries

You might be able to use a third-party C standard library with your CodeWarrior tools. To determine if the CodeWarrior tools will generate object code that is compatible with a third-party library, compare the file stdarg.h from the third-party library with stdarg.h from the MSL library. The CodeWarrior C/C++ compiler for Power Architecture processors uses the MSL files stdarg.h and runtime library file __va_arg.c to generate variable-length parameter functions. Your third-party library must be compatible with these files.

You cannot use a third-party standard C++ library with your CodeWarrior product.

22.3 Embedded Warrior Library

This section explains how to use the Power Architecture version of the Embedded Warrior Library (EWL).

- Using the Embedded Warrior Libraries
• EWL Naming Convention
• How to Rebuild the EWL Libraries

For more information refer to the *EWL C Reference* and the *EWL C++ Reference*.

## 22.3.1 Using the Embedded Warrior Libraries

Embedded Warrior Library (EWL) is the next generation of MSL. With this release, EWL will be an alternative library. The sources are based on MSL and are more MISRA compliant. Existing standard prefix file name, library (archive) names have been modified. some of the legacy libraries have been deprecated and a greater number of processor core specific libraries are introduced.

**NOTE**
EWL is not supported by all products. All of your existing projects and makefile access paths will not use the EWL unless you specifically select it.

## 22.3.2 EWL Naming Convention

Each archive name has 3 pieces: prefix, core and flags. Following are the details of each piece:

• The prefix is one of the following:
  • libm_ - mean that the library has math features
  • librt_ - mean that the library has runtime features
  • libc_ - mean that the library has reduced code size C features
  • libc99_ - mean that the library has faster and increased C99 conformant C features
  • libstdc++_ - mean that the library has latest C++ features
  • libc++_ - mean that the library has reduced code size C++ features
• The core starts with the processor family (like e200 or e500) and optionally ends with the core name (such as z750).
• Current flags are VLE, Soft (software floating point) and SPFP_Only (math library only has single precision sources and source file doubles are treated as if they are single precision). SPFP_Only is only used with e200 and e500 which have single precision floating point instructions but no double precision instructions.
NOTE

EWL can only build unsigned char libraries. CodeWarrior no longer provides signed char library in EWL as it is not compliant with EABI. If required, users must build their own signed char libraries.

The Prefix Name used in EWL also differs from that of the MSL Prefix Name. For example, the prefix name ansi_prefix.PPCEABI.bare.h in MSL is referred to as ansi_prefix.PA_EABI.bare.h in EWL.

The table below lists the EWL Library Core And Flag Name and its equivalent MSL Suffix Name.

NOTE

Some of the libraries listed in the table below may not be available in the current release.

<table>
<thead>
<tr>
<th>EWL Library Core and Flag Name</th>
<th>Equivalent MSL Suffix Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic_N</td>
<td>PPCEABI.N.UC</td>
</tr>
<tr>
<td>82xx_soft</td>
<td>PPCEABI.S.UC</td>
</tr>
<tr>
<td>E200z0_VLE_Soft</td>
<td>PPCEABI.VS.UC</td>
</tr>
<tr>
<td>E200z150_VLE_Soft</td>
<td>PPCEABI.VS.UC</td>
</tr>
<tr>
<td>E200z335_VLE</td>
<td>PPCEABI.V.UC</td>
</tr>
<tr>
<td>E200z335_VLE_SPFP_Only</td>
<td>PPCEABI.V.SP.UC</td>
</tr>
<tr>
<td>E200z336_VLE</td>
<td>PPCEABI.V.UC</td>
</tr>
<tr>
<td>E200z336_VLE_SPFP_Only</td>
<td>PPCEABI.V.SP.UC</td>
</tr>
<tr>
<td>E200z446_VLE</td>
<td>PPCEABI.V.UC</td>
</tr>
<tr>
<td>E200z446_VLE_SPFP_Only</td>
<td>PPCEABI.V.SP.UC</td>
</tr>
<tr>
<td>E200z448_VLE</td>
<td>PPCEABI.V.UC</td>
</tr>
<tr>
<td>E200z448_VLE_SPFP_Only</td>
<td>PPCEABI.V.SP.UC</td>
</tr>
<tr>
<td>E200z650</td>
<td>PPCEABI.E.UC</td>
</tr>
<tr>
<td>E200z650_SPFP_Only</td>
<td>PPCEABI.E.SP.UC</td>
</tr>
<tr>
<td>E200z650_VLE</td>
<td>PPCEABI.V.UC</td>
</tr>
<tr>
<td>E200z650_VLE_SPFP_Only</td>
<td>PPCEABI.V.SP.UC</td>
</tr>
<tr>
<td>E200z652</td>
<td>PPCEABI.E.UC</td>
</tr>
<tr>
<td>E200z652_SPFP_Only</td>
<td>PPCEABI.E.SP.UC</td>
</tr>
<tr>
<td>E200z652_VLE</td>
<td>PPCEABI.V.UC</td>
</tr>
<tr>
<td>E200z652_VLE_SPFP_Only</td>
<td>PPCEABI.V.SP.UC</td>
</tr>
<tr>
<td>E200z750_VLE</td>
<td>PPCEABI.V.UC</td>
</tr>
<tr>
<td>E200z750_VLE_SPFP_Only</td>
<td>PPCEABI.V.SP.UC</td>
</tr>
<tr>
<td>E200z760_VLE</td>
<td>PPCEABI.V.UC</td>
</tr>
</tbody>
</table>

Table continues on the next page...
Table 22-2. EWL Library Core and Flag Name and its equivalent MSL Suffix Name (continued)

<table>
<thead>
<tr>
<th>EWL Library Core and Flag Name</th>
<th>Equivalent MSL Suffix Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>E200z760_VLE_SPFP_Only</td>
<td>PPCEABI.V.SP.UC</td>
</tr>
<tr>
<td>E300c1</td>
<td>PPCEABI.H.UC</td>
</tr>
<tr>
<td>E300c2</td>
<td>PPCEABI.H.UC</td>
</tr>
<tr>
<td>E300c3</td>
<td>PPCEABI.H.UC</td>
</tr>
<tr>
<td>E300c4</td>
<td>PPCEABI.H.UC</td>
</tr>
<tr>
<td>E500V1</td>
<td>PPCEABI.E.UC</td>
</tr>
<tr>
<td>E500V1_SPFP_Only</td>
<td>PPCEABI.E.SP.UC</td>
</tr>
<tr>
<td>E500V2</td>
<td>PPCEABI.E2.UC</td>
</tr>
<tr>
<td>E600</td>
<td>PPCEABI.A.UC</td>
</tr>
</tbody>
</table>

22.3.3 How to Rebuild the EWL Libraries

The EWL library files are present in the ewl\lib folder. To rebuild the EWL library files, perform the following steps:

NOTE

Ensure that you have access to a make utility within DOS, before rebuilding the EWL libraries.

1. Open a DOS command prompt.
2. Define the CWINSTALL environment variable.
   - If your PA product layout is in the folder C:\Program Files\Freescale\CW PA V10.x then you can define CWINSTALL as
     
     ```
     set CWINSTALL='C:\Program Files\Freescale\CW PA v10.x\PA'
     ```
   - For PA
     
     ```
     cd C:\Program Files\Freescale\CW PA V10.x\PA\PA_Support\ewl
     ```
3. Change your working directory to the ewl folder.
   - For PA
     
     ```
     ```
4. Set OSTYPE environment variable as console to run makefiles on windows command prompt.
   
   ```
   ```
5. Clean the existing library files using the following command:
<CWInstallDir>\gnu\bin\make -f makefile clean PLATFORM=PA TARGETS="libm_XXX libc_XXX libc99_XXX"

**NOTE**

The `make` utility is available in `<CWInstallDir>\gnu\bin` folder, where `<CWInstallDir>` is the CodeWarrior installation directory.

- For example, in PA, the following command will delete only the `libm_E500V2.a`, `libc_E500V2.a`, `libc99_E500V2.a` library files.

  <CWInstallDir>\gnu\bin\make -f makefile clean PLATFORM=PA TARGETS="libm_E500V2 libc_E500V2 libc99_E500V2"

6. Rebuild a C or math or C99 library file using the following command:

  <CWInstallDir>\gnu\bin\make -f EWL_C.PA.mak -C EWL_C TARGETS="libm_XXX libc_XXX libc99_XXX"

- For example, in PA, the following command will build the `libm_E500V2.a`, `libc_E500V2.a`, `libc99_E500V2.a` files:

  <CWInstallDir>\gnu\bin\make -f EWL_C.PA.mak -C EWL_C TARGETS="libm_E500V2 libc_E500V2 libc99_E500V2"

7. Rebuild a C++ or libc++ library file using the following command:

  <CWInstallDir>\gnu\bin\make -f EWL_C++.PA.mak -C EWL_C++ TARGETS="libstdc++_XXX libc++_XXX"

- For example, in PA, the following command will build the `libstdc++_E500V2.a`, `libc++_E500V2.a` files:

  <CWInstallDir>\gnu\bin\make -f EWL_C++.PA.mak -C EWL_C++ TARGETS="libstdc++_E500V2 libc++_E500V2"

8. Rebuild a Runtime library file using the following command:

  <CWInstallDir>\gnu\bin\make -f EWL_Runtime.PA.mak -C EWL_Runtime "librt_XXX"

- For example, in PA, the following command will build the `librt_E500V2.a` file:

  <CWInstallDir>\gnu\bin\make -f EWL_Runtime.PA.mak -C EWL_Runtime "librt_E500V2"

Upon successful execution of the `make` command, check the lib folder for the EWL libraries.
22.4 Runtime Libraries

A runtime library provides low-level functions that support high-level C and C++ language features, such as memory management and file system access. Conceptually, a runtime library acts as an interface between a target system's hardware or operating system and the CodeWarrior C or C++ runtime environment.

This CodeWarrior product includes many runtime libraries and support code files. These files are here:

```
installDir \PowerPC_EABI_Support\Runtime
```

where `installDir` is a placeholder for the path in which you installed your product.

For your projects to build and run, you must include the correct runtime library and startup code. These sections explain how to pick the correct files:

- Required Libraries and Source Code Files
- Allocating Additional Heap Space
- Choosing a Runtime Library

22.4.1 Required Libraries and Source Code Files

Every CodeWarrior project *must* include a runtime library.

Select the library appropriate for your project, given the language your are using (C or C++), the processor on your target board, and your target setting choices. Use the information in Runtime Library Naming Conventions to help you pick the correct library.

The runtime libraries are in this directory:

```
installDir \PowerPC_EABI_Support\Runtime\Lib\n```

Along with the pre-built runtime libraries, this CodeWarrior product includes the source code and project files required to build the runtime libraries. As a result, you can modify them as necessary.

All runtime library source code files are in this directory:

```
installDir\PowerPC_EABI_Support\Runtime\Src
```

The runtime library project files are in this directory:
The project names are `Runtime.PPCEABI.mcp` and `Run_EC++.PPCEABI.mcp`. Each project has a different build target for each configuration of the runtime library.

For more information about customizing the runtime libraries, read the comments in the source code files as well as the runtime library release notes.

**NOTE**

The C and C++ runtime libraries do not initialize hardware. The CodeWarrior tools assume that you load and run the programs linked with these libraries with the CodeWarrior debugger. When your program is ready to run as a standalone application, you must add the required hardware initialization code.

Finally, in addition to a runtime library, every C and C++ project must include one of the startup files listed below. These files contain functions called by the runtime code that you can customize if necessary. One kind of customization is board-specific initialization. For other customization examples, see either of these files:

- `__ppc_eabi_init.c` (for C language projects)
- `__ppc_eabi_init.cpp` (for C++ projects)

### 22.4.2 Allocating Additional Heap Space

If you specify a heap size in the EPPC Target settings panel, the linker creates a default heap of this size. The default heap needs no initialization.

You can create additional heaps by:

- **Defining** `ALLOC_ADDITIONAL_HEAPS` equal to 1 in either `__ppc_eabi_init.c` or `__ppc_eabi_init.cpp`.

  Doing so causes the stub implementation of `AllocMoreHeaps()` to be called by the runtime initialization code.

- **Implementing** the `AllocMoreHeaps()` stub by calling `init_alloc()` as many times as desired.

  Each time `init_alloc()` is called, the heap is expanded by the amount specified.
22.4.3 Choosing a Runtime Library

Substrings embedded in the name of a runtime library indicate the type of support the library provides. Use these substrings to pick the runtime library appropriate for your project. The table below lists and defines the meaning of each library filename substring.

<table>
<thead>
<tr>
<th>Substring</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime</td>
<td>The library is a C language library.</td>
</tr>
<tr>
<td>Run_EC++</td>
<td>The library is an embedded C++ library.</td>
</tr>
<tr>
<td>PPCEABI</td>
<td>The library conforms to the PowerPC Embedded Application Binary Interface (EABI) standard.</td>
</tr>
<tr>
<td>A</td>
<td>The library provides Altivec™ support.</td>
</tr>
<tr>
<td>E</td>
<td>The library is for e500 and e200z (formerly, Zen) targets.</td>
</tr>
<tr>
<td>E.fast</td>
<td>The library is for e500 and e200z (formerly, Zen) targets. Further, this library's floating-point operations are faster than those of an E library, but they do not strictly conform to the IEEE floating-point standard.</td>
</tr>
<tr>
<td>E.2</td>
<td>The library is for e500v2 targets and supports double-precision floating-point operations.</td>
</tr>
<tr>
<td>H</td>
<td>The library supports hardware floating-point operations.</td>
</tr>
<tr>
<td>HC</td>
<td>The library supports hardware floating-point operations and code compression.</td>
</tr>
<tr>
<td>S</td>
<td>The library provides software emulation of floating-point operations.</td>
</tr>
<tr>
<td>SP</td>
<td>Single Precision Floating Point only.</td>
</tr>
<tr>
<td>N</td>
<td>The library provides no floating-point support.</td>
</tr>
<tr>
<td>NC</td>
<td>The library provides no floating-point support, but supports code compression.</td>
</tr>
<tr>
<td>LE</td>
<td>The library is for a processor running in little-endian mode.</td>
</tr>
<tr>
<td>UC</td>
<td>The library was built with the Use Unsigned Chars option of the C++ Language target settings panel enabled. As a result, all library function parameters declared char are treated as if they were declared unsigned char. Use a UC library in build targets for which the Use Unsigned Chars option is enabled. Use a non- UC library in build targets for which this option is disabled. If the option used by the build target is different from the option used to generate the build target's runtime library, the linker issue a warning.</td>
</tr>
<tr>
<td>V</td>
<td>The library's functions:</td>
</tr>
<tr>
<td></td>
<td>• Contain VLE instructions.</td>
</tr>
<tr>
<td></td>
<td>• Perform single-precision floating point operations using the core's SPE auxiliary processing unit (APU).</td>
</tr>
<tr>
<td></td>
<td>• Perform double-precision floating using software routines.</td>
</tr>
<tr>
<td></td>
<td>Use only with processors that have an SPE APU.</td>
</tr>
</tbody>
</table>

Table continues on the next page...
Table 22-3. Run time Library Naming Conventions (continued)

<table>
<thead>
<tr>
<th>Substring</th>
<th>Meaning</th>
</tr>
</thead>
</table>
| VS        | The library's functions:  
• Contain VLE instructions.  
• Perform all floating-point operations using software routines  
Use only with processors that have an e200z (formerly Zen) core. |

22.5 Board Initialization Code

For each supported board, CodeWarrior for Power Architecture Processors includes a hardware initialization routine. Each routine is in a source code file whose name reflects the board for which the routine is designed. These files are in this directory:

```
installDir\PowerPC_EABI_Support\Runtime\Src
```

The initialization routines are in source code form, so you can modify them to work with different configurations of a board or with a different board.

If you run your program under control of the CodeWarrior debugger, the program must not perform hardware initialization because the debugger performs the required board initialization.

However, if your program is running standalone (that is, without the debugger), the program may need to execute hardware initialization code. The easiest way to include this code in your program is to add the appropriate board initialization file to your project.

Each board initialization file includes a function named `usr_init()`. This function performs the required hardware initialization for your board. `usr_init()` is conditionally called by the `__init_hardware()` function in `ppc_eabi_init.c` (or in `ppc_eabi_init.cpp`, if you are using C++) . The startup code always calls `__init_hardware()`.

The default implementation of the `__init_hardware()` function calls `usr_init()` if either the `ROM_VERSION` or `CACHE_VERSION` preprocessor constant is defined. (See Listing 22-1 on page 388.) However, you can change the implementation of `__init_hardware()` to suit your project's requirements.

Listing 22-1. Code Showing call of `usr_init()` in `__init_hardware()`
asm void __init_hardware(void) 
{
    /*
     * If not running with the CodeWarrior debugger, initialize the
     * board. Define the preprocessor symbols for the initialization
     * your program requires. You may need to perform other
     * initializations.
     */
    nofralloc

    /* ... code omitted */

    #if defined(ROM_VERSION) || defined(CACHE_VERSION)
        mflr r31      /* save off return address in NV reg */
        bl  usr_init   /* init board hardware */

        mtlr r31      /* get saved return address */
    #endif

    blr
}

To get you program to perform hardware initialization when run outside the CodeWarrior debugger, follow these steps:

1. Add the appropriate board initialization file to your project.
2. Change the preprocessor symbol that conditionalizes the `usr_init()` call in 
    `__init_hardware()` to a symbol that makes sense for your project.
3. Define this symbol in the prefix file for each build target for which you want to run
    the hardware initialization code.
Chapter 23  
Declaration Specifications

Declaration specifications describe special properties to associate with a function or variable at compile time. You insert these specifications in the object's declaration.

- Syntax for Declaration Specifications
- Syntax for Attribute Specifications

23.1 Syntax for Declaration Specifications

The syntax for a declaration specification is

\[
\text{\_declspec(}
\text{spec [}
\text{options ])}
\text{function-declaration ;}
\]

where \text{spec} is the declaration specification, \text{options} represents possible arguments for the declaration specification, and \text{function-declaration} represents the declaration of the function. Unless otherwise specified in the declaration specification's description, a function's definition does not require a declaration specification.

23.1.1 \_declspec(never\_inline)

Specifies that a function must not be inlined.

Syntax

\[
\text{\_declspec (never\_inline)}
\text{function\_prototype ;}
\]
Remarks
Declaring a function's prototype with this declaration specification tells the compiler not to inline the function, even if the function is later defined with the `inline`, `__inline__`, or `__inline` keywords.

23.2 Syntax for Attribute Specifications

The syntax for an attribute specification is

```
__attribute__((
    list-of-attributes ))
```

where `list-of-attributes` is a comma-separated list of zero or more attributes to associate with the object. Place an attribute specification at the end of the declaration and definition of a function, function parameter, or variable. Listing 23-1 on page 392 shows an example.

Listing 23-1. Example of an attribute specification

```c
int f(int x __attribute__((unused))) __attribute__((never_inline));

int f(int x __attribute__((unused))) __attribute__((never_inline))
{
    return 20;
}
```

23.2.1 `__attribute__((noreturn))`

Specifies that a function will never return if it is called. This allows the compiler to make certain modifications to the control flow graph to improve the code generation. An example of an improvement would be to not conserve the LR or non-volatiles on the stack.

23.2.2 `__attribute__((deprecated))`

Specifies that the compiler must issue a warning when a program refers to an object.
variable-declaration __attribute__((deprecated));

variable-definition __attribute__((deprecated));

function-declaration __attribute__((deprecated));

function-definition __attribute__((deprecated));

Remarks

This attribute instructs the compiler to issue a warning when a program refers to a function or variable. Use this attribute to discourage programmers from using functions and variables that are obsolete or will soon be obsolete.

Listing 23-2. Example of deprecated attribute

```c
int velocipede(int speed) __attribute__((deprecated));
int bicycle(int speed);

int f(int speed)
{
    return velocipede(speed); /* Warning. */
}

int g(int speed)
{
    return bicycle(speed * 2); /* OK */
}
```

23.2.3 __attribute__((force_export))

Prevents a function or static variable from being dead-stripped.

Syntax

```c
function-declaration __attribute__((force_export));

function-definition __attribute__((force_export));

variable-declaration __attribute__((force_export));

variable-definition __attribute__((force_export));
```
Remarks

This attribute specifies that the linker must not dead-strip a function or static variable even if the linker determines that the rest of the program does not refer to the object.

23.2.4  __attribute__((malloc))

Specifies that the pointers returned by a function will not point to objects that are already referred to by other variables.

Syntax

\[
\text{function-declaration} \quad \text{__attribute__((malloc))}; \\
\text{function-definition} \quad \text{__attribute__((malloc))};
\]

Remarks

This attribute specification gives the compiler extra knowledge about pointer aliasing so that it can apply stronger optimizations to the object code it generates.

23.2.5  __attribute__((noalias))

Prevents access of data object through an indirect pointer access.

Syntax

\[
\text{function-parameter} \quad \text{__attribute__((noalias))}; \\
\text{variable-declaration} \quad \text{__attribute__((noalias))}; \\
\text{variable-definition} \quad \text{__attribute__((noalias))};
\]

Remarks
This attribute specifies to the compiler that a data object is only accessed directly, helping the optimizer to generate a better code. The sample code in Listing 23-3 on page 395 will not return a correct result if \( \text{ip} \) is pointed to \( a \).

### Listing 23-3. Example of the noalias attribute

```c
extern int a __attribute__((noalias));
int f(int *ip)
{
    a = 1;
    *ip = 0;
    return a; // optimized to return 1;
}
```

#### 23.2.6 __attribute__((returns_twice))

Specifies that a function may return more than one time because of multithreaded or non-linear execution.

**Syntax**

```c
function-declaration __attribute__((returns_twice));

function-definition __attribute__((returns_twice));
```

**Remarks**

This attribute specifies to the compiler that the program's flow of execution might enter and leave a function without explicit function calls and returns. For example, the standard library's `setjmp()` function allows a program to change its execution flow arbitrarily.

With this information, the compiler limits optimizations that require explicit program flow.

#### 23.2.7 __attribute__((unused))

Specifies that the programmer is aware that a variable or function parameter is not referred to.

**Syntax**

```c
function-parameter __attribute__((unused));
```
variable-declaration __attribute__((unused));

variable-definition __attribute__((unused));

Remarks

This attribute specifies that the compiler should not issue a warning for an object if the object is not referred to. This attribute specification has no effect if the compiler's unused warning setting is off.

Listing 23-4. Example of the unused attribute

void f(int a __attribute__((unused))) /* No warning for a. */
{  
   int b __attribute__((unused)); /* No warning for b. */
   int c; /* Possible warning for c. */

   return 20;
}

23.2.8 __attribute__((used))

Prevents a function or static variable from being dead-stripped.

Syntax

function-declaration __attribute__((used));

function-definition __attribute__((used));

variable-declaration __attribute__((used));

variable-definition __attribute__((used));

Remarks

This attribute specifies that the linker must not dead-strip a function or static variable even if the linker determines that the rest of the program does not refer to the object.
23.2.9  __attribute__((never_inline))

Specifies that a function must not be inlined.

Syntax

__attribute__((never_inline)) function_prototype ;

Remarks

Declaring a function's prototype with this attribute tells the compiler not to inline the function, even if the function is later defined with the inline, __inline__, or __inline keywords.
Chapter 24
Declaration Specifications for Power Architecture Code

Declaration specifications describe special attributes to associate with a function at compile time. For example, these attributes might change how the compiler translates the function, describe what properties it has, or pass other information to the compiler.

- Syntax for Declaration Specifications
- Declaration Specifications

24.1 Syntax for Declaration Specifications

The syntax for a declaration specification is

```c
__declspec(
    spec [ 
        options ]
) function-prototype ;
```

where `spec` is the declaration specification, `options` represents possible arguments for the declaration specification, and `function-prototype` represents the declaration of the function. Unless otherwise specified in the declaration specification's description, a function's definition does not require a matching declaration specification.

24.2 Declaration Specifications

This section explains declaration specifications supported by the current release.

- `__declspec(do_not_merge)`
- `__declspec(final)`
24.2.1 __declspec(do_not_merge)

Specifies that a function must not be removed from object code during code merging optimization.

Syntax

__declspec(do_not_merge)
function-declaration ;

Remarks

When you declare or define a function with this declaration specification the linker does not consider this function when applying the code merging optimization. This declaration specification ensures that the linker will not remove a function from object code during code merging if another function with identical object code exists.

Use this declaration specification for functions that your program refers to with function pointers.

24.2.2 __declspec(final)

Provides Java-style function override checking.

Syntax

__declspec(final)
declaration

Example

struct A { virtual __declspec(final) void vf1(); }; struct B : A { void vf1(); /* Error : the final function A::vf() is overridden by B::vf() */ };
24.2.3  __declspec(force_export)

Specifies that a function or variable must not be dead-stripped.

Syntax

```c
__declspec(force_export)
function-declaration;
```

```c
__declspec(force_export)
variable-declaration;
```

Remarks

When the linker determines that a function or static variable is not referred to by the rest
of the program, the linker removes the object from the final executable image. The linker
does not remove a static variable or function declared with this specification even if this
object is not referred to.

24.2.4  __declspec(interrupt)

Controls the compilation of object code for interrupt service routines.

Syntax

```c
__declspec (interrupt [ 
  option [ 
    optionN ]]) void
__InterruptHandler__(void);
```

```c
__declspec (interrupt [ 
  option [ 
    optionN ]]) void
__InterruptHandler__(void)
{    /* ... */
}
```

where `option` and `optionN` are zero or more of the following choices:

**NOTE**

If no choice is specified, `save_spe` is enabled.

enable
Enables interrupts while this interrupt handler is executing.

SRR
Saves the appropriate save/restore register (SRR0 or SRR1) on the stack.

DAR
Saves the data address register on the stack.

DSISR
Saves the DSI status register on the stack.

save_fprs
Saves the floating-point registers on the stack.

save_altivec
Saves the Altivec® registers on the stack.

save_spe
Saves the special-purpose embedded registers on the stack. This is the default.

nowarn
Does not issue a warning message if the function being compiled as an interrupt service routine is larger than the processor's interrupt vector area (256 bytes/64 instructions).

noncritical | critical | machine | debug

Specify the type of interrupt service routine to generate. Specifically, the compiler choose a return instruction based on this choice. The default is noncritical.

vle_multiple
Enables generation of new VLE instructions in the interrupt prolog/epilog in C-functions by the compiler. This option is only valid for e200 (Zen) core. Compile error will be returned if this option is used for any other cores.

When enabled, the compiler:

- emits e_lmvgprw and e_stmvgprw if any of [r3,r12] needs to be saved to stack
- emits e_lmvsprw and e_stmvsprw if any of the CR, LR, CTR and XER needs to be saved to stack
- emits e_lmvsvrrw and e_stmvsvrr if any of the SRR0 and SRR1 needs to be saved to stack
• emits e_lmvcsrrw and e_stmvcsrrw if any of the CSRR0 and CSRR1 needs to be saved to stack
• emits e_lmvdsrrw and e_stmvdsrrw if any of the DSRR0 and DSRR1 needs to be saved to stack.

NOTE
Enabling vle_multiple could increase the space being taken up by the stack, especially when saving the volatile gprs.

Remarks
When you declare or define a function with this declaration specification the compiler generates a special prologue and epilogue for the function so that it can respond to a processor interrupt. For convenience, the compiler also marks interrupt functions so that the linker does not dead-strip them.

24.2.5  __declspec(no_linker_opts)

Specifies that the linker must not apply its optimizations to a function.

Syntax
__declspec(no_linker_opts)
  function-declaration

24.2.6  __declspec(section name)

Specifies where to store a variable or function in a section that has been predefined or defined with the #pragma section directive.

Syntax
__declspec(section <section_name>) declaration

__declspec(section <section_name>) definition

Parameters
section_name

Specifies the name of an initialized data section.
NOTE
The name of a section must be enclosed in double-quotes (""").
To use a user defined section, create the section using #pragma
section directive before using __declspec(section
<section_name>).

Remarks
• When you specify an uninitialized section name while declaring or defining a
function or variable with this declaration specification, the compiler generates an
error.

For example, if you use __declspec (section ".bss") extern int my_var;, where .bss is
an uninitialized section you will get a descriptive error. In this case, use __declspec
(section ".data") extern int my_var; as .data is normally paired with .bss and .data is
an initialized section. Assuming the variable you are attaching this __declspec to is
an uninitialized object (which is the case with my_var), the object will go into .bss.

• When you use __declspec on the definition, __declspec on a declaration is ignored.
The listing below shows an example of __declspec used to put data and code objects
into specific sections.

Figure 24-1. Example of a declaration specification

__declspec (section ".init") extern void cache_init1(void);
__declspec (section ".text") extern void cache_init2(void);
extern void cache_init3(void);

void cache_init1(){} // goes into .init if the prototype is visible for
this definition

__declspec (section ".init") void cache_init2(){} // ignores previous
section .text and goes into .init

__declspec (section ".init") void cache_init3(){} // by default the
declaration implies .text but the definition forces it to .init

Predefined sections and default sections
The predefined sections set with an object type become the default section for that type.
The compiler predefines the sections listed in the table below.

Table 24-1. Predefined sections

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Data mode</th>
<th>Code mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>code_type</td>
<td>&quot;.text&quot;</td>
<td>data_mode=far_abs</td>
<td>code_mode=pc_rel</td>
</tr>
<tr>
<td>data_type</td>
<td>&quot;.data&quot;</td>
<td>data_mode=far_abs</td>
<td>code_mode=pc_rel</td>
</tr>
<tr>
<td>const_type</td>
<td>&quot;.rodata&quot;</td>
<td>data_mode=far_abs</td>
<td>code_mode=pc_rel</td>
</tr>
<tr>
<td>sdata_type</td>
<td>&quot;.sdata&quot;</td>
<td>data_mode=sda_rel</td>
<td>code_mode=pc_rel</td>
</tr>
<tr>
<td>sconst_type</td>
<td>&quot;.sdata2&quot;</td>
<td>data_mode=sda_rel</td>
<td>code_mode=pc_rel</td>
</tr>
<tr>
<td></td>
<td>&quot;.sbss2&quot;</td>
<td>data_mode=sda_rel</td>
<td>code_mode=pc_rel</td>
</tr>
<tr>
<td></td>
<td>&quot;.PPC.EMB.sdata0&quot;</td>
<td>data_mode=sda_rel</td>
<td>code_mode=pc_rel</td>
</tr>
<tr>
<td></td>
<td>&quot;.PPC.EMB.sbss0&quot;</td>
<td>data_mode=sda_rel</td>
<td>code_mode=pc_rel</td>
</tr>
</tbody>
</table>
NOTE
The ".PPC.EMB.sdata0" and ".PPC.EMB.sbss0" sections are predefined as an alternative to the sdata_type object type.

24.2.7 __declspec(vle_off)

Forces the compiler to use the regular instruction set instead of the Variable Length Encoded (VLE) instruction set.

Syntax

```c
__declspec (vle_off)
function_prototype ;
```

Remarks
Declaring a function's prototype with this declaration specification tells the compiler to use the regular instruction set defined by the Freescale Book E Implementation Standard (EIS) for Power Architecture processors. The compiler and linker must arrange such instructions more rigidly than VLE instructions, resulting in larger object code. For information on the availability of VLE instructions, refer to your processor's documentation.

This declaration specification overrides other compiler settings for VLE code generation.

24.2.8 __declspec(vle_on)

Forces the compiler to use the VLE (Variable Length Encoded) instruction set for a function.

Syntax

```c
__declspec (vle_on)
function_prototype;
```

Remarks
Declaring a function's prototype with this declaration specification tells the compiler to use the VLE instruction set when generating object code for the function. The compiler and linker can arrange VLE instructions more compactly than regular instructions.
The VLE instruction set extends the regular instruction set defined by the Freescale Book E Implementation Standard (EIS). For information on the availability of VLE instructions, refer to your processor's documentation.

This declaration specification overrides other compiler settings for VLE code generation.

24.2.9 __declspec(weak)

Makes a global code or data definition "weak", so that it can be replaced by another definition, in another translation unit.

Syntax

```
__declspec(weak)
function _declaration;
```

Remarks

This declaration replaces #pragma overload.
Chapter 25
Predefined Macros

The compiler preprocessor has predefined macros (some refer to these as predefined symbols). The compiler simulates variable definitions that describe the compile-time environment and properties of the target processor.

This chapter lists the predefined macros that all CodeWarrior compilers make available.

- __ASSEMBLER__
- __COMPILER__
- __COUNTER__
- __TARGETARCH__
- __VERSION_NUM__
- __VERSION__
- __cplusplus
- __CWBUILD__
- __CWCC__
- __DATE__
- __embedded_cplusplus
- __FILE__
- __func__
- __FUNCTION__
- __ide_target()
- __LINE__
- __MWERKS__
- __optlevel
- __PRETTY_FUNCTION__
- __profile__
- __STDC__
- __TIME__
25.1 __ASSEMBLER__

Preprocessor macro defined by the CodeWarrior assembler only.

**Syntax**

__ASSEMBLER__

**Remarks**

This macro can be used to determine if the tool under which a file is processed is the CodeWarrior assembler.

25.2 __COMPILER__

Preprocessor macro defined by the CodeWarrior compiler only.

**Syntax**

__COMPILER__

**Remarks**

This macro can be used to determine if the tool under which a file is processed is the CodeWarrior compiler.

25.3 __COUNTER__

Preprocessor macro that expands to an integer.

**Syntax**

__COUNTER__

**Remarks**

The compiler defines this macro as an integer that has an initial value of 0 incrementing by 1 every time the macro is used in the translation unit.

The value of this macro is stored in a precompiled header and is restored when the precompiled header is used by a translation unit.
25.4  __TARGETARCH__

Preprocessor macro defined as the eRISC architecture mode.

Syntax

__TARGETARCH__

Remarks

This macro can be used to determine the assembly instructions to use in the eRISC architecture.

25.5  __VERSION_NUM__

Preprocessor macro defined as the CodeWarrior tool version number.

Syntax

__VERSION_NUM__

Remarks

This macro can be used to determine in which tool version we are using.

Both Compiler and Assembler tools support this macro (the value returned from this macro depends on the tool compiling the file in which the macro lies).

This macro is comparable, and returns a numeric comparable value (i.e. version 10.0.0.1 returns the value: 0x10000001).

25.6  __VERSION__

Preprocessor macro defined as the CodeWarrior compiler version string.

Syntax

__VERSION__
25.7 __cplusplus

Preprocessor macro defined if compiling C++ source code.

Syntax
__cplusplus

Remarks
The compiler defines this macro when compiling C++ source code. This macro is undefined otherwise.

25.8 __CWBUILD__

Preprocessor macro defined as the build number of the CodeWarrior compiler.

Syntax
__CWBUILD__

Remarks
The ISO standards do not specify this symbol.

25.9 __CWCC__

Preprocessor macro defined as the version of the CodeWarrior compiler frontend.

Syntax
__CWCC__
Remarks

CodeWarrior compilers issued after 2006 define this macro with the compiler's frontend version. For example, if the compiler frontend version is 4.2.0, the value of \_CWCC\_ is 0x4200.

CodeWarrior compilers issued prior to 2006 used the pre-defined macro \_MWERKS\_. The \_MWERKS\_ predefined macro is still functional as an alias for \_CWCC\_.

The ISO standards do not specify this symbol.

25.10  \_DATE\_

Preprocessor macro defined as the date of compilation.

Syntax

\_DATE\_

Remarks

The compiler defines this macro as a character string representation of the date of compilation. The format of this string is

"Mmm dd yyyy"

where $Mmm$ is the a three-letter abbreviation of the month, $dd$ is the day of the month, and $yyyy$ is the year.

25.11  \_embedded_cplusplus

Defined as 1 when compiling embedded C++ source code, undefined otherwise.

Syntax

\_embedded_cplusplus

Remarks

The compiler defines this macro as 1 when the compiler's settings are configured to restrict the compiler to translate source code that conforms to the Embedded C++ proposed standard. The compiler does not define this macro otherwise.
25.12  __FILE__

Preprocessor macro of the name of the source code file being compiled.

**Syntax**

__FILE__

**Remarks**

The compiler defines this macro as a character string literal value of the name of the file being compiled, or the name specified in the last instance of a #line directive.

25.13  __func__

Predefined variable of the name of the function being compiled.

**Prototype**

```c
static const char __func__[] = "function-name ";
```

**Remarks**

The compiler implicitly defines this variable at the beginning of each function if the function refers to __func__. The character string contained by this array, function-name , is the name of the function being compiled.

This implicit variable is undefined outside of a function body. This variable is also undefined when C99 (ISO/IEC 9899-1999) or GCC (GNU Compiler Collection) extension settings are off.

25.14  __FUNCTION__

Predefined variable of the name of the function being compiled.

**Prototype**
static const char __FUNCTION__[] = "
 function-name ";

**Remarks**

The compiler implicitly defines this variable at the beginning of each function if the function refers to __FUNCTION__. The character string contained by this array, function-name, is the name of the function being compiled.

This implicit variable is undefined outside of a function body.

### 25.15 __ide_target()**

Preprocessor operator for querying the IDE about the active build target.

**Syntax**

```c
__ide_target("target_name ")
```

**target-name**

The name of a build target in the active project in the CodeWarrior IDE.

**Remarks**

Expands to 1 if target_name is the same as the active build target in the CodeWarrior IDE's active project. Expands to 0 otherwise. The ISO standards do not specify this symbol.

### 25.16 __LINE__**

Preprocessor macro of the number of the line of the source code file being compiled.

**Syntax**

```c
__LINE__
```

**Remarks**

The compiler defines this macro as a integer value of the number of the line of the source code file that the compiler is translating. The #line directive also affects the value that this macro expands to.
25.17  __MWERKS__

Deprecated. Preprocessor macro defined as the version of the CodeWarrior compiler.

Syntax

__MWERKS__

Remarks

Replaced by the built-in preprocessor macro __CWCC__.

CodeWarrior compilers issued after 1995 define this macro with the compiler's version. For example, if the compiler version is 4.0, the value of __MWERKS__ is 0x4000.

This macro is defined as 1 if the compiler was issued before the CodeWarrior CW7 that was released in 1995.

The ISO standards do not specify this symbol.

25.18  __optlevel

Optimization level exported as a predefined macro.

Syntax

__optlevel0
__optlevel1
__optlevel2
__optlevel3
__optlevel4

Remarks
Using these macros, user can conditionally compile code for a particular optimization level. The following table lists the level of optimization provided by the \_optlevelx macro.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Optimization Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>_optlevel0</td>
<td>00</td>
</tr>
<tr>
<td>_optlevel1</td>
<td>01</td>
</tr>
<tr>
<td>_optlevel2</td>
<td>02</td>
</tr>
<tr>
<td>_optlevel3</td>
<td>03</td>
</tr>
<tr>
<td>_optlevel4</td>
<td>04</td>
</tr>
</tbody>
</table>

**Table 25-1. Optimization Levels**

**Example**

The listing below shows an example of \_optlevelx macro usage.

**Listing 25-1. Example usage of \_optlevel macro**

```c
int main()
{
    #if __optlevel0
        ...
        // This code compiles only if this code compiled with Optimization level 0
    #elif __optlevel1
        ...
        // This code compiles only if this code compiled with Optimization level 1
    #elif __optlevel2
        ...
        // This code compiles only if this code compiled with Optimization level 2
    #elif __optlevel3
        ...
        // This code compiles only if this code compiled with Optimization level 3
    #elif __optlevel4
        ...
        // This code compiles only if this code compiled with Optimization level 4
    #endif

    // function code
}
```

**25.19 \_PRETTY\_FUNCTION\_**

Predefined variable containing a character string of the "unmangled" name of the C++ function being compiled.

**Syntax**

**Prototype**

```c
static const char \_PRETTY\_FUNCTION\_[] = "
    function-name ";
```
Remarks

The compiler implicitly defines this variable at the beginning of each function if the function refers to `__PRETTY_FUNCTION__`. This name, *function-name*, is the same identifier that appears in source code, not the "mangled" identifier that the compiler and linker use. The C++ compiler "mangles" a function name by appending extra characters to the function's identifier to denote the function's return type and the types of its parameters.

The ISO/IEC 14882:2003 C++ standard does not specify this symbol.

### 25.20 `__profile__`

Preprocessor macro that specifies whether or not the compiler is generating object code for a profiler.

**Syntax**

`__profile__`

**Remarks**

Defined as 1 when generating object code that works with a profiler. Undefined otherwise. The ISO standards does not specify this symbol.

### 25.21 `__STDC__`

Defined as 1 when compiling ISO/IEC Standard C source code, undefined otherwise.

**Syntax**

`__STDC__`

**Remarks**

The compiler defines this macro as 1 when the compiler's settings are configured to restrict the compiler to translate source code that conforms to the ISO/IEC 9899-1990 and ISO/IEC 9899-1999 standards. The compiler does not define this macro otherwise.
25.22  __TIME__

Preprocessor macro defined as a character string representation of the time of compilation.

Syntax

__TIME__

Remarks

The compiler defines this macro as a character string representation of the time of compilation. The format of this string is

"hh : mm : ss"

where hh is a 2-digit hour of the day, mm is a 2-digit minute of the hour, and ss is a 2-digit second of the minute.
Chapter 26
Predefined Macros for Power Architecture Compiler

As listed in the previous chapter, the compiler preprocessor has predefined macros. The macros in the previous chapter are available to all architecture. This chapter describes the predefined macros available specifically for the Power Architecture compiler.

- __ALTIVEC__
- __POWERPC__
- __PPCBROADWAY__
- __PPCGECKO__
- Processor Core Specific Macros

26.1 __ALTIVEC__

Preprocessor macro defined if language extensions for using Power Architecture AltiVec™ instructions are available.

**Syntax**

__ALTIVEC__

**Remarks**

The compiler defines this macro when pragma altivec_model is on. This macro is undefined otherwise.

26.2 __POWERPC__

Preprocessor macro defined to describe the target powerpc processor.
__POWERPC__

Remarks
The compiler defines this macro to describe powerpc processor that the compiler is generating code for.

26.3 __PPCBROADWAY__

Preprocessor macro defined if the compiler is generating object code for the Power Architecture "Broadway" processor.

Syntax
__PPCBROADWAY__

Remarks
The compiler defines this macro as 1 when generating object code for the "Broadway" processor. The __PPCGECKO__ macro is also defined.
This macro is undefined otherwise.

26.4 __PPCGECKO__

Preprocessor macro defined if the compiler is generating object code for the Power Architecture "Gecko" processor.

Syntax
__PPCGECKO__

Remarks
The compiler defines this macro as 1 when generating object code for the "Gecko" processor. This macro is undefined otherwise.
26.5 Processor Core Specific Macros

The compiler defines the macros, discussed in the following table, to describe powerpc processor core that the compiler is generating code for.

<table>
<thead>
<tr>
<th>Processor Core</th>
<th>Compiler Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>401</td>
<td><strong>PPC401</strong></td>
</tr>
<tr>
<td>403</td>
<td><strong>PPC403</strong></td>
</tr>
<tr>
<td>405</td>
<td><strong>PPC405</strong></td>
</tr>
<tr>
<td>505</td>
<td><strong>PPC505</strong></td>
</tr>
<tr>
<td>509</td>
<td><strong>PPC509</strong></td>
</tr>
<tr>
<td>555</td>
<td><strong>PPC555</strong></td>
</tr>
<tr>
<td>56x</td>
<td><strong>PPC56x</strong></td>
</tr>
<tr>
<td>601</td>
<td><strong>PPC601</strong></td>
</tr>
<tr>
<td>602</td>
<td><strong>PPC602</strong></td>
</tr>
<tr>
<td>603</td>
<td><strong>PPC603</strong></td>
</tr>
<tr>
<td>603e</td>
<td><strong>PPC603e</strong></td>
</tr>
<tr>
<td>604</td>
<td><strong>PPC604</strong></td>
</tr>
<tr>
<td>604e</td>
<td><strong>PPC604e</strong></td>
</tr>
<tr>
<td>7400</td>
<td><strong>PPC7400</strong></td>
</tr>
<tr>
<td>740/74x</td>
<td><strong>PPC740</strong></td>
</tr>
<tr>
<td>7450/745x</td>
<td><strong>PPC7450</strong></td>
</tr>
<tr>
<td>74xx</td>
<td><strong>PPC74xx</strong></td>
</tr>
<tr>
<td>750/75x</td>
<td><strong>PPC750</strong></td>
</tr>
<tr>
<td>801</td>
<td><strong>PPC801</strong></td>
</tr>
<tr>
<td>821</td>
<td><strong>PPC821</strong></td>
</tr>
<tr>
<td>823</td>
<td><strong>PPC823</strong></td>
</tr>
<tr>
<td>8240</td>
<td><strong>PPC8240</strong></td>
</tr>
<tr>
<td>82xx</td>
<td><strong>PPC82xx</strong></td>
</tr>
<tr>
<td>824x</td>
<td><strong>PPC8240</strong></td>
</tr>
<tr>
<td>8260</td>
<td><strong>PPC8260</strong></td>
</tr>
<tr>
<td>827x</td>
<td><strong>PPC827x</strong></td>
</tr>
<tr>
<td>850/85x</td>
<td><strong>PPC850</strong></td>
</tr>
<tr>
<td>860/86x</td>
<td><strong>PPC860</strong></td>
</tr>
<tr>
<td>87x</td>
<td><strong>PPC87x</strong></td>
</tr>
<tr>
<td>88x</td>
<td><strong>PPC88x</strong></td>
</tr>
<tr>
<td>8280</td>
<td><strong>PPC8280</strong></td>
</tr>
<tr>
<td>5100</td>
<td><strong>PPC5100</strong></td>
</tr>
<tr>
<td>5200</td>
<td><strong>PPC5200</strong></td>
</tr>
<tr>
<td>e300[c1]</td>
<td><strong>PPCe300</strong></td>
</tr>
</tbody>
</table>

Table continues on the next page...
### Table 26-1. Processor core specific macros (continued)

<table>
<thead>
<tr>
<th>Processor Core</th>
<th>Compiler Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>e300c2</td>
<td><strong>PPCe300c2</strong></td>
</tr>
<tr>
<td>e300c3</td>
<td><strong>PPCe300c3</strong></td>
</tr>
<tr>
<td>e300c4</td>
<td><strong>PPCe300c4</strong></td>
</tr>
<tr>
<td>e500[v1]</td>
<td><strong>PPCe500</strong></td>
</tr>
<tr>
<td>e500v2</td>
<td><strong>PPCe500v2</strong></td>
</tr>
<tr>
<td>e600</td>
<td><strong>PPCe600</strong></td>
</tr>
<tr>
<td>e700</td>
<td><strong>PPCe700</strong></td>
</tr>
<tr>
<td>Zen</td>
<td><strong>PPCZen</strong></td>
</tr>
<tr>
<td>generic</td>
<td><strong>PPCGENERIC</strong></td>
</tr>
</tbody>
</table>
Chapter 27
Using Pragmas

The #pragma preprocessor directive specifies option settings to the compiler to control the compiler and linker's code generation.

- Checking Pragma Settings
- Saving and Restoring Pragma Settings
- Determining Which Settings Are Saved and Restored
- Invalid Pragmas
- Pragma Scope

27.1 Checking Pragma Settings

The preprocessor function __option() returns the state of pragma settings at compile-time. The syntax is

```c
__option( setting-name )
```

where setting-name is the name of a pragma that accepts the on, off, and reset arguments.

If setting-name is on, __option(setting-name) returns 1. If setting-name is off, __option(setting-name) returns 0. If setting-name is not the name of a pragma, __option(setting-name) returns false. If setting-name is the name of a pragma that does not accept the on, off, and reset arguments, the compiler issues a warning message.

Listing 27-1 on page 423 shows an example.

Listing 27-1. Using the __option() preprocessor function

```c
#if __option(ANSI_strict)
#include "portable.h" /* Use the portable declarations. */
#else
```

CodeWarrior Development Studio for Power Architecture Processors Build Tools Reference Manual, Rev. 10.x, 06/2015

Freescale Semiconductor, Inc.
27.2 Saving and Restoring Pragma Settings

There are some occasions when you would like to apply pragma settings to a piece of source code independently from the settings in the rest of the source file. For example, a function might require unique optimization settings that should not be used in the rest of the function's source file.

Remembering which pragmas to save and restore is tedious and error-prone. Fortunately, the compiler has mechanisms that save and restore pragma settings at compile time. Pragma settings may be saved and restored at two levels:

- all pragma settings
- some individual pragma settings

Settings may be saved at one point in a compilation unit (a source code file and the files that it includes), changed, then restored later in the same compilation unit. Pragma settings cannot be saved in one source code file then restored in another unless both source code files are included in the same compilation unit.

Pragmas `push` and `pop` save and restore, respectively, most pragma settings in a compilation unit. Pragmas `push` and `pop` may be nested to unlimited depth. Listing 27-2 on page 424 shows an example.

Listing 27-2. Using push and pop to save and restore pragma settings

```c
/* Settings for this file. */
#pragma opt_unroll_loops on
#pragma optimize_for_size off
void fast_func_A(void)
{
    /* ... */
}

/* Settings for slow_func(). */
#pragma push /* Save file settings. */
#pragma optimization_size 0
void slow_func(void)
{
    /* ... */
#pragma pop /* Restore file settings. */

void fast_func_B(void)
{
    /* ... */
}
```
Pragmas that accept the `reset` argument perform the same actions as pragmas `push` and `pop`, but apply to a single pragma. A pragma's `on` and `off` arguments save the pragma's current setting before changing it to the new setting. A pragma's `reset` argument restores the pragma's setting. The `on`, `off`, and `reset` arguments may be nested to an unlimited depth. Listing 27-3 on page 425 shows an example.

Listing 27-3. Using the `reset` option to save and restore a pragma setting

```c
/* Setting for this file. */
#pragma opt_unroll_loops on

void fast_func_A(void)
{
    /* ... */
}

/* Setting for smallslowfunc(). */
#pragma opt_unroll_loops off
void small_func(void)
{
    /* ... */
}

/* Restore previous setting. */
#pragma opt_unroll_loops reset

void fast_func_B(void)
{
    /* ... */
}
```

27.3 Determining Which Settings Are Saved and Restored

Not all pragma settings are saved and restored by pragmas `push` and `pop`. Pragmas that do not change compiler settings are not affected by `push` and `pop`. For example, pragma `message` cannot be saved and restored.

Listing 27-4 on page 425 shows an example that checks if the `ANSI_strict` pragma setting is saved and restored by pragmas `push` and `pop`.

Listing 27-4. Testing if pragmas `push` and `pop` save and restore a setting

```c
/* Preprocess this source code. */
#pragma ANSI_strict on
#pragma push
#pragma ANSI_strict off
#pragma pop
#if __option(ANSI_strict)
#error "Saved and restored by push and pop."
#else
#error "Not affected by push and pop."
#endif
```
27.4 Invalid Pragmas

If you enable the compiler's setting for reporting invalid pragmas, the compiler issues a warning when it encounters a pragma it does not recognize. For example, the pragma statements in the listing below, generates warning with the invalid pragmas setting enabled.

Listing 27-5. Invalid Pragmas

```c
#pragma silly_data off       // WARNING: silly_data is not a pragma.
#pragma ANSI_strict select  // WARNING: select is not defined
#pragma ANSI_strict on      // OK
```

The table below lists various options on how to control the recognition of invalid pragmas.

Table 27-1. Controlling invalid pragmas

<table>
<thead>
<tr>
<th>To control this option from here...</th>
<th>use this setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeWarrior IDE</td>
<td>Illegal Pragmas in the C/C++ PowerPC Compiler &gt; Warnings panel</td>
</tr>
<tr>
<td>source code</td>
<td>#pragma warn_illpragma</td>
</tr>
<tr>
<td>command line</td>
<td>-warnings illpragmas</td>
</tr>
</tbody>
</table>

27.5Pragma Scope

The scope of a pragma setting is limited to a compilation unit (a source code file and the files that it includes).

At the beginning of compilation unit, the compiler uses its default settings. The compiler then uses the settings specified by the CodeWarrior IDE's build target or in command-line options.

The compiler uses the setting in a pragma beginning at the pragma's location in the compilation unit. The compilers continues using this setting:

- until another instance of the same pragma appears later in the source code
- until an instance of pragma `pop` appears later in the source code
- until the compiler finishes translating the compilation unit
Chapter 28
Pragma for Standard C Conformance

28.1 ANSI_strict

Controls the use of non-standard language features.

Syntax

#pragma ANSI_strict on | off | reset

Remarks

If you enable the pragma ANSI_strict, the compiler generates an error message if it encounters some CodeWarrior extensions to the C language defined by the ISO/IEC 9899-1990 ("C90") standard:

- C++-style comments
- unnamed arguments in function definitions
- non-standard keywords

By default, this pragma is off.

28.2 c99

Controls the use of a subset of ISO/IEC 9899-1999 ("C99") language features.

Syntax

#pragma c99 on | off | reset

Remarks
If you enable this pragma, the compiler accepts many of the language features described by the ISO/IEC 9899-1999 standard:

- More rigid type checking.
- Trailing commas in enumerations.
- GCC/C99-style compound literal values.
- Designated initializers.
- __func__ predefined symbol.
- Implicit return 0; in main().
- Non-const static data initializations.
- Variable argument macros (__VA_ARGS__).
- bool and_Bool support.
- long long support (separate switch).
- restrict support.
- // comments.
- inline support.
- Digraphs.
- _Complex and _Imaginary (treated as keywords but not supported).
- Empty arrays as last struct members.
- Designated initializers
- Hexadecimal floating-point constants.
- Variable length arrays are supported within local or function prototype scope (as required by the C99 standard).
- Unsuffixed decimal constant rules.
- ++bool-- expressions.
- (T) (int-list) are handled/parsed as cast-expressions and as literals.
- __STDC_HOSTED__ is 1.

By default, this pragma is disabled.

28.3 c9x

Equivalent to #pragma c99.

28.4 ignore_oldstyle

Controls the recognition of function declarations that follow the syntax conventions used before ISO/IEC standard C (in other words, "K&R" style).
**Syntax**

```
#pragma ignore_oldstyle on | off | reset
```

**Remarks**

If you enable this pragma, the compiler ignores old-style function declarations and lets you prototype a function any way you want. In old-style declarations, you specify the types of arguments on separate lines instead of the function's argument list. For example, the code in Listing 28-1 on page 429 defines a prototype for a function with an old-style definition.

**Listing 28-1. Mixing Old-style and Prototype Function Declarations**

```c
int f(char x, short y, float z);
#pragma ignore_oldstyle on
f(x, y, z)
char x;
short y;
float z;
{ return (int)x+y+z;
}
#pragma ignore_oldstyle reset
```

This pragma does not correspond to any panel setting. By default, this setting is disabled.

### 28.5 **only_std_keywords**

Controls the use of ISO/IEC keywords.

**Syntax**

```
#pragma only_std_keywords on | off | reset
```

**Remarks**

The compiler recognizes additional reserved keywords. If you are writing source code that must follow the ISO/IEC C standards strictly, enable the pragma `only_std_keywords`.

By default, this pragma is disabled.
28.6 require_prototypes

Controls whether or not the compiler should expect function prototypes.

Syntax

```
#pragma require_prototypes on | off | reset
```

Remarks

This pragma only affects non-static functions.

If you enable this pragma, the compiler generates an error message if you use a function that does not have a preceding prototype. Use this pragma to prevent error messages caused by referring to a function before you define it. For example, without a function prototype, you might pass data of the wrong type. As a result, your code might not work as you expect even though it compiles without error.

In **Listing 28-2 on page 430**, function `main()` calls `PrintNum()` with an integer argument even though `PrintNum()` takes an argument of type `float`.

```
#include <stdio.h>

void main(void)
{
    PrintNum(1);  /* PrintNum() tries to interpret the integer as a float. Prints 0.000000. */
}

void PrintNum(float x)
{
    printf("%f\n", x);
}
```

When you run this program, you could get this result:

```
0.000000
```

Although the compiler does not complain about the type mismatch, the function does not give the result you intended. Since `PrintNum()` does not have a prototype, the compiler does not know to generate instructions to convert the integer to a floating-point number before calling `PrintNum()` . Consequently, the function interprets the bits it received as a floating-point number and prints nonsense.
A prototype for `PrintNum()` as in Listing 28-3 on page 431, gives the compiler sufficient information about the function to generate instructions to properly convert its argument to a floating-point number. The function prints what you expected.

**Listing 28-3. Using a Prototype to Avoid Type-mismatch**

```c
#include <stdio.h>

void PrintNum(float x); /* Function prototype. */

void main(void)
{
    PrintNum(1);        /* Compiler converts int to float. */
    /* Prints 1.000000. */
}

void PrintNum(float x)
{
    printf("%f\n", x);
}
```

In other situations where automatic conversion is not possible, the compiler generates an error message if an argument does not match the data type required by a function prototype. Such a mismatched data type error is easier to locate at compile time than at runtime.
Chapter 29
Pragmas for C++

29.1 access_errors

Controls whether or not to change invalid access errors to warnings.

Syntax

#pragma access_errors on | off | reset

Remarks

If you enable this pragma, the compiler issues an error message instead of a warning when it detects invalid access to protected or private class members.

This pragma does not correspond to any IDE panel setting. By default, this pragma is on.

29.2 always_inline

Controls the use of inlined functions.

Syntax

#pragma always_inline on | off | reset

Remarks

This pragma is deprecated. We recommend that you use the inline_depth() pragma instead.
29.3  arg_dep_lookup

Controls C++ argument-dependent name lookup.

Syntax

#pragma arg_dep_lookup on | off | reset

Remarks

If you enable this pragma, the C++ compiler uses argument-dependent name lookup. This pragma does not correspond to any IDE panel setting. By default, this setting is on.

29.4  ARM_conform

This pragma is no longer available. Use ARM_scoping instead.

29.5  ARM_scoping

Controls the scope of variables declared in the expression parts of if, while, do, and for statements.

Syntax

#pragma ARM_scoping on | off | reset

Remarks

If you enable this pragma, any variables you define in the conditional expression of an if, while, do, or for statement remain in scope until the end of the block that contains the statement. Otherwise, the variables only remain in scope until the end of that statement. Listing 29-1 on page 434 shows an example.

This pragma corresponds to the Legacy for-scoping setting in the CodeWarrior IDE’s Properties > C/C++ Build > Settings > Tool Settings > PowerPC Compiler > C/C++ Language settings panel. By default, this pragma is off.

Listing 29-1. Example of Using Variables Declared in for Statement

CodeWarrior Development Studio for Power Architecture Processors Build Tools Reference Manual, Rev. 10.x, 06/2015

Freescale Semiconductor, Inc.
for(int i=1; i<1000; i++) { /* . . . */ }  
return i;  // OK if ARM_scoping is on, error if ARM_scoping is off.

29.6 array_new_delete

Enables the operator new[] and delete[] in array allocation and deallocation operations, respectively.

Syntax

#pragma array_new_delete on | off | reset

Remarks

By default, this pragma is on.

29.7 auto_inline

Controls which functions to inline.

Syntax

#pragma auto_inline on | off | reset

Remarks

If you enable this pragma, the compiler automatically chooses functions to inline for you, in addition to functions declared with the inline keyword.

Note that if you enable either the Do not Inline setting or the dont_inline pragma, the compiler ignores the setting of the auto_inline pragma and does not inline any functions.

This pragma corresponds to the Auto-Inline setting in the CodeWarrior IDE's Properties > C/C++ Build > Settings > Tool Settings > PowerPC Compiler > C/C++ Language settings panel. By default, this pragma is disabled.

29.8 bool

Determines whether or not bool, true, and false are treated as keywords in C++ source code.
Syntax

#pragma bool on | off | reset

Remarks

If you enable this pragma, you can use the standard C++ bool type to represent true and false. Disable this pragma if bool, true, or false are defined in your source code.

Enabling the bool data type and its true and false values is not equivalent to defining them in source code with typedef, enum, or #define. The C++ bool type is a distinct type defined by the ISO/IEC 14882:2003 C++ Standard. Source code that does not treat bool as a distinct type might not compile properly.

This pragma corresponds to the Enable bool Support setting in the CodeWarrior IDE's Properties > C/C++ Build > Settings > Tool Settings > PowerPC Compiler > C/C++ Language settings panel. By default, this setting is on.

29.9 cplusplus

Controls whether or not to translate subsequent source code as C or C++ source code.

Syntax

#pragma cplusplus on | off | reset

Remarks

If you enable this pragma, the compiler translates the source code that follows as C++ code. Otherwise, the compiler uses the suffix of the filename to determine how to compile it. If a file name ends in .c, .h, or .pch, the compiler automatically compiles it as C code, otherwise as C++. Use this pragma only if a file contains both C and C++ code.

NOTE

The CodeWarrior C/C++ compilers do not distinguish between uppercase and lowercase letters in file names and file name extensions except on UNIX-based systems.

This pragma corresponds to the Force C++ Compilation setting in the CodeWarrior IDE's Properties > C/C++ Build > Settings > Tool Settings > PowerPC Compiler > C/C++ Language settings panel. By default, this pragma is disabled.
29.10  cpp1x

Controls whether or not to enable support to experimental features made available in the 1x version of C++ standard.

Syntax

#pragma cpp1x on | off | reset

Remarks

If you enable this pragma, you can use the following extensions to the 1x or 05 version of the C++ standard that would otherwise be invalid:

- Enables support for __alignof__.
- Enables support for __decltype__, which is a reference type preserving typeof.
- Enables support for nullptr.
- Enables support to allow >> to terminate nested template argument lists.
- Enables support for static_assert.

NOTE

This pragma enables support to experimental and unvalidated implementations of features that may or may not be available in the final version of the C++ standard. The features should not be used for critical or production code.

29.11  cpp_extensions


Syntax

#pragma cpp_extensions on | off | reset

Remarks

If you enable this pragma, you can use the following extensions to the ISO/IEC 14882:2003 C++ standard that would otherwise be invalid:
• Anonymous `struct` & `union` objects. Listing 29-2 on page 438 shows an example.

Listing 29-2. Example of Anonymous `struct` & `union` Objects

```c
#pragma cpp_extensions on
void func() {
    union {
        long hilo;
        struct { short hi, lo; }; // anonymous struct
    };
    hi=0x1234;
    lo=0x5678; // hilo==0x12345678
}
```

• Unqualified pointer to a member function. Listing 29-3 on page 438 shows an example.

Listing 29-3. Example of an Unqualified Pointer to a Member Function

```c
#pragma cpp_extensions on
struct RecA { void f(); } void RecA::f() {
    void (RecA::*ptmf1)() = &RecA::f; // ALWAYS OK
    void (RecA::*ptmf2)() = f; // OK if you enable cpp_extensions.
}
```

• Inclusion of `const` data in precompiled headers.

By default, this pragma is disabled.

### 29.12 `debuginline`

Controls whether the compiler emits debugging information for expanded inline function calls.

**Syntax**

```c
#pragma debuginline on | off | reset
```

**Remarks**

If the compiler emits debugging information for inline function calls, then the debugger can step to the body of the inlined function. This behavior more closely resembles the debugging experience for un-inlined code.

**NOTE**

Since the actual "call" and "return" instructions are no longer present when stepping through inline code, the debugger will immediately jump to the body of an inlined function and
"return" before reaching the return statement for the function. Thus, the debugging experience of inlined functions may not be as smooth as debugging un-inlined code.

This pragma does not correspond to any panel setting. By default, this pragma is on.

29.13 def_inherited

Controls the use of inherited.

Syntax

#pragma def_inherited on | off | reset

Remarks

The use of this pragma is deprecated. It lets you use the non-standard inherited symbol in C++ programming by implicitly adding
typedef base inherited;
as the first member in classes with a single base class.

NOTE

The ISO/IEC 14882:2003 C++ standard does not support the inherited symbol. Only the CodeWarrior C++ language implements the inherited symbol for single inheritance.

By default, this pragma is off.

29.14 defer_codegen

Obsolete pragma. Replaced by interprocedural analysis options. See Interprocedural Analysis.

29.15 defer_defarg_parsing

Defers the parsing of default arguments in member functions.
Syntax

#pragma defer_defarg_parsing on | off

Remarks

To be accepted as valid, some default expressions with template arguments will require additional parenthesis. For example, Listing 29-4 on page 440 results in an error message.

Listing 29-4. Deferring parsing of default arguments

template<typename T,typename U> struct X { T t; U u; };

struct Y {
  // The following line is not accepted, and generates
  // an error message with defer_defarg_parsing on.
  void f(X<int,int> = X<int,int>());
};

Listing 29-5 on page 440 does not generate an error message.

Listing 29-5. Correct default argument deferral

template<typename T,typename U> struct X { T t; U u; };

struct Y {
  // The following line is OK if the default
  // argument is parenthesized.
  void f(X<int,int> = (X<int,int>()));
};

This pragma does not correspond to any panel setting. By default, this pragma is on.

29.16 direct_destruction

This pragma is obsolete. It is no longer available.

29.17 direct_to_som

This pragma is obsolete. It is no longer available.

29.18 dont_inline
Controls the generation of inline functions.

Syntax

#pragma dont_inline on | off | reset

Remarks

If you enable this pragma, the compiler does not inline any function calls, even those declared with the `inline` keyword or within a class declaration. Also, it does not automatically inline functions, regardless of the setting of the `auto_inline` pragma, described in "auto_inline" on page 195. If you disable this pragma, the compiler expands all inline function calls, within the limits you set through other inlining-related pragmas.

This pragma corresponds to the **Do not Inline** setting in the CodeWarrior IDE's C/C++ Language settings panel. By default, this pragma is **off**.

### 29.19 ecplusplus

Controls the use of embedded C++ features.

Syntax

#pragma ecplusplus on | off | reset

Remarks

If you enable this pragma, the C++ compiler disables the non-EC++ features of ISO/IEC 14882:2003 C++ such as templates, multiple inheritance, and so on.

This pragma corresponds to the **EC++ Compatibility Mode** setting in the CodeWarrior IDE's Properties > C/C++ Build > Settings > Tool Settings > PowerPC Compiler> C/C++ Language settings panel. By default, this pragma is **off**.

### 29.20 exceptions

Controls the availability of C++ exception handling.

Syntax

#pragma exceptions on | off | reset
Remarks

If you enable this pragma, you can use the try and catch statements in C++ to perform exception handling. If your program does not use exception handling, disable this setting to make your program smaller.

You can throw exceptions across any code compiled by the CodeWarrior C/C++ compiler with #pragma exceptions on.

You cannot throw exceptions across libraries compiled with #pragma exceptions off. If you throw an exception across such a library, the code calls terminate() and exits.

This pragma corresponds to the Enable C++ Exceptions setting in the CodeWarrior IDE's Properties > C/C++ Build > Settings > Tool Settings > PowerPC Compiler > C/C++ Language settings panel. By default, this pragma is on.

29.21 inline_bottom_up

Controls the bottom-up function inlining method.

Syntax

#pragma inline_bottom_up on | off | reset

Remarks

Bottom-up function inlining tries to expand up to eight levels of inline leaf functions. The maximum size of an expanded inline function and the caller of an inline function can be controlled by the pragmas shown in Listing 29-6 on page 442 and Listing 29-7 on page 442.

Listing 29-6. Maximum Complexity of an Inlined Function

// Maximum complexity of an inlined function
#pragma inline_max_size( max ) // default max == 256

Listing 29-7. Maximum Complexity of a Function that Calls Inlined Functions

// Maximum complexity of a function that calls inlined functions
#pragma inline_max_total_size( max ) // default max == 10000

where max loosely corresponds to the number of instructions in a function.
If you enable this pragma, the compiler calculates inline depth from the last function in the call chain up to the first function that starts the call chain. The number of functions the compiler inlines from the bottom depends on the values of `inline_depth`, `inline_max_size`, and `inline_max_total_size`. This method generates faster and smaller source code for some (but not all) programs with many nested inline function calls.

If you disable this pragma, top-down inlining is selected, and the `inline_depth` setting determines the limits for top-down inlining. The `inline_max_size` and `inline_max_total_size` pragmas do not affect the compiler in top-down mode.

### 29.22 inline_bottom_up_once

Performs a single bottom-up function inlining operation.

**Syntax**

```plaintext
#pragma inline_bottom_up_once on | off | reset
```

**Remarks**

By default, this pragma is off.

### 29.23 inline_depth

Controls how many passes are used to expand inline function calls.

**Syntax**

```plaintext
#pragma inline_depth(n)
```

```plaintext
#pragma inline_depth(smart)
```

**Parameters**

- `n`

Sets the number of passes used to expand inline function calls. The number `n` is an integer from 0 to 1024 or the `smart` specifier. It also represents the distance allowed in the call chain from the last function up. For example, if \( d \) is the total depth of a call chain, then functions below a depth of \( d - n \) are inlined if they do not exceed the following size settings:
The first pragma sets the maximum function size to be considered for inlining; the second sets the maximum size to which a function is allowed to grow after the functions it calls are inlined. Here, \( n \) is the number of statements, operands, and operators in the function, which turns out to be roughly twice the number of instructions generated by the function. However, this number can vary from function to function. For the `inline_max_size` pragma, the default value of \( n \) is 256; for the `inline_max_total_size` pragma, the default value of \( n \) is 10000.

smart

The `smart` specifier is the default mode, with four passes where the passes 2-4 are limited to small inline functions. All inlineable functions are expanded if `inline_depth` is set to 1-1024.

**Remarks**

The pragmas `dont_inline` and `always_inline` override this pragma. This pragma corresponds to the **Inline Depth** setting in the CodeWarrior IDE’s **C/C++ Language** settings panel. Setting the Inline Depth option to "Do not Inline" in the settings panel will also override this pragma. By default, this pragma is disabled.

### 29.24 inline_max_auto_size

Determines the maximum complexity for an auto-inlined function.

**Syntax**

```c
#pragma inline_max_auto_size ( complex )
```

**Parameters**

*complex*

The `complex` value is an approximation of the number of statements in a function, the current default value is 15. Selecting a higher value will inline more functions, but can lead to excessive code bloat.

**Remarks**

This pragma does not correspond to any panel setting.
29.25 inline_max_size

Sets the maximum number of statements, operands, and operators used to consider the function for inlining.

Syntax

#pragma inline_max_size ( size )

Parameters

size

The maximum number of statements, operands, and operators in the function to consider it for inlining, up to a maximum of 256.

Remarks

This pragma does not correspond to any panel setting.

29.26 inline_max_total_size

Sets the maximum total size a function can grow to when the function it calls is inlined.

Syntax

#pragma inline_max_total_size ( max_size )

Parameters

max_size

The maximum number of statements, operands, and operators the inlined function calls that are also inlined, up to a maximum of 7000.

Remarks

This pragma does not correspond to any panel setting.

29.27 internal
Controls the internalization of data or functions.

**Syntax**

```c
#pragma internal on | off | reset
```

```c
#pragma internal list name1 [, name2 ]*
```

**Remarks**

When using the `#pragma internal on` format, all data and functions are automatically internalized.

Use the `#pragma internal list` format to tag specific data or functions for internalization. It applies to all names if it is used on an overloaded function. You cannot use this pragma for C++ member functions or static class members.

**Listing 29-8 on page 446** shows an example:

**Listing 29-8. Example of an Internalized List**

```c
extern int f(), g;
#pragma internal list f,g
```

This pragma does not correspond to any panel setting. By default, this pragma is disabled.

## 29.28 iso_templates

Controls whether or not to use the new parser supported by the CodeWarrior 2.5 C++ compiler and issue warning messages for missing typenames.

**Syntax**

```c
#pragma iso_templates on | off | reset
```

**Remarks**

This pragma combines the functionality of pragmas `parse_func_templ`, `parse_mfunc_templ` and `warn_no_typename`.

This pragma ensures that your C++ source code is compiled using the newest version of the parser, which is stricter than earlier versions. The compiler issues a warning message if a typenames required by the C++ standard is missing but can still be determined by the compiler based on the context of the surrounding C++ syntax.

By default, this pragma is on.
29.29  new_mangler

Controls the inclusion or exclusion of a template instance's function return type to the mangled name of the instance.

Syntax

#pragma new_mangler on | off | reset

Remarks

The C++ standard requires that the function return type of a template instance to be included in the mangled name, which can cause incompatibilities. Enabling this pragma within a prefix file resolves those incompatibilities.

This pragma does not correspond to any panel setting. By default, this pragma is on.

29.30  no_conststringconv


Syntax

#pragma no_conststringconv on | off | reset

Remarks

When enabled, the compiler generates an error message when it encounters an implicit const string conversion.

Listing 29-9. Example of const string conversion

#pragma no_conststringconv on
char *cp = "Hello World"; /* Generates an error message. */

This pragma does not correspond to any panel setting. By default, this pragma is off.

29.31  no_static_dtors
Controls the generation of static destructors in C++.

Syntax

#pragma no_static_dtors on | off | reset

Remarks

If you enable this pragma, the compiler does not generate destructor calls for static data objects. Use this pragma to generate smaller object code for C++ programs that never exit (and consequently never need to call destructors for static objects).

This pragma does not correspond to any panel setting. By default, this setting is disabled.

29.32 nosyminline

Controls whether debug information is gathered for inline/template functions.

Syntax

#pragma nosyminline on | off | reset

Remarks

When on, debug information is not gathered for inline/template functions.

This pragma does not correspond to any panel setting. By default, this pragma is disabled.

29.33 old_friend_lookup

Implements non-standard C++ friend declaration behavior that allows friend declarations to be visible in the enclosing scope.

#pragma old_friend_lookup on | off | reset

Example

This example shows friend declarations that are invalid without #pragma old_friend_lookup.

Listing 29-10. Valid and invalid declarations without #pragma old_friend_lookup
class C2;
void f2();
struct S {
    friend class C1;
    friend class C2;
    friend void f1();
    friend void f2();
};
C1 *cp1;    // error, C1 is not visible without namespace declaration
C2 *cp2;    // OK
int main()
{
    f1();  // error, f1() is not visible without namespace declaration
    f2();  // OK
}

### 29.34 old_pods

Permits non-standard handling of classes, structs, and unions containing pointer-to-pointer members

**Syntax**

```
#pragma old_pods on | off | reset
```

**Remarks**

According to the ISO/IEC 14882:2003 C++ Standard, classes/structs/unions that contain pointer-to-pointer members are now considered to be plain old data (POD) types.

This pragma can be used to get the old behavior.

### 29.35 old_vtable

This pragma is no longer available.
29.36  **opt_classresults**

Controls the omission of the copy constructor call for class return types if all return statements in a function return the same local class object.

**Syntax**

```
#pragma opt_classresults on | off | reset
```

**Remarks**

*Listing 29-11 on page 450* shows an example.

**Listing 29-11. Example #pragma opt_classresults**

```c
#pragma opt_classresults on
struct X {
    X();
    X(const X&);
    // ...
};
X f() {
    X x; // Object x will be constructed in function result buffer.
    // ...
    return x; // Copy constructor is not called.
}
```

This pragma does not correspond to any panel setting. By default, this pragma is on.

29.37  **parse_functempl**

Controls whether or not to use the new parser supported by the CodeWarrior 2.5 C++ compiler.

**Syntax**

```
#pragma parse_functempl on | off | reset
```

**Remarks**

If you enable this pragma, your C++ source code is compiled using the newest version of the parser, which is stricter than earlier versions.
This option actually corresponds to the ISO C++ Template Parser option (together with pragmas parse_functempl and warn_no_typename). By default, this pragma is disabled.

29.38 parse_mfunctempl

Controls whether or not to use the new parser supported by the CodeWarrior 2.5 C++ compiler for member function bodies.

Syntax

#pragma parse_mfunctempl on | off | reset

Remarks

If you enable this pragma, member function bodies within your C++ source code is compiled using the newest version of the parser, which is stricter than earlier versions.

This pragma does not correspond to any panel setting. By default, this pragma is disabled.

29.39 RTTI

Controls the availability of runtime type information.

Syntax

#pragma RTTI on | off | reset

Remarks

If you enable this pragma, you can use runtime type information (or RTTI) features such as dynamic_cast and typeid. The other RTTI expressions are available even if you disable the Enable RTTI setting. Note that *type_info::before(const type_info&) is not implemented.

This pragma corresponds to the Enable RTTI setting in the CodeWarrior IDE's Properties > C/C++ Build > Settings > Tool Settings > PowerPC Compiler > C/C++ Language settings panel.
29.40  suppress_init_code

Controls the suppression of static initialization object code.

Syntax

#pragma suppress_init_code on | off | reset

Remarks

If you enable this pragma, the compiler does not generate any code for static data initialization such as C++ constructors.

NOTE
Using this pragma because it can produce erratic or unpredictable behavior in your program.

This pragma does not correspond to any panel setting. By default, this pragma is disabled.

29.41  template_depth

Controls how many nested or recursive class templates you can instantiate.

#pragma template_depth(n)

Remarks

This pragma lets you increase the number of nested or recursive class template instantiations allowed. By default, n equals 64; it can be set from 1 to 30000. You should always use the default value unless you receive the error message

template too complex or recursive

This pragma does not correspond to any panel setting.

29.42  thread_safe_init
Controls the addition of extra code in the binary to ensure that multiple threads cannot enter a static local initialization at the same time.

**Syntax**

```
#pragma thread_safe_init on | off | reset
```

**Remarks**

A C++ program that uses multiple threads and static local initializations introduces the possibility of contention over which thread initializes static local variable first. When the `pragma` is `on`, the compiler inserts calls to mutex functions around each static local initialization to avoid this problem. The C++ runtime library provides these mutex functions.

**Listing 29-12. Static local initialization example**

```c
int func(void) {
    // There may be synchronization problems if this function is
    // called by multiple threads.
    static int countdown = 20;

    return countdown--;  
}
```

**NOTE**

This pragma requires runtime library functions which may not be implemented on all platforms, due to the possible need for operating system support.

**Listing 29-13 on page 453** shows another example.

**Listing 29-13. Example thread_safe_init**

```
#pragma thread_safe_init on

void thread_heavy_func() {
    // Multiple threads can now safely call this function:
    // the static local variable will be constructed only once.
    static std::string localstring = thread_unsafe_func();
}
```

**NOTE**

When an exception is thrown from a static local initializer, the initializer is retried by the next client that enters the scope of the local.

This pragma does not correspond to any panel setting. By default, this pragma is off.
29.43 warn_hidevirtual

Controls the recognition of a non-virtual member function that hides a virtual function in a superclass.

Syntax

```
#pragma warn_hidevirtual on | off | reset
```

Remarks

If you enable this pragma, the compiler issues a warning message if you declare a non-virtual member function that hides a virtual function in a superclass. One function hides another if it has the same name but a different argument type. Listing 29-14 on page 454 shows an example.

Listing 29-14. Hidden Virtual Functions

```cpp
class A {
    public:
        virtual void f(int);
        virtual void g(int);
};

class B: public A {
    public:
        void f(char);        // WARNING: Hides A::f(int)
        virtual void g(int); // OK: Overrides A::g(int)
};
```

The ISO/IEC 14882:2003 C++ Standard does not require this pragma.

NOTE

A warning message normally indicates that the pragma name is not recognized, but an error indicates either a syntax problem or that the pragma is not valid in the given context.

This pragma corresponds to the Hidden Virtual Functions setting in the CodeWarrior IDE's Properties > C/C++ Build > Settings > Tool Settings > PowerPC Compiler > Warnings C/C++ Language settings panel.

29.44 warn_no_explicit_virtual

Controls the issuing of warning messages if an overriding function is not declared with a virtual keyword.
Syntax

#pragma warn_no_explicit_virtual on | off | reset

Remarks

Listing 29-15 on page 455 shows an example.

Listing 29-15. Example of warn_no_explicit_virtual pragma

#pragma warn_no_explicit_virtual on
struct A {
    virtual void f();
};
struct B {
    void f();
    // WARNING: override B::f() is declared without virtual keyword
}

Tip

This warning message is not required by the ISO/IEC 14882:2003 C++ standard, but can help you track down unwanted overrides.

This pragma does not correspond to any panel setting. By default, this pragma is off.

29.45 warn_no_typename

Controls the issuing of warning messages for missing typenames.

Syntax

#pragma warn_no_typename on | off | reset

Remarks

The compiler issues a warning message if a typenames required by the C++ standard is missing but can still be determined by the compiler based on the context of the surrounding C++ syntax.

This pragma does not correspond to any panel setting. This pragma is enabled by the ISO/IEC 14882:2003 C++ template parser.
29.46  warn_notinlined

Controls the issuing of warning messages for functions the compiler cannot inline.

Syntax

#pragma warn_notinlined on | off | reset

Remarks

The compiler issues a warning message for non-inlined inline (i.e., on those indicated by the `inline` keyword or in line in a class declaration) function calls.

This pragma corresponds to the **Non-Inlined Functions** setting in the CodeWarrior IDE's **Properties > C/C++ Build > Settings > Tool Settings > PowerPC Compiler > Warnings C/C++ Warnings** settings panel. By default, this pragma is disabled.

29.47  warn_structclass

Controls the issuing of warning messages for the inconsistent use of the `class` and `struct` keywords.

Syntax

#pragma warn_structclass on | off | reset

Remarks

If you enable this pragma, the compiler issues a warning message if you use the `class` and `struct` keywords in the definition and declaration of the same identifier.

Listing 29-16. Inconsistent use of `class` and `struct`

```c
class X;
struct X { int a; }; // WARNING
```

Use this warning when using static or dynamic libraries to link with object code produced by another C++ compiler that distinguishes between class and structure variables in its name "mangling."
This pragma corresponds to the **Inconsistent `class' / `struct' Usage** setting in the CodeWarrior IDE's **Properties > C/C++ Build > Settings > Tool Settings > PowerPC Compiler > Warnings C/C++ Warnings** settings panel. By default, this pragma is disabled.

### 29.48 wchar_type

Controls the availability of the wchar_t data type in C++ source code.

**Syntax**

```
#pragma wchar_type on | off | reset
```

**Remarks**

If you enable this pragma, `wchar_t` is treated as a built-in type. Otherwise, the compiler does not recognize this type.

This pragma corresponds to the **Enable wchar_t Support** setting in the CodeWarrior IDE's **Properties > C/C++ Build > Settings > Tool Settings > PowerPC Compiler > C/C++ Language** settings panel. By default, this pragma is enabled.
Chapter 30
Pragmas for Language Translation

30.1 asmpoundcomment

Controls whether the "#" symbol is treated as a comment character in inline assembly.

Syntax

#pragma asmpoundcomment on | off | reset

Remarks

Some targets may have additional comment characters, and may treat these characters as comments even when

#pragma asmpoundcomment off

is used.

Using this pragma may interfere with the function-level inline assembly language.

This pragma does not correspond to any panel setting. By default, this pragma is on.

30.2 asmsemicolcomment

Controls whether the " ; " symbol is treated as a comment character in inline assembly.

Syntax

#pragma asmsemicolcomment on | off | reset

Remarks
Some targets may have additional comment characters, and may treat these characters as comments even when

```plaintext
#pragma asmsemicolcomment off
```

is used.

Using this pragma may interfere with the assembly language of a specific target.

This pragma does not correspond to any panel setting. By default, this pragma is on.

### 30.3 const_strings

Controls the `const`-ness of character string literals.

**Syntax**

```plaintext
#pragma const_strings [ on | off | reset ]
```

**Remarks**

If you enable this pragma, the type of string literals is an array `const char[n]`, or `const wchar_t[n]` for wide strings, where `n` is the length of the string literal plus 1 for a terminating NULL character. Otherwise, the type `char[n]` or `wchar_t[n]` is used.

By default, this pragma is on when compiling C++ source code and off when compiling C source code.

### 30.4 dollar_identifiers

Controls use of dollar signs ($) in identifiers.

**Syntax**

```plaintext
#pragma dollar_identifiers on | off | reset
```

**Remarks**

If you enable this pragma, the compiler accepts dollar signs ($) in identifiers. Otherwise, the compiler issues an error if it encounters anything but underscores, alphabetic, numeric character, and universal characters (\uxxxx, \Uxxxxxxxx) in an identifier.

This pragma does not correspond to any panel setting. By default, this pragma is off.
30.5 gcc_extensions

Controls the acceptance of GNU C language extensions.

Syntax

#pragma gcc_extensions on | off | reset

Remarks

If you enable this pragma, the compiler accepts GNU C extensions in C source code. This includes the following non-ANSI C extensions:

- Initialization of automatic `struct` or `array` variables with non-`const` values.
- Illegal pointer conversions
- `sizeof( void ) == 1`
- `sizeof( function-type ) == 1`
- Limited support for GCC statements and declarations within expressions.
- Macro redefinitions without a previous `#undef`.
- The GCC keyword `typeof`
- Function pointer arithmetic supported
- `void*` arithmetic supported
- Void expressions in return statements of void
- `__builtin_constant_p (expr)` supported
- Forward declarations of arrays of incomplete type
- Forward declarations of empty static arrays
- Pre-C99 designated initializer syntax (deprecated)
- shortened conditional expression ( `c ?: y` )
- `long __builtin_expect (long exp, long c)` now accepted

By default, this pragma is disabled.

30.6 mark

Adds an item to the Function pop-up menu in the IDE editor.

Syntax

#pragma mark

itemName
### Remarks

This pragma adds *itemName* to the source file's **Function** pop-up menu. If you open the file in the CodeWarrior Editor and select the item from the **Function** pop-up menu, the editor brings you to the pragma. Note that if the pragma is inside a function definition, the item does not appear in the **Function** pop-up menu.

If *itemName* begins with " -- ", a menu separator appears in the IDE's **Function** pop-up menu:

```
#pragma mark --
```

This pragma does not correspond to any panel setting.

### 30.7  mpwc_newline

Controls the use of newline character convention.

**Syntax**

```
#pragma mpwc_newline on | off | reset
```

**Remarks**

If you enable this pragma, the compiler translates `\n` as a Carriage Return (0x0D) and `\r` as a Line Feed (0x0A). Otherwise, the compiler uses the ISO standard conventions for these characters.

If you enable this pragma, use ISO standard libraries that were compiled when this pragma was enabled.

If you enable this pragma and use the standard ISO standard libraries, your program will not read and write `\n` and `\r` properly. For example, printing `\n` brings your program's output to the beginning of the current line instead of inserting a newline.

This pragma does not correspond to any IDE panel setting. By default, this pragma is disabled.

### 30.8  mpwc_relax

Controls the compatibility of the `char*` and `unsigned char*` types.
Syntax

#pragma mpwc_relax on | off | reset

Remarks

If you enable this pragma, the compiler treats `char*` and `unsigned char*` as the same type. Use this setting to compile source code written before the ISO C standards. Old source code frequently uses these types interchangeably.

This setting has no effect on C++ source code.

**NOTE**

Turning this option on may prevent the compiler from detecting some programming errors. We recommend not turning on this option.

Listing 30-1 on page 463 shows how to use this pragma to relax function pointer checking.

**Listing 30-1. Relaxing function pointer checking**

```c
#pragma mpwc_relax on
extern void f(char *);

/* Normally an error, but allowed. */
extern void(*fp1)(void *) = &f;

/* Normally an error, but allowed. */
extern void(*fp2)(unsigned char *) = &f;
```

This pragma does not correspond to any panel setting. By default, this pragma is disabled.

### 30.9 multibyteaware

Controls how the **Source encoding** option in the IDE is treated

Syntax

#pragma multibyteaware on | off | reset

Remarks

This pragma is deprecated. See `#pragma text_encoding` for more details.

By default, this pragma is `off`.
### 30.10 multibyteaware_preserve_literals

Controls the treatment of multibyte character sequences in narrow character string literals.

**Syntax**

```
#pragma multibyteaware_preserve_literals on | off | reset
```

**Remarks**

This pragma does not correspond to any panel setting. By default, this pragma is on.

### 30.11 text_encoding

Identifies the character encoding of source files.

**Syntax**

```
#pragma text_encoding ( "name" | unknown | reset [, global] )
```

**Parameters**

- **name**

The IANA or MIME encoding name or an OS-specific string that identifies the text encoding. The compiler recognizes these names and maps them to its internal decoders:

```
  system US-ASCII ASCII ANSI_X3.4-1968
  ANSI_X3.4-1968 ANSI_X3.4 UTF-8 UTF8 ISO-2022-JP
  CSISO2022JP ISO2022JP CSSHIFTJIS SHIFT-JIS
  SHIFT_JIS SJIS EUC-JP EUCJP UCS-2 UCS-2BE
  UCS-2LE UCS2 UCS2BE UCS2LE UTF-16 UTF-16BE
  UTF-16LE UTF16 UTF16BE UTF16LE UCS-4 UCS-4BE
  UCS-4LE UCS4 UCS4BE UCS4LE 10646-1:1993
  ISO-10646-1 ISO-10646 unicode
```
global

Tells the compiler that the current and all subsequent files use the same text encoding. By default, text encoding is effective only to the end of the file.

Remarks

By default, #pragma text_encoding is only effective through the end of file. To affect the default text encoding assumed for the current and all subsequent files, supply the "global" modifier.

By default, this setting is ASCII.

30.12 trigraphs

Controls the use trigraph sequences specified in the ISO standards.

Syntax

#pragma trigraphs on | off | reset

Remarks

If you are writing code that must strictly adhere to the ANSI standard, enable this pragma.

<table>
<thead>
<tr>
<th>Trigraph</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>??=</td>
<td>#</td>
</tr>
<tr>
<td>??/</td>
<td>\</td>
</tr>
<tr>
<td>??'</td>
<td>^</td>
</tr>
<tr>
<td>??(</td>
<td>[</td>
</tr>
<tr>
<td>??)</td>
<td>]</td>
</tr>
<tr>
<td>??!</td>
<td></td>
</tr>
<tr>
<td>??&lt;</td>
<td>{</td>
</tr>
<tr>
<td>??&gt;</td>
<td>}</td>
</tr>
<tr>
<td>??-</td>
<td>~</td>
</tr>
</tbody>
</table>

NOTE

Use of this pragma may cause a portability problem for some targets.
Be careful when initializing strings or multi-character constants that contain question marks.

**Listing 30-2. Example of Pragma trigraphs**

```c
char c = '????'; /* ERROR: Trigraph sequence expands to '??^ */
char d = '\\?\\?\\?\?'; /* OK */
```

By default, this pragma is disabled.

### 30.13 unsigned_char

Controls whether or not declarations of type `char` are treated as `unsigned char`.

**Syntax**

```
#pragma unsigned_char on | off | reset
```

**Remarks**

If you enable this pragma, the compiler treats a `char` declaration as if it were an `unsigned char` declaration.

**NOTE**

If you enable this pragma, your code might not be compatible with libraries that were compiled when the pragma was disabled. In particular, your code might not work with the ISO standard libraries included with CodeWarrior.

By default, this setting is disabled.
Chapter 31
Pragmas for Diagnostic Messages

31.1 extended_errorcheck

Controls the issuing of warning messages for possible unintended logical errors.

Syntax

#pragma extended_errorcheck on | off | reset

Remarks

If you enable this pragma, the compiler generates a warning message (not an error) if it encounters some common programming errors:

It also issues a warning message when it encounters a delete operator for a class or structure that has not been defined yet. The listing below shows an example.

Listing 31-1. Attempting to delete an undefined structure

```c
#pragma extended_errorcheck on
struct X;
int func(X *xp)
{
    delete xp;  // Warning: deleting incomplete type X
}
```

- An integer or floating-point value assigned to an enum type. The listing below shows an example.

Listing 31-2. Assigning to an Enumerated Type

```c
enum Day { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday } d;

d = 5;    /* WARNING */
d = Monday; /* OK */
d = (Day)3;/* OK */
```
• An empty `return` statement in a function that is not declared `void`. For example, a non-void function with an empty return statement, listed below, results in a warning message.

**Listing 31-3. A non-void function with an empty return statement**

```c
int MyInit(void)
{
    int err = GetMyResources();
    if (err != -1)
    {
        err = GetMoreResources();
    }
    return; /* WARNING: empty return statement */
}
```

A non-void function with a proper return statement, listed below, shows how to prevent this warning message.

**Listing 31-4. A non-void function with a proper return statement**

```c
int MyInit(void)
{
    int err = GetMyResources();
    if (err != -1)
    {
        err = GetMoreResources();
    }
    return err; /* OK */
}
```

By default, this setting is `off`.

### 31.2 maxerrorcount

Limits the number of error messages emitted while compiling a single file.

**Syntax**

```
#pragma maxerrorcount(
    num | off )
```

**Parameters**

- `num`
  Specifies the maximum number of error messages issued per source file.

- `off`
  Does not limit the number of error messages issued per source file.

**Remarks**
The total number of error messages emitted may include one final message:

Too many errors emitted

This pragma does not correspond to any panel setting. By default, this pragma is off.

### 31.3 message

Tells the compiler to issue a text message to the user.

**Syntax**

```c
#pragma message(
    msg  
)
```

**Parameter**

*msg*

Actual message to issue. Does not have to be a string literal.

**Remarks**

In the CodeWarrior IDE, the message appears in the Errors & Warnings window Console view. On the command line, the message is sent to the standard error stream.

This pragma does not correspond to any panel setting.

### 31.4 showmessagenumber

Controls the appearance of warning or error numbers in displayed messages.

**Syntax**

```c
#pragma showmessagenumber on | off | reset
```

**Remarks**

When enabled, this pragma causes messages to appear with their numbers visible. You can then use the `warning` pragma with a warning number to suppress the appearance of specific warning messages.

This pragma does not correspond to any panel setting. By default, this pragma is off.
31.5 show_error_filestack

Controls the appearance of the current `# include` file stack within error messages occurring inside deeply-included files.

Syntax

```plaintext
#pragma show_error_filestack on | off | reset
```

Remarks

This pragma does not correspond to any panel setting. By default, this pragma is **on**.

31.6 suppress_warnings

Controls the issuing of warning messages.

Syntax

```plaintext
#pragma suppress_warnings on | off | reset
```

Remarks

If you enable this pragma, the compiler does not generate warning messages, including those that are enabled.

This pragma does not correspond to any panel setting. By default, this pragma is **off**.

31.7 sym

Controls the generation of debugger symbol information for subsequent functions.

Syntax

```plaintext
#pragma sym on | off | reset
```

Remarks
The compiler pays attention to this pragma only if you enable the debug marker for a file in the IDE project window. If you disable this pragma, the compiler does not put debugging information into the source file debugger symbol file (SYM or DWARF) for the functions that follow.

The compiler always generates a debugger symbol file for a source file that has a debug diamond next to it in the IDE project window. This pragma changes only which functions have information in that symbol file.

This pragma does not correspond to any panel setting. By default, this pragma is enabled.

### 31.8 unused

Controls the suppression of warning messages for variables and parameters that are not referenced in a function.

**Syntax**

```c
#pragma unused ( 
    var_name [, 
    var_name ]... )
```

`var_name`

The name of a variable.

**Remarks**

This pragma suppresses the compile time warning messages for the unused variables and parameters specified in its argument list. You can use this pragma only within a function body. The listed variables must be within the scope of the function.

In C++, you cannot use this pragma with functions defined within a class definition or with template functions.

#### Listing 31-5. Example of Pragma unused() in C

```c
#pragma warn_unusedvar on
#pragma warn_unusedarg on

static void ff(int a)
{
    int b;
#pragma unused(a,b)
    /* Compiler does not warn that a and b are unused. */
}
```

#### Listing 31-6. Example of Pragma unused() in C++
#pragma warn_unusedvar on
#pragma warn_unusedarg on

static void ff(int /* No warning */) { int b;
  #pragma unused(b)
  /* Compiler does not warn that b is unused. */
}

This pragma does not correspond to any CodeWarrior IDE panel setting.

### 31.9 warning

Controls which warning numbers are displayed during compiling.

**Syntax**

```
#pragma warning on | off | reset (num [, ...])
```

This alternate syntax is allowed but ignored (message numbers do not match):

```
#pragma warning
{
  warning_type : warning_num_list [, warning_type :
    warning_num_list , ...])
```

**Parameters**

- `num`
  The number of the warning message to show or suppress.

- `warning_type`
  Specifies one of the following settings:

  - default
  - disable
  - enable

- `warning_num_list`
  The `warning_num_list` is a list of warning numbers separated by spaces.

**Remarks**
Use the pragma `showmessagenumber` to display warning messages with their warning numbers.

This pragma only applies to CodeWarrior front-end warnings. Using the pragma for the Power Architecture back-end warnings returns invalid message number warning.

The CodeWarrior compiler allows, but ignores, the alternative syntax for compatibility with Microsoft® compilers.

This pragma does not correspond to any panel setting. By default, this pragma is on.

### 31.10 warning_errors

Controls whether or not warnings are treated as errors.

**Syntax**

```
#pragma warning_errors on | off | reset
```

**Remarks**

If you enable this pragma, the compiler treats all warning messages as though they were errors and does not translate your file until you resolve them.

### 31.11 warn_any_ptr_int_conv

Controls if the compiler generates a warning message when an integral type is explicitly converted to a pointer type or vice versa.

**Syntax**

```
#pragma warn_any_ptr_int_conv on | off | reset
```

**Remarks**

This pragma is useful to identify potential 64-bit pointer portability issues. An example is shown in.

#### Listing 31-7. Example of warn_any_ptr_int_conv

```
#pragma warn_ptr_int_conv on
short i, *ip
```
**warn_emptydecl**

```c
void func() {
    i = (short)ip;
    /* WARNING: short type is not large enough to hold pointer. */
}
```

```c
#pragma warn_any_ptr_int_conv on
void bar() {
    i = (int)ip; /* WARNING: pointer to integral conversion. */
    ip = (short *)i; /* WARNING: integral to pointer conversion. */
}
```

**Remarks**

By default, this pragma is **off**.

### 31.12 warn_emptydecl

Controls the recognition of declarations without variables.

**Syntax**

```c
#pragma warn_emptydecl on | off | reset
```

**Remarks**

If you enable this pragma, the compiler displays a warning message when it encounters a declaration with no variables.

**Listing 31-8. Examples of empty declarations in C and C++**

```c
#pragma warn_emptydecl on
int ; /* WARNING: empty variable declaration. */
int i; /* OK */
long j; /* WARNING */
long j; /* OK */
```

**Listing 31-9. Example of empty declaration in C++**

```c
#pragma warn_emptydecl on
extern "C" {
}; /* WARNING */
```

By default, this pragma is disabled.

### 31.13 warn_extracomma

Controls the recognition of superfluous commas in enumerations.
Syntax

#pragma warn_extracomma on | off | reset

Remarks

If you enable this pragma, the compiler issues a warning message when it encounters a trailing comma in enumerations. For example, Listing 31-10 on page 475 is acceptable source code but generates a warning message when you enable this setting.

Listing 31-10. Warning about extra commas

#pragma warn_extracomma on
enum { mouse, cat, dog, }; /* WARNING: compiler expects an identifier after final comma. */

The compiler ignores terminating commas in enumerations when compiling source code that conforms to the ISO/IEC 9899-1999 ("C99") standard.

By default, this pragma is disabled.

31.14 warn_filenamecaps

Controls the recognition of conflicts involving case-sensitive filenames within user includes.

Syntax

#pragma warn_filenamecaps on | off | reset

Remarks

If you enable this pragma, the compiler issues a warning message when an #include directive capitalizes a filename within a user include differently from the way the filename appears on a disk. It also detects use of "8.3" DOS filenames in Windows® operating systems when a long filename is available. Use this pragma to avoid porting problems to operating systems with case-sensitive file names.

By default, this pragma only checks the spelling of user includes such as the following:

#include "
file
"

For more information on checking system includes, see warn_filenamecaps_system.

By default, this pragma is off.
31.15  warn_filenamecaps_system

Controls the recognition of conflicts involving case-sensitive filenames within system includes.

Syntax

#pragma warn_filenamecaps_system on | off | reset

Remarks

If you enable this pragma along with warn_filenamecaps, the compiler issues a warning message when an #include directive capitalizes a filename within a system include differently from the way the filename appears on a disk. It also detects use of "8.3" DOS filenames in Windows® systems when a long filename is available. This pragma helps avoid porting problems to operating systems with case-sensitive file names.

To check the spelling of system includes such as the following:

#include <
file
>

Use this pragma along with the warn_filenamecaps pragma.

By default, this pragma is off.

NOTE

Some SDKs (Software Developer Kits) use "colorful" capitalization, so this pragma may issue a lot of unwanted messages.

31.16  warn_hiddenlocals

Controls the recognition of a local variable that hides another local variable.

Syntax

#pragma warn_hiddenlocals on | off | reset

Remarks
When on, the compiler issues a warning message when it encounters a local variable that hides another local variable. An example appears in Listing 31-11 on page 477.

**Listing 31-11. Example of hidden local variables warning**

```c
#pragma warn_hiddenlocals on
void func(int a)
{
   int a; /* WARNING: this 'a' obscures argument 'a'.
}
```

This pragma does not correspond to any CodeWarrior IDE panel setting. By default, this setting is off.

### 31.17 warn_illpragma

Controls the recognition of invalid pragma directives.

**Syntax**

```c
#pragma warn_illpragma on | off | reset
```

**Remarks**

If you enable this pragma, the compiler displays a warning message when it encounters a pragma it does not recognize.

By default, this setting is off.

### 31.18 warn_illtokenpasting

Controls whether or not to issue a warning message for improper preprocessor token pasting.

**Syntax**

```c
#pragma warn_illtokenpasting on | off | reset
```

**Remarks**

An example of this is shown below:
Token pasting is used to create a single token. In this example, \( y \) and \( x \) cannot be combined. Often the warning message indicates the macros uses "##" unnecessarily.

This pragma does not correspond to any panel setting. By default, this pragma is on.

### 31.19 warn_illunionmembers

Controls whether or not to issue a warning message for invalid union members, such as unions with reference or non-trivial class members.

**Syntax**

```
#pragma warn_illunionmembers on | off | reset
```

**Remarks**

This pragma does not correspond to any panel setting. By default, this pragma is on.

### 31.20 warn_impl_f2i_conv

Controls the issuing of warning messages for implicit float-to-int conversions.

**Syntax**

```
#pragma warn_impl_f2i_conv on | off | reset
```

**Remarks**

If you enable this pragma, the compiler issues a warning message for implicitly converting floating-point values to integral values. Listing 31-12 on page 478 provides an example.

**Listing 31-12. Example of Implicit float-to-int Conversion**

```
#pragma warn_impl_f2i_conv on
float f;
signed int si;
int main()
{
    f = si; /* WARNING */
#pragma warn_impl_f2i_conv off
```
By default, this pragma is `on`.

### 31.21 `warn_impl_i2f_conv`

Controls the issuing of warning messages for implicit `int`-to-`float` conversions.

**Syntax**

```c
#pragma warn_impl_i2f_conv on | off | reset
```

**Remarks**

If you enable this pragma, the compiler issues a warning message for implicitly converting integral values to floating-point values. Listing 31-13 on page 479 shows an example.

**Listing 31-13. Example of implicit `int`-to-`float` conversion**

```c
#pragma warn_impl_i2f_conv on
float f;
signed int si;

int main()
{
    si = f; /* WARNING */
#pragma warn_impl_i2f_conv off
    f = si; /* OK */
}
```

By default, this pragma is `off`.

### 31.22 `warn_impl_s2u_conv`

Controls the issuing of warning messages for implicit conversions between the `signed int` and `unsigned int` data types.

**Syntax**

```c
#pragma warn_impl_s2u_conv on | off | reset
```

**Remarks**
If you enable this pragma, the compiler issues a warning message for implicitly converting either from `signed int` to `unsigned int` or vice versa. Listing 31-14 on page 480 provides an example.

Listing 31-14. Example of implicit conversions between `signed int` and `unsigned int`

```c
#pragma warn_impl_s2u_conv on
signed int si;
unsigned int ui;

int main()
{
    ui = si; /* WARNING */
    si = ui; /* WARNING */
}
```

By default, this pragma is enabled.

31.23 `warn_implicitconv`

Controls the issuing of warning messages for all implicit arithmetic conversions.

**Syntax**

```c
#pragma warn_implicitconv on | off | reset
```

**Remarks**

If you enable this pragma, the compiler issues a warning message for all implicit arithmetic conversions when the destination type might not represent the source value. Listing 31-15 on page 480 provides an example.

Listing 31-15. Example of Implicit Conversion

```c
#pragma warn_implicitconv on
float f;
signed int si;
unsigned int ui;

int main()
{
    f = si; /* WARNING */
    si = f; /* WARNING */
    ui = si; /* WARNING */
    si = ui; /* WARNING */
}
```
NOTE
This option "opens the gate" for the checking of implicit conversions. The sub-pragmas warn_impl_f2i_conv, warn_impl_i2f_conv, and warn_impl_s2u_conv control the classes of conversions checked.
By default, this pragma is off.

31.24 warn_largeargs

Controls the issuing of warning messages for passing non-'int' numeric values to unprototyped functions.

Syntax

#pragma warn_largeargs on | off | reset

Remarks

If you enable this pragma, the compiler issues a warning message if you attempt to pass a non-integer numeric value, such as a float or long long, to an unprototyped function when the require_prototypes pragma is disabled.

This pragma does not correspond to any panel setting. By default, this pragma is off.

31.25 warn_missingreturn

Issues a warning message when a function that returns a value is missing a return statement.

Syntax

#pragma warn_missingreturn on | off | reset

Remarks

An example is shown in Listing 31-16 on page 481.

Listing 31-16. Example of warn_missingreturn pragma

#pragma warn_missingreturn on
31.26  warn_no_side_effect

Controls the issuing of warning messages for redundant statements.

Syntax

#pragma warn_no_side_effect on | off | reset

Remarks

If you enable this pragma, the compiler issues a warning message when it encounters a statement that produces no side effect. To suppress this warning message, cast the statement with `(void)`.

Listing 31-17 on page 482 provides an example.

Listing 31-17. Example of Pragma warn_no_side_effect

```c
#pragma warn_no_side_effect on
void func(int a, int b)
{
    a+b;    /* WARNING: expression has no side effect */
    (void)(a+b); /* OK: void cast suppresses warning. */
}
```

By default, this pragma is `off`.

31.27  warn_padding

Controls the issuing of warning messages for data structure padding.

Syntax

#pragma warn_padding on | off | reset

Remarks

If you enable this pragma, the compiler warns about any bytes that were implicitly added after an ANSI C `struct` member to improve memory alignment. Refer to the appropriate `Targeting` manual for more information on how the compiler pads data structures for a particular processor or operating system.

By default, this setting is `off`.

Freescale Semiconductor, Inc.
### 31.28 warn_pch_portability

Controls whether or not to issue a warning message when `#pragma once` is used in a precompiled header.

**Syntax**

```
#pragma warn_pch_portability on | off | reset
```

**Remarks**

If you enable this pragma, the compiler issues a warning message when you use `#pragma once` in a precompiled header. This helps you avoid situations in which transferring a precompiled header from machine to machine causes the precompiled header to produce different results. For more information, see `pragma once`.

This pragma does not correspond to any panel setting. By default, this setting is `off`.

### 31.29 warn_possunwant

Controls the recognition of possible unintentional logical errors.

**Syntax**

```
#pragma warn_possunwant on | off | reset
```

**Remarks**

If you enable this pragma, the compiler checks for common, unintended logical errors:

- An assignment in either a logical expression or the conditional portion of an `if`, `while`, or `for` expression. This warning message is useful if you use `=` when you mean to use `==`. The listing below shows an example.

```
Listing 31-18. Confusing = and == in Comparisons

if (a=b) f(); /* WARNING: a=b is an assignment. */
if ((a=b)!=0) f(); /* OK: (a=b)!=0 is a comparison. */
if (a==b) f(); /* OK: (a==b) is a comparison. */
```
• An equal comparison in a statement that contains a single expression. This check is useful if you use `==` when you meant to use `=`. The listing below shows an example.

Listing 31-19. Confusing `=` and `==` Operators in Assignments

```cpp
a == 0;        // WARNING: This is a comparison.
a = 0;         // OK: This is an assignment, no warning
```

• A semicolon (`;`) directly after a `while`, `if`, or `for` statement.

For example, the listing below, generates a warning message.

Listing 31-20. Empty statement

```cpp
i = sockcount();
while (--i);  /* WARNING: empty loop. */
    matchsock(i);
```

If you intended to create an infinite loop, put white space or a comment between the while statement and the semicolon. The statements in the listing below, suppress the above error or warning messages.

Listing 31-21. Intentional empty statements

```cpp
while (i++) ; /* OK: White space separation. */
while (i++) /* OK: Comment separation */ ;
```

By default, this pragma is `off`.

### 31.30 `warn_ptr_int_conv`

Controls the recognition the conversion of pointer values to incorrectly-sized integral values.

**Syntax**

```cpp
#pragma warn_ptr_int_conv on | off | reset
```

**Remarks**

If you enable this pragma, the compiler issues a warning message if an expression attempts to convert a pointer value to an integral type that is not large enough to hold the pointer value.

**Listing 31-22. Example for `pragma warn_ptr_int_conv`**

```cpp
#pragma warn_ptr_int_conv on
char *my_ptr;
char too_small = (char)my_ptr;  /* WARNING: char is too small. */
```
See also "warn_any_ptr_int_conv" on page 231.

By default, this pragma is off.

### 31.31 warn_resultnotused

Controls the issuing of warning messages when function results are ignored.

**Syntax**

`#pragma warn_resultnotused on | off | reset`

**Remarks**

If you enable this pragma, the compiler issues a warning message when it encounters a statement that calls a function without using its result. To prevent this, cast the statement with `(void)`. *Listing 31-23 on page 485* provides an example.

Listing 31-23. Example of Function Calls with Unused Results

```c
#pragma warn_resultnotused on
extern int bar();
void func()
{
    bar(); /* WARNING: result of function call is not used. */
    void(bar()); /* OK: void cast suppresses warning. */
}
```

This pragma does not correspond to any panel setting. By default, this pragma is off.

### 31.32 warn_undefmacro

Controls the detection of undefined macros in `#if` and `#elif` directives.

**Syntax**

`#pragma warn_undefmacro on | off | reset`

**Remarks**

*Listing 31-24 on page 485* provides an example.

Listing 31-24. Example of Undefined Macro

```c
#if BADMACRO == 4 /* WARNING: undefined macro. */
```
Use this pragma to detect the use of undefined macros (especially expressions) where the default value 0 is used. To suppress this warning message, check if defined first.

**NOTE**
A warning message is only issued when a macro is evaluated. A short-circuited "&&" or "||" test or unevaluated "?:" will not produce a warning message.

By default, this pragma is `off`.

### 31.33 `warn_uninitializedvar`

Controls the compiler to perform some dataflow analysis and emits warning messages whenever local variables are initialized before being used.

**Syntax**

```
#pragma warn_uninitializedvar on | off | reset
```

**Remarks**

This pragma has no corresponding setting in the CodeWarrior IDE. By default, this pragma is `on`.

**NOTE**

`#pragma warn_possiblyuninitializedvar` is a superset of `#pragma warn_uninitializedvar`.

### 31.34 `warn_unusedarg`

Controls the recognition of unreferenced arguments.

**Syntax**

```
#pragma warn_unusedarg on | off | reset
```

**Remarks**

If you enable this pragma, the compiler issues a warning message when it encounters an argument you declare but do not use.
This check helps you find arguments that you either misspelled or did not use in your program. The listing below shows an example.

**Listing 31-25. Warning about unused function arguments**

```c
void func(int temp, int error);
{
    error = do_something(); /* WARNING: temp is unused. */
}
```

To prevent this warning, you can declare an argument in a few ways:

- Use the pragma `unused`, as shown in the listing below.

  **Listing 31-26. Using pragma unused() to prevent unused argument messages**

  ```c
  void func(int temp, int error)
  {
      #pragma unused (temp)
      /* Compiler does not warn that temp is not used. */
      error=do_something();
  }
  ```

- Do not give the unused argument a name. The listing below shows an example.

  The compiler allows this feature in C++ source code. To allow this feature in C source code, disable ANSI strict checking.

  **Listing 31-27. Unused, Unnamed Arguments**

  ```c
  void func(int /* temp */, int error)
  {
      /* Compiler does not warn that "temp" is not used. */
      error=do_something();
  }
  ```

This pragma corresponds to the **Unused Arguments** setting in the **C/C++ Warnings Panel**. By default, this pragma is **off**.

### 31.35 `warn_unusedvar`

Controls the recognition of unreferenced variables.

**Syntax**

```
#pragma warn_unusedvar on | off | reset
```

**Remarks**
If you enable this pragma, the compiler issues a warning message when it encounters a variable you declare but do not use.

This check helps you find variables that you either misspelled or did not use in your program. Listing 31-28 on page 488 shows an example.

Listing 31-28. Unused Local Variables Example

```c
int error;
void func(void)
{
    int temp, errer; /* NOTE: errer is misspelled. */
    error = do_something(); /* WARNING: temp and errer are unused. */
}
```

If you want to use this warning but need to declare a variable that you do not use, include the pragma `unused`, as in Listing 31-29 on page 488.

Listing 31-29. Suppressing Unused Variable Warnings

```c
void func(void)
{
    int i, temp, error;

    #pragma unused (i, temp) /* Do not warn that i and temp */
    error = do_something();    /* are not used */
}
```

By default, this pragma is off.
Chapter 32
Pragmas for Preprocessing

32.1 check_header_flags

Controls whether or not to ensure that a precompiled header's data matches a project's target settings.

Syntax

#pragma check_header_flags on | off | reset

Remarks

This pragma affects precompiled headers only.

If you enable this pragma, the compiler verifies that the precompiled header's preferences for double size, int size, and floating point math correspond to the build target's settings. If they do not match, the compiler generates an error message.

If your precompiled header file depends on these settings, enable this pragma. Otherwise, disable it.

This pragma does not correspond to any CodeWarrior IDE panel setting. By default, this pragma is off.

32.2 faster_pch_gen

Controls the performance of precompiled header generation.

Syntax

#pragma faster_pch_gen on | off | reset
Remarks

If you enable this pragma, generating a precompiled header can be much faster, depending on the header structure. However, the precompiled file can also be slightly larger.

This pragma does not correspond to any panel setting. By default, this setting is off.

32.3 flat_include

Controls whether or not to ignore relative path names in #include directives.

Syntax

#pragma flat_include on | off | reset

Remarks

For example, when on, the compiler converts this directive

#include <sys/stat.h>

to

#include <stat.h>

Use this pragma when porting source code from a different operating system, or when a CodeWarrior IDE project's access paths cannot reach a given file.

By default, this pragma is off.

32.4 fullpath_file

Controls if __FILE__ macro expands to a full path or the base file name.

Syntax

#pragma fullpath_file on | off | reset

Remarks

When this pragma on, the __FILE__ macro returns a full path to the file being compiled, otherwise it returns the base file name.
32.5  **fullpath_prepdump**

Shows the full path of included files in preprocessor output.

**Syntax**

```
#pragma fullpath_prepdump on | off | reset
```

**Remarks**

If you enable this pragma, the compiler shows the full paths of files specified by the `#include` directive as comments in the preprocessor output. Otherwise, only the file name portion of the path appears.

By default, this pragma is off.

32.6  **keepcomments**

Controls whether comments are emitted in the preprocessor output.

**Syntax**

```
#pragma keepcomments on | off | reset
```

**Remarks**

By default, this pragma is off.

32.7  **line_prepdump**

Shows `#line` directives in preprocessor output.

**Syntax**

```
#pragma line_prepdump on | off | reset
```

**Remarks**
If you enable this pragma, #line directives appear in preprocessing output. The compiler also adjusts line spacing by inserting empty lines.

Use this pragma with the command-line compiler's -E option to make sure that #line directives are inserted in the preprocessor output.

By default, this pragma is off.

### 32.8 macro_prepdump

Controls whether the compiler emits `#define` and `#undef` directives in preprocessing output.

**Syntax**

```
#pragma macro_prepdump on | off | reset
```

**Remarks**

Use this pragma to help unravel confusing problems like macros that are aliasing identifiers or where headers are redefining macros unexpectedly.

### 32.9 msg_show_lineref

Controls diagnostic output involving #line directives to show line numbers specified by the #line directives in error and warning messages.

**Syntax**

```
#pragma msg_show_lineref on | off | reset
```

**Remarks**

This pragma does not correspond to any CodeWarrior IDE panel setting. By default, this pragma is on.

### 32.10 msg_show_realref

Controls diagnostic output involving #line directives to show actual line numbers in error and warning messages.
Syntax

#pragma msg_show_realref on | off | reset

Remarks

This pragma does not correspond to any CodeWarrior IDE panel setting. By default, this pragma is **on**.

### 32.11 notonce

Controls whether or not the compiler lets included files be repeatedly included, even with #pragma once on.

Syntax

#pragma notonce

Remarks

If you enable this pragma, files can be repeatedly included, even if you have enabled #pragma once on. For more information, see "once" on page 253.

This pragma does not correspond to any CodeWarrior IDE panel setting.

### 32.12 oldPragma_once

This pragma is no longer available.

### 32.13 once

Controls whether or not a header file can be included more than once in the same compilation unit.

Syntax

#pragma once [ on ]

Remarks
Use this pragma to ensure that the compiler includes header files only once in a source file. This pragma is especially useful in precompiled header files.

There are two versions of this pragma:

```c
#pragma once
```

and

```c
#pragma once on
```

Use `#pragma once` in a header file to ensure that the header file is included only once in a source file. Use `#pragma once on` in a header file or source file to ensure that any file is included only once in a source file. When a `once` option or pragma is used, a header file of same name in another directory is not included.

Beware that when using `#pragma once on`, precompiled headers transferred from one host machine to another might not give the same results during compilation. This inconsistency is because the compiler stores the full paths of included files to distinguish between two distinct files that have identical file names but different paths. Use the `warn_pch_portability` pragma to issue a warning message when you use `#pragma once on` in a precompiled header.

Also, if you enable the `oldPragma_once on` pragma, the `once` pragma completely ignores path names.

This pragma does not correspond to any panel setting. By default, this pragma is `off`.

### 32.14 pop, push

Saves and restores pragma settings.

**Syntax**

```c
#pragma push
```

```c
#pragma pop
```

**Remarks**

The pragma `push` saves all the current pragma settings. The pragma `pop` restores all the pragma settings that resulted from the last `push` pragma. For example, see Listing 32-1 on page 494.

**Listing 32-1. push and pop example**
#pragma ANSI_strict on
#pragma push /* Saves all compiler settings. */
#pragma ANSI_strict off
#pragma pop /* Restores ANSI_strict to on. */

This pragma does not correspond to any panel setting. By default, this pragma is off.

**Tip**
Pragmas directives that accept `on | off | reset` already form a stack of previous option values. It is not necessary to use `#pragma pop` or `#pragma push` with such pragmas.

### 32.15 `pragma_prepdump`

Controls whether pragma directives in the source text appear in the preprocessing output.

**Syntax**

```c
#pragma pragma_prepdump on | off | reset
```

**Remarks**

By default, this pragma is off.

**Tip**
When submitting bug reports with a preprocessor dump, be sure this option is enabled.

### 32.16 `precompile_target`

Specifies the file name for a precompiled header file.

**Syntax**

```c
#pragma precompile_target
filename
```

**Parameters**

`filename`

A simple file name or an absolute path name. If `filename` is a simple file name, the compiler saves the file in the same folder as the source file. If `filename` is a path name, the compiler saves the file in the specified folder.
Remarks

If you do not specify the file name, the compiler gives the precompiled header file the same name as its source file.

Listing 32-2 on page 496 shows sample source code from a precompiled header source file. By using the predefined symbols __cplusplus and the pragma precompile_target, the compiler can use the same source code to create different precompiled header files for C and C++.

Listing 32-2. Using #pragma precompile_target

```c
#ifdef __cplusplus
    #pragma precompile_target "MyCPPHeaders"
#else
    #pragma precompile_target "MyCHeaders"
#endif
```

This pragma does not correspond to any panel setting.

32.17 simple_prepdump

Controls the suppression of comments in preprocessing output.

Syntax

```
#pragma simple_prepdump on | off | reset
```

Remarks

By default, the compiler adds comments about the current include file being in preprocessing output. Enabling this pragma disables these comments.

By default, this pragma is off.

32.18 space_prepdump

Controls whether or not the compiler removes or preserves whitespace in the preprocessor's output.

Syntax

```
#pragma space_prepdump on | off | reset
```
Remarks

This pragma is useful for keeping the starting column aligned with the original source code, though the compiler attempts to preserve space within the line. This pragma does not apply to expanded macros.

By default, this pragma is off.

32.19 srcselincludes

Controls the lookup of #include files.

Syntax

#pragma srcselincludes on | off | reset

Remarks

When on, the compiler looks for #include files relative to the previously included file (not just the source file). When off, the compiler uses the CodeWarrior IDE's access paths or the access paths specified with the -ir option.

Use this pragma when multiple files use the same file name and are intended to be included by another header file in that directory. This is a common practice in UNIX programming.

This pragma corresponds to the Source-relative includes option in the Access Paths panel. By default, this pragma is off.

32.20 syspath_once

Controls how included files are treated when #pragmaonce is enabled.

Syntax

#pragma syspath_once on | off | reset

Remarks

When this pragma and pragma once are set to on, the compiler distinguishes between identically-named header files referred to in
When this pragma is **off** and pragma once is **on**, the compiler will ignore a file that uses a

```c
#include <file-name>
```

directive if it has previously encountered another directive of the form

```c
#include "file-name"
```

for an identically-named header file.

shows an example.

This pragma does not correspond to any panel setting. By default, this setting is **on**.

**Listing 32-3. Pragma syspath_once example**

```c
#pragma syspath_once off
#pragma once on /* Include all subsequent files only once. */
#include "sock.h"
#include <sock.h> /* Skipped because syspath_once is off. */
```
Chapter 33
Pragmas for Library and Linking

33.1 always_import

Controls whether or not # include directives are treated as #pragma import directives.

Syntax

#pragma always_import on | off | reset

Remarks

If you enable this pragma, the compiler treats all #include statements as #pragma import statements.

This pragma does not correspond to any CodeWarrior IDE panel setting. By default, this pragma is off.

33.2 export

Controls the exporting of data and functions to be accessible from outside a program or library.

Syntax

#pragma export on | off | reset

#pragma export list
    name1 [, name2 , ...]
Names of functions or global variables to export.

Remarks
When using the \texttt{#pragma export} format, all functions in the source file being compiled will be accessible from outside the program or library that the compiler and linker are building.

Use the \texttt{#pragma export list} format to specify global variables and functions for exporting. In C++, this form of the pragma applies to all variants of an overloaded function. You cannot use this pragma for C++ member functions or static class members. \textbf{Listing 33-1 on page 500} shows an example:

\begin{verbatim}
extern int f(),g;
#pragma export list f,g
\end{verbatim}

### 33.3 \texttt{import}

Controls the importing of global data or functions.

Syntax

\begin{verbatim}
#pragma import on | off | reset
#pragma import list
   name1 [, 
   name2 , ...]
\end{verbatim}

Names of functions or global variables to import.

Remarks
When using the \texttt{#pragma import} on format, all functions are automatically imported.

Use the \texttt{#pragma import list} format to specify data or functions for importing. In C++, this form of the pragma applies to all variants of an overloaded function. You cannot use this pragma for C++ member functions or static class members. \textbf{Listing 33-2 on page 500} shows an example:

\begin{verbatim}
extern int f(),g;
#pragma import list f,g
\end{verbatim}
This pragma does not correspond to any CodeWarrior IDE panel setting. By default, this pragma is off.

### 33.4 lib_export

Controls the exporting of data or functions.

#### Syntax

```c
#pragma lib_export on | off | reset
#pragma lib_export list
   name1 [, name2 ]*
```

#### Remarks

When using the `#pragma lib_export on` format, the linker marks all data and functions that are within the pragma's scope for export.

Use the `#pragma lib_export list` format to tag specific data or functions for exporting. In C++, this form of the pragma applies to all variants of an overloaded function. You cannot use this pragma for C++ member functions or static class members.

Listing 33-3 on page 501 shows an example:

#### Listing 33-3. Example of a lib_export List

```c
extern int f(),g;
#pragma lib_export list f,g
```

This pragma does not correspond to any panel setting. By default, this pragma is disabled.
Chapter 34
Pragmas for Code Generation

34.1 aggressive_inline

Instructs compiler to be more aggressive when inlining.

Syntax

#pragma aggressive_inline on | off | reset

Remarks

The IPA-based inliner (-ipa file) will inline more functions when this option is enabled. This option can cause code bloat in programs that overuse inline functions. Default is off.

34.2 dont_reuse_strings

Controls whether or not to store identical character string literals separately in object code.

Syntax

#pragma dont_reuse_strings on | off | reset

Remarks

Normally, C and C++ programs should not modify character string literals. Enable this pragma if your source code follows the unconventional practice of modifying them.
If you enable this pragma, the compiler separately stores identical occurrences of character string literals in a source file.

If this pragma is disabled, the compiler stores a single instance of identical string literals in a source file. The compiler reduces the size of the object code it generates for a file if the source file has identical string literals.

The compiler always stores a separate instance of a string literal that is used to initialize a character array. Listing 34-1 on page 504 shows an example.

Although the source code contains 3 identical string literals, "cat", the compiler will generate 2 instances of the string in object code. The compiler will initialize str1 and str2 to point to the first instance of the string and will initialize str3 to contain the second instance of the string.

Using str2 to modify the string it points to also modifies the string that str1 points to. The array str3 may be safely used to modify the string it points to without inadvertently changing any other strings.

By default, this pragma is off.

Listing 34-1. Reusing string literals

```c
#pragma dont_reuse_strings off
void strchange(void)
{
    const char* str1 = "cat";
    char* str2 = "cat";
    char str3[] = "cat";

    *str2 = 'h'; /* str1 and str2 point to "hat"! */
    str3[0] = 'b';
    /* OK: str3 contains "bat", *str1 and *str2 unchanged. */
}
```

34.3 enumsalwaysint

Specifies the size of enumerated types.

Syntax

```
#pragma enumsalwaysint on | off | reset
```

Remarks
If you enable this pragma, the C/C++ compiler makes an enumerated type the same size as an `int`. If an enumerated constant is larger than `int`, the compiler generates an error message. Otherwise, the compiler makes an enumerated type the size of any integral type. It chooses the integral type with the size that most closely matches the size of the largest enumerated constant. The type could be as small as a `char` or as large as a `long long`.

Listing 34-2 on page 505 shows an example.

**Listing 34-2. Example of Enumerations the Same as Size as int**

```c
enum SmallNumber { One = 1, Two = 2 };
/* If you enable enumsalwaysint, this type is
the same size as an int. Otherwise, this type is
the same size as a char. */

enum BigNumber
{ ThreeThousandMillion = 3000000000 };   
/* If you enable enumsalwaysint, the compiler might
generate an error message. Otherwise, this type is
the same size as a long long. */
```

By default, this pragma is **off**.

### 34.4 enums_signed

Changes the underlying enumeration type search order.

**Syntax**

```c
#pragma enums_signed on | off | reset
```

**Remarks**

Enabling this option changes the underlying enumeration type search order. The underlying type for an enumeration where all enumerators are >= 0 is the first one of these types in which all values can be represented:

- `signed char` (*)
- `unsigned char`
- `signed short` (*)
- `unsigned short`
- `signed int` (*)
- `unsigned int`
- `signed long` (*)
34.5  _errno_name_

Tells the optimizer how to find the _errno_ identifier.

**Syntax**

```c
#pragma errno_name id | ...
```

**Remarks**

When this pragma is used, the optimizer can use the identifier _errno_ (either a macro or a function call) to optimize standard C library functions better. If not used, the optimizer makes worst-case assumptions about the effects of calls to the standard C library.

**NOTE**

The MSL C library already includes a use of this pragma, so you would only need to use it for third-party C libraries.

If _errno_ resolves to a variable name, specify it like this:

```c
#pragma errno_name _Errno
```

If _errno_ is a function call accessing ordinarily inaccessible global variables, use this form:

```c
#pragma errno_name ...
```

Otherwise, do not use this pragma to prevent incorrect optimizations.

This pragma does not correspond to any panel setting. By default, this pragma is unspecified (worst case assumption).

34.6  _explicit_zero_data_

Controls the placement of zero-initialized data.
Syntax

#pragma explicit_zero_data on | off | reset

Remarks

Places zero-initialized data into the initialized data section instead of the BSS section when on.

By default, this pragma is off.

34.7 float_constants

Controls how floating pointing constants are treated.

Syntax

#pragma float_constants on | off | reset

Remarks

If you enable this pragma, the compiler assumes that all unqualified floating point constant values are of type float, not double. This pragma is useful when porting source code for programs optimized for the "float" rather than the "double" type.

When you enable this pragma, you can still explicitly declare a constant value as double by appending a "D" suffix.

This pragma does not correspond to any panel setting. By default, this pragma is disabled.

34.8 instmgr_file

Controls where the instance manager database is written, to the target data directory or to a separate file.

Syntax

#pragma instmgr_file "name"

Remarks
When the Use Instance Manager option is on, the IDE writes the instance manager database to the project’s data directory. If the #pragma instmgr_file is used, the database is written to a separate file.

Also, a separate instance file is always written when the command-line tools are used.

NOTE
Should you need to report a bug, you can use this option to create a separate instance manager database, which can then be sent to technical support with your bug report.

NOTE

34.9  longlong

Controls the availability of the long long type.

Syntax

#pragma longlong on | off | reset

Remarks

When this pragma is enabled and the compiler is translating "C90" source code (ISO/IEC 9899-1990 standard), the compiler recognizes a data type named long long. The long long type holds twice as many bits as the long data type.

This pragma does not correspond to any CodeWarrior IDE panel setting.

By default, this pragma is on for processors that support this type. It is off when generating code for processors that do not support, or cannot turn on, the long long type.

34.10  longlong Enums

Controls whether or not enumerated types may have the size of the long long type.

Syntax

#pragma longlong Enums on | off | reset

Remarks
This pragma lets you use enumerators that are large enough to be \texttt{long long} integers. It is ignored if you enable the \texttt{enumsalwaysint} pragma (described in "\texttt{enumsalwaysint}" on page 260).

This pragma does not correspond to any panel setting. By default, this setting is enabled.

### 34.11 min_enum_size

Specifies the size, in bytes, of enumeration types.

**Syntax**

\begin{verbatim}
#pragma min_enum_size 1 | 2 | 4
\end{verbatim}

**Remarks**

Turning on the \texttt{enumsalwaysint} pragma overrides this pragma. The default is 1.

### 34.12 pool_strings

Controls how string literals are stored.

**Syntax**

\begin{verbatim}
#pragma pool_strings on | off | reset
\end{verbatim}

**Remarks**

If you enable this pragma, the compiler collects all string constants into a single data object so your program needs one data section for all of them. If you disable this pragma, the compiler creates a unique data object for each string constant. While this decreases the number of data sections in your program, on some processors it also makes your program bigger because it uses a less efficient method to store the address of the string.

This pragma is especially useful if your program is large and has many string constants or uses the CodeWarrior Profiler.

\textbf{NOTE}

If you enable this pragma, the compiler ignores the setting of the \texttt{pcrelstrings} pragma.
### 34.13 readonly_strings

Controls whether string objects are placed in a read-write or a read-only data section.

**Syntax**

```plaintext
#pragma readonly_strings on | off | reset
```

**Remarks**

If you enable this pragma, literal strings used in your source code are output to the read-only data section instead of the global data section. In effect, these strings act like `const char*`, even though their type is really `char*`.

This pragma does not correspond to any IDE panel setting.

### 34.14 reverse_bitfields

Controls whether or not the compiler reverses the bitfield allocation.

**Syntax**

```plaintext
#pragma reverse_bitfields on | off | reset
```

**Remarks**

This pragma reverses the bitfield allocation, so that bitfields are arranged from the opposite side of the storage unit from that ordinarily used on the target. The compiler still orders the bits within a single bitfield such that the lowest-valued bit is in the right-most position.

This pragma does not correspond to any panel setting. By default, this pragma is disabled.

**NOTE**

Limitation: please be aware of the following limitations when this pragma is set to on:
- The data types of the bit-fields must be the same data type.
- The structure (`struct`) or class must not contain non-bit-field members; however, the structure (`struct`) can be the member of another structure.
34.15 store_object_files

Controls the storage location of object data, either in the target data directory or as a separate file.

Syntax

#pragma store_object_files on | off | reset

Remarks

By default, the IDE writes object data to the project's target data directory. When this pragma is on, the object data is written to a separate object file.

NOTE

For some targets, the object file emitted may not be recognized as actual object data.

This pragma does not correspond to any panel setting. By default, this pragma is off.
Pragmas for Optimization should be placed outside the functions to which it is desired that they be applied.

This chapter explains the following pragmas:

- `global_optimizer`
- `ipa`
- `ipa_inline_max_auto_size`
- `ipa_not_complete`
- `load_store_elimination`
- `opt_common_subs`
- `opt_dead_assignments`
- `opt_dead_code`
- `opt_lifetimes`
- `opt_loop_invariants`
- `opt_propagation`
- `opt_strength_reduction`
- `opt_strength_reduction_strict`
- `opt_unroll_loops`
- `opt_vectorize_loops`
- `optimization_level`
- `optimize_for_size`
- `optimizewithasm`
- `pack`
- `strictheonheaderchecking`

### 35.1 global_optimizer

Controls whether the Frontend IR Optimizer is invoked by the compiler.
### Syntax

```c
#pragma global_optimizer on | off | reset
```

### Remarks

In most compilers, this #pragma determines whether the Frontend IR Optimizer is invoked. If disabled, only simple optimizations and back-end optimizations are performed.

**NOTE**

This is not the same as `#pragma optimization_level`. The Frontend IR Optimizer is invoked even at `optimization_level0` if `#pragma global_optimizer` is enabled.

This pragma does not correspond to any panel setting. By default, this setting is `on`.

### 35.2 ipa

Specifies how to apply interprocedural analysis optimizations.

#### Syntax

```c
#pragma ipa program | file | on | function | off
```

#### Remarks

See [Interprocedural Analysis](#).

Place this pragma at the beginning of a source file, before any functions or data have been defined. There are three levels of interprocedural analysis:

- program-level: the compiler translates all source files in a program then optimizes object code for the entire program
- file-level: the compiler translates each file and applies this optimization to the file
- function-level: the compiler does not apply interprocedural optimization

The options `file` and `on` are equivalent. The options `function` and `off` are equivalent.

### 35.3 ipa_inline_max_auto_size

Determines the maximum complexity for an auto-inlined function.
Syntax

#pragma ipa_inline_max_auto_size (intval)

Parameters

intval

The intval value is an approximation of the number of statements in a function, the current default value is 500, which is approximately equal to 100 statement function. Selecting a zero value will disable the IPA auto inlining.

Remarks

The size of the code objects that are not referenced by address and are only called once is specified above a certain threshold using this pragma, preventing them from being marked as inline.

35.4 ipa_not_complete

Controls the usage of Complete Program IPA mode by the compiler.

Syntax

#pragma ipa_not_complete on | off | reset

Remarks

In Complete Program IPA mode, the compiler assumes that the IPA graph is complete and that there are no external entry points other than main(), static initialization or force export functions. The compiler will not use this mode if the

The Complete Program IPA mode is not used by the compiler if:

- the program has no main() and no force export functions.
- the pragma is on the context of main() or force export functions.

NOTE

The compiler will be more aggressive in the Complete Program IPA mode.

Any extern object that is not main(), static initialization code or force export and not directly or indirectly used, will be deadstipped by the compiler and will not appear in the object and/or executable files. By default, this setting is off.
### 35.5 load_store_elimination

Controls the use of load store elimination optimization.

**Syntax**

```c
#pragma load_store_elimination on|off|reset
```

**Remarks**

This optimization will track all loads and stores within the entire function and figure out which ones are unnecessary and deletes them.

This optimization will be enabled by default only when the opt level is 3 or higher.

This pragma does not correspond to any panel setting.

### 35.6 opt_common_subs

Controls the use of common subexpression optimization.

**Syntax**

```c
#pragma opt_common_subs on | off | reset
```

**Remarks**

If you enable this pragma, the compiler replaces similar redundant expressions with a single expression. For example, if two statements in a function both use the expression

```c
a * b * c + 10
```

the compiler generates object code that computes the expression only once and applies the resulting value to both statements.

The compiler applies this optimization to its own internal representation of the object code it produces.

This pragma does not correspond to any panel setting. By default, this settings is related to the `global_optimizer` pragma.
35.7 opt_dead_assignments

Controls the use of dead store optimization.

Syntax

#pragma opt_dead_assignments on | off | reset

Remarks

If you enable this pragma, the compiler removes assignments to unused variables before reassigning them.

This pragma does not correspond to any panel setting. By default, this setting is related to the "global_optimizer" on page 471 level.

35.8 opt_dead_code

Controls the use of dead code optimization.

Syntax

#pragma opt_dead_code on | off | reset

Remarks

If you enable this pragma, the compiler removes a statement that other statements never execute or call.

This pragma does not correspond to any panel setting. By default, this setting is related to the "global_optimizer" on page 471 level.

35.9 opt_lifetimes

Controls the use of lifetime analysis optimization.

Syntax

#pragma opt_lifetimes on | off | reset

Remarks
If you enable this pragma, the compiler uses the same processor register for different variables that exist in the same routine but not in the same statement.

This pragma does not correspond to any panel setting. By default, this settings is related to the "global_optimizer" on page 471 level.

35.10 opt_loop_invariants

Controls the use of loop invariant optimization.

Syntax

```plaintext
#pragma opt_loop_invariants on | off | reset
```

Remarks

If you enable this pragma, the compiler moves all computations that do not change inside a loop outside the loop, which then runs faster.

This pragma does not correspond to any panel setting.

35.11 opt_propagation

Controls the use of copy and constant propagation optimization.

Syntax

```plaintext
#pragma opt_propagation on | off | reset
```

Remarks

If you enable this pragma, the compiler replaces multiple occurrences of one variable with a single occurrence.

This pragma does not correspond to any panel setting. By default, this settings is related to the "global_optimizer" on page 471 level.

35.12 opt_strength_reduction
Controls the use of strength reduction optimization.

Syntax

#pragma opt_strength_reduction on | off | reset

Remarks

If you enable this pragma, the compiler replaces array element arithmetic instructions with pointer arithmetic instructions to make loops faster.

This pragma does not correspond to any panel setting. By default, this settings is related to the "global_optimizer" on page 471 level.

35.13 opt_strength_reduction_strict

Uses a safer variation of strength reduction optimization.

Syntax

#pragma opt_strength_reduction_strict on | off | reset

Remarks

Like the opt_strength_reduction pragma, this setting replaces multiplication instructions that are inside loops with addition instructions to speed up the loops. However, unlike the regular strength reduction optimization, this variation ensures that the optimization is only applied when the array element arithmetic is not of an unsigned type that is smaller than a pointer type.

This pragma does not correspond to any panel setting. The default varies according to the compiler.

35.14 opt_unroll_loops

Controls the use of loop unrolling optimization.

Syntax

#pragma opt_unroll_loops on | off | reset

Remarks
If you enable this pragma, the compiler places multiple copies of a loop's statements inside a loop to improve its speed.

This pragma does not correspond to any panel setting. By default, this settings is related to the "global_optimizer" on page 471 level.

### 35.15 opt_vectorize_loops

Controls the use of loop vectorizing optimization.

**Syntax**

```
#pragma opt_vectorize_loops on | off | reset
```

**Remarks**

If you enable this pragma, the compiler improves loop performance.

**NOTE**

Do not confuse loop vectorizing with the vector instructions available in some processors. Loop vectorizing is the rearrangement of instructions in loops to improve performance. This optimization does not optimize a processor's vector data types.

By default, this pragma is off.

### 35.16 optimization_level

Controls global optimization.

**Syntax**

```
#pragma optimization_level 0 | 1 | 2 | 3 | 4 | reset
```

**Remarks**

This pragma specifies the degree of optimization that the global optimizer performs.

To select optimizations, use the pragma optimization_level with an argument from 0 to 4. The higher the argument, the more optimizations performed by the global optimizer. The reset argument specifies the previous optimization level.
For more information on the optimization the compiler performs for each optimization level, refer to the Targeting manual for your target platform.

These pragmas correspond to the settings in the Global Optimizations panel. By default, this pragma is disabled.

### 35.17 optimize_for_size

Controls optimization to reduce the size of object code.

```
#pragma optimize_for_size on | off | reset
```

**Remarks**

This setting lets you choose what the compiler does when it must decide between creating small code or fast code. If you enable this pragma, the compiler creates smaller object code at the expense of speed. It also ignores the `inline` directive and generates function calls to call any function declared `inline`. If you disable this pragma, the compiler creates faster object code at the expense of size.

The pragma corresponds to the **Optimize for Size** setting on the **Global Optimizations** panel.

### 35.18 optimizewithasm

Controls optimization of assembly language.

**Syntax**

```
#pragma optimizewithasm on | off | reset
```

**Remarks**

If you enable this pragma, the compiler also optimizes assembly language statements in C/C++ source code.

This pragma does not correspond to any panel setting. By default, this pragma is disabled.
35.19  pack

Stores data to reduce data size instead of improving execution performance.

Syntax

#pragma pack()

#pragma pack(0 |
  n | push | pop)

n

One of these integer values: 1, 2, 4, 8, or 16.

Remarks

Use this pragma to align data to use less storage even if the alignment might affect program performance or does not conform to the target platform's application binary interface (ABI).

If this pragma's argument is a power of 2 from 1 to 16, the compiler will store subsequent data structures to this byte alignment.

The push argument saves this pragma's setting on a stack at compile time. The pop argument restores the previously saved setting and removes it from the stack. Using this pragma with no argument or with 0 as an argument specifies that the compiler will use ABI-conformant alignment.

Not all processors support misaligned accesses, which could cause a crash or incorrect results. Even on processors which allow misaligned access, your program's performance might be reduced. Your program may have better performance if it treats the packed structure as a byte stream, then packs and unpacks each byte from the stream.

NOTE

Pragma pack is implemented somewhat differently by most compiler vendors, especially when used with bitfields. If you need portability, you are probably better off using explicit shift and mask operations in your program instead of bitfields.

35.20  strictheaderchecking

Controls how strict the compiler checks headers for standard C library functions.
Syntax

#pragma strictheaderchecking on | off | reset

Remarks

The 3.2 version compiler recognizes standard C library functions. If the correct prototype is used, and, in C++, if the function appears in the "std" or root namespace, the compiler recognizes the function, and is able to optimize calls to it based on its documented effects.

When this #pragma is on (default), in addition to having the correct prototype, the declaration must also appear in the proper standard header file (and not in a user header or source file).

This pragma does not correspond to any panel setting. By default, this pragma is on.
Chapter 36
Pragmas for Power Architecture Compiler

This chapter describes the pragmas that control how the compiler translates source code into instructions and data for Power Architecture processors.

- Diagnostic Pragmas
- Debugging Information Pragmas
- Library and Linking Pragmas
- Code Generation Pragmas
- Optimization Pragmas

36.1 Diagnostic Pragmas

This section describes the diagnostic pragmas.

- incompatible_return_small_structs
- incompatible_sfpe_double_params

36.1.1 incompatible_return_small_structs

Warns when returning structures using the R3 and R4 registers.

Syntax

#pragma incompatible_return_small_structs on | off | reset

Remarks

This pragma makes CodeWarrior-built object files more compatible with those created using a GNU compiler.
The PowerPC EABI specifies that structures that are up to 8 bytes in size should be in registers \texttt{R3} and \texttt{R4}, while larger structures are returned by accessing a hidden argument in \texttt{R3}. GCC compilers always use the hidden argument method regardless of size.

The CodeWarrior linker checks to see if you are including objects in your project that have incompatible EABI settings. If you do, a warning message is issued.

\textbf{NOTE}

Different versions of GCC compilers may fix these incompatibilities, so you should check your version if you will be mixing GCC and CodeWarrior objects.

### 36.1.2 incompatible\_sfpe\_double\_params

Warns when skipping registers for \texttt{double} data types.

\textbf{Syntax}

\begin{verbatim}
#pragma incompatible\_sfpe\_double\_params on | off | reset
\end{verbatim}

\textbf{Remarks}

This pragma makes CodeWarrior-built object files more compatible with those created with a GCC compiler.

The PowerPC EABI states that software floating-point parameters of type \texttt{double} always begin on an odd register. In other words, in the function

\begin{verbatim}
void red(long a, double b)
\end{verbatim}

\texttt{a} is passed in register \texttt{R3} and \texttt{b} is passed in registers \texttt{R5} and \texttt{R6} (effectively skipping \texttt{R4}). GCC compilers do not skip registers if objects of type \texttt{double} are passed (although it does skip them for values of type \texttt{long long}).

The CodeWarrior linker checks to see if you are including objects in your project that have incompatible EABI settings. If you do, a warning message is issued.

\textbf{NOTE}

Different versions of GCC compilers may fix these incompatibilities, so you should check your version if you will be mixing GCC and CodeWarrior object code.
36.2 Debugging Information Pragmas

This section describes the debugging information pragmas.

- `dwarf2typedefchains`
- `dwarf2lexblockcodeaddrcalc`

36.2.1 dwarf2typedefchains

Generates DWARF2 debugging information `typedef` statements.

**Syntax**

```
#pragma dwarf2typedefchains on | off | reset
```

**Remarks**

When this pragma is `on`, the compiler generates DWARF2 debugging information for type definitions.

By default, this option is `off`.

36.2.2 dwarf2lexblockcodeaddrcalc

Calculates the first and the last instruction in a lexical block by inspecting the entire lexical block start and end lines.

**Syntax**

```
#pragma dwarf2lexblockcodeaddrcalc on | off | reset
```

**Remarks**

When this pragma is `off`, the compiler looks for the instruction following the lexical block start and end but, does not emit the correct address range for unparenthesized loop expressions.

By default, this option is `off`.
36.3 Library and Linking Pragmas

This section describes the library and linking pragmas.

- `force_active`
- `section`

36.3.1 `force_active`

Deprecated.

Syntax

```c
#pragma force_active on | off | reset
```

Remarks

In source code, use `__declspec(force_export)`, `__attribute__((force_export))`, or `__attribute__((used))`.

In a linker command file, use the `FORCEACTIVE` command.

36.3.2 `section`

This sophisticated and powerful pragma lets you arrange compiled object code into predefined sections and sections you define.

**NOTE**

Deprecated only when used without an associated `__declspec(section)`. To avoid C++ parsing ambiguities and other possible inadvertent errors, use `__declspec(section)` instead.

Syntax

```c
#pragma section [ objecttype | permission ][iname][uname]
  [data_mode=datamode][code_mode=codemode]
```

Parameter

`objecttype`

specifies where types of object data are stored. It may be one or more of these values:
• code_type - executable object code
• data_type - non-constant data of a size greater than the size specified in the small data threshold option in the EPPC Target settings panel
• sdata_type - non-constant data of a size less than or equal to the size specified in the small data threshold option in the EPPC Target settings panel
• const_type - constant data of a size greater than the size specified in the small const data threshold option in the EPPC Target settings panel
• sconst_type - constant data of a size less than or equal to the size specified in the small const data threshold option in the EPPC Target settings panel
• all_types - all code and data

Specify one or more of these object types without quotes separated by spaces.

The CodeWarrior C/C++ compiler generates some of its own data, such as exception and static initializer objects, which are not affected by #pragma section.

**NOTE**

To classify character strings, the CodeWarrior C/C++ compiler uses the setting of the Make Strings Read Only checkbox in the EPPC Processor settings panel. If the checkbox is checked, character strings are stored in the same section as data of type const_type. If the checkbox is clear, strings are stored in the same section as data for data_type.

**permission**

specifies access permission. It may be one or more of these values:

• R - read only permission
• W - write permission
• X - execute permission

Specify one or more of these object types without quotes separated by spaces.

For more information on access permission, see "Section access permissions" on page 487.

**iname**

specifies the name of the section where the compiler stores initialized objects. Variables that are initialized at the time they are defined, functions, and character strings are examples of initialized objects.

The iname parameter may be of the form .abs.xxxxxxxx where xxxxxxxx is an 8-digit hexadecimal number specifying the address of the section.
specifies the name of the section where the compiler stores uninitialized objects. This parameter is required for sections that have data objects. The uname parameter value may be a unique name or it may be the name of any previous iname or uname section. If the uname section is also an iname section then uninitialized data is stored in the same section as initialized objects.

The special uname COMM specifies that uninitialized data will be stored in the common section. The linker will put all common section data into the ".bss" section. When the Use Common Section checkbox is checked in the EPPC Processor panel, COMM is the default uname for the .data section. If the Use Common Section checkbox is clear, .bss is the default name of .data section.

The uname parameter value may be changed. For example, you may want most uninitialized data to go into the .bss section while specific variables be stored in the COMM section.

Figure 36-1 shows an example where specific uninitialized variables are stored in the COMM section.

<table>
<thead>
<tr>
<th>Figure 36-1. Storing Uninitialized Data in the COMM Section</th>
</tr>
</thead>
</table>

```plaintext
#pragma push // save the current state
#pragma section ".data" "COMM"
int red;
int sky;
#pragma pop // restore the previous state
```

NOTE
You may not use any of the object types, data modes, or code modes as the names of sections. Also, you may not use predefined section names in the PowerPC EABI for your own section names.

data_mode=datamode

specifies the compiler for the kind of addressing mode to be used for referring to data objects for a section.

The permissible addressing modes for datamode are:

- near_abs - objects must be within the range -65,536 bytes to 65,536 bytes (16 bits on each side)
- far_abs - objects must be within the first 32 bits of RAM
- sda_rel - objects must be within a 32K range of the linker-defined small data base address
The sda_rel addressing mode may only be used with the ".sdata", ".sbss", ".sdata2", ".sbss2", ".PPC.EMB.sdata0", and ".PPC.EMB.sbss0" sections.

The default addressing mode for large data sections is far_abs. The default addressing mode for the predefined small data sections is sda_rel.

Specify one of these addressing modes without quotes.

\texttt{code\_mode=codemode}

specifies the compiler for the kind of addressing mode to be used for referring to executable routines of a section.

The permissible addressing modes for codemode are:

- \texttt{pc\_rel} - routines must be within plus or minus 24 bits of where \texttt{pc\_rel} is called from
- \texttt{near\_abs} - routines must be within the first 24 bits of memory address space
- \texttt{far\_abs} - routines must be within the first 32 bits of memory address space

The default addressing mode for executable code sections is \texttt{pc\_rel}.

Specify one of these addressing modes without quotes.

\textbf{NOTE}

All sections have a data addressing mode (\texttt{data\_mode=datamode}) and a code addressing mode (\texttt{code\_mode=codemode}). Although the CodeWarrior C/C++ compiler for PowerPC embedded allows you to store executable code in data sections and data in executable code sections, this practice is not encouraged.

\textbf{Remarks}

CodeWarrior compilers generate their own data, such as exception and static initializer objects, which the \texttt{#pragma section} statement does not affect.

\textbf{Section access permissions}

When you define a section by using \texttt{#pragma section}, its default access permission is read only. Changing the definition of the section by associating an object type with it sets the appropriate access permissions for you. The compiler adjusts the access permission to allow the storage of newly-associated object types while continuing to allow objects of previously-allowed object types. For example, associating \texttt{code\_type} with a section adds execute permission to that section. Associating \texttt{data\_type}, \texttt{sdata\_type}, or \texttt{sconst\_type} with a section adds write permission to that section.
Occasionally you might create a section without associating it with an object type. You might do so to force an object into a section with the __declspec keyword. In this case, the compiler automatically updates the access permission for that section to allow the object to be stored in the section, then issue a warning. To avoid such a warning, make sure to give the section the proper access permissions before storing object code or data into it. As with associating an object type to a section, passing a specific permission adds to the permissions that a section already has.

**Predefined sections and default sections**

When an object type is associated with the predefined sections, the sections are set as default sections for that object type. After assigning an object type to a non-standard section, you may revert to the default section with one of the forms in "Forms for #pragma section" on page 488.

The compiler predefined the sections in Figure 36-2.

![Figure 36-2. Predefined sections](image)

```plaintext
#pragma section code_type ".text" data_mode=far_abs code_mode=pc.rel
#pragma section data_type ".data" ".bss" data_mode=far_abs
code_mode=pc.rel
#pragma section const_type ".rodata" ".rodata" data_mode=far_abs
   code_mode=pc.rel
#pragma section sdata_type ".sdata" ".sbss" data_mode=sda.rel
   code_mode=pc.rel
#pragma section sconst_type ".sdata2" ".sbss2" data_mode=sda.rel
   code_mode=pc.rel
#pragma section ".PPC.EMB.sdata0" ".PPC.EMB.sbss0" data_mode=sda.rel
code_mode=pc.rel
#pragma section RX ".init" ".init" data_mode=far_abs code_mode=pc.rel
```

**NOTE**

The .PPC.EMB.sdata0 and .PPC.EMB.sbss0 sections are predefined as an alternative to the sdata_type object type. The .init section is also predefined, but it is not a default section. The .init section is used for startup code.

**Forms for #pragma section**

```plaintext
#pragma section ".name1"
```

This form simply creates a section called .name1 if it does not already exist. With this form, the compiler does not store objects in the section without an appropriate, subsequent #pragma section statement or an item defined with the __declspec keyword. If only one section name is specified, it is considered the name of the initialized object.
section, iname. If the section is already declared, you may also optionally specify the uninitialized object section, uname. If you know that the section must have read and write permission, use #pragma section RW .name1 instead, especially if you use the __declspec keyword.

#pragma section objecttype ".name2"

With the addition of one or more object types, the compiler stores objects of the types specified in the section .name2. If .name2 does not exist, the compiler creates it with the appropriate access permissions. If only one section name is specified, it is considered the name of the initialized object section, iname. If the section is already declared, you may also optionally specify the uninitialized object section, uname.

#pragma section objecttype

When there is no iname parameter, the compiler resets the section for the object types specified to the default section. Resetting the section for an object type does not reset its addressing modes. You must reset them.

When declaring or setting sections, you also can add an uninitialized section to a section that did not have one originally by specifying a uname parameter. The corresponding uninitialized section of an initialized section may be the same.

Forcing individual objects into specific sections

You may store a specific object of an object type into a section other than the current section for that type without changing the current section. Use the __declspec keyword with the name of the target section and put it next to the extern declaration or static definition of the item you want to store in the section.

Figure 36-3 shows examples.

**Figure 36-3. Using __declspec to Force Objects into Specific Sections**

__declspec(section ".data") extern int myVar;
#pragma section "constants"
__declspec(section "constants") const int myConst = 0x12345678

Using #pragma section with #pragma push and #pragma pop

You can use this pragma with #pragma push and #pragma pop to ease complex or frequent changes to sections settings.

See Figure 36-1 for an example.
NOTE
The #pragma pop does not restore any changes to the access permissions of sections that exist before or after the corresponding #pragma push.

36.4 Code generation Pragmas

This section describes the code generation pragmas.

- alignment_metrowerks
- altivec_codegen
- altivec_model
- altivec_pim_warnings
- altivec_vrsave
- cats
- disable_registers
- e500_floatingpoint
- e500v2_floatingpoint
- function_align
- gen_fsel
- gen_isel
- gprfloatcopy
- has8bytebitfields
- interrupt
- legacy_struct_alignment
- merge_floatconsts
- min_struct_align
- misaligned_mem_access
- no_register_save_helpers
- options
- pool_data
- ppc_lvxL_stvxl_errata
- profile
- read_only_switch_tables
- read_only_vtable_RTTI
- strict_ieee_fp
- switch_tables
- uchar_bool
- use_lmw_stmw
• ushort_wchar_t
• vec2x32float_align_4
• z4_16bit_cond_branch_errata_5116
• z4_mtlr_se_rfi_errata_26553

36.4.1 alignment_metrowerks

Determines which structure alignment policy to use.

Syntax
#pragma alignment_metrowerks on | off | reset

Remarks
This pragmas specifies which policy that #pragma options align=power will use for aligning data in structures. When this pragma is on, then the directive #pragma options align=power is equivalent to this directive #pragma options align=power_mw

When this pragma is off, then the directive #pragma options align=power is equivalent to this directive #pragma options align=power_gcc

36.4.2 altivec_codegen

Controls the use Power Architecture AltiVec™ instructions during optimization.

Syntax
#pragma altivec_codegen on | off | reset

Remarks
If this pragma is on, the compiler uses Power Architecture AltiVec instructions, if possible, during optimization. When this pragma is off, the pragma altivec_model is also set to off.

36.4.3 altivec_model
Controls the use Power Architecture AltiVec™ language extensions.

Syntax

```c
#pragma altivec_model on | off | reset
```

Remarks

If you enable this pragma, the compiler allows language extensions to take advantage of the AltiVec instructions available on some Power Architecture processors. When this pragma is `on` the compiler sets `#pragma altivec_codegen` to `on` and defines the `__VEC__` preprocessor directive.

### 36.4.4 `altivec_pim_warnings`

Controls how the compiler translates literal values for AltiVec vectors.

Syntax

```c
#pragma altivec_pim_warnings on | off | reset
```

Remarks

When this pragma is `on`, the compiler follows the syntax rules described in *AltiVec™ Technology Programming Interface Manual* ("PIM") to specify literal values for vector objects. This syntax specifies these requirements:

- vector values must be enclosed in parentheses
- vector values must be preceded by a type specifier

When this pragma is `off`, the compiler expects a style more consistent with C source code conventions:

- vector values must be enclosed in braces
- vector values do not need to be preceded by a type specifier

Listing 36-1 on page 536 shows an example.

Listing 36-1. Example of using `altivec_pim_warnings`

```c
#pragma altivec_pim_warnings on
vector signed int vsi1 = (__vector signed int)(1, 2, 3, 4);
#pragma altivec_pim_warnings off
vector signed int vsi2 = {1, 2, 3, 4};
```
36.4.5  altivec_vrsave

Controls which AltiVec™ registers to save to the stack between function calls.

Syntax

#pragma altivec_vrsave on | off | reset | allon

Parameter

allon

Tells the compiler to set all bits in the VRSAVE register.

Remarks

When generating instructions to call a function, the compiler uses the VRSAVE register's contents to determine which AltiVec registers should be saved to and restored from the stack.

When this pragma is on, the compiler determines which AltiVec registers a function uses. It generates instructions to record this information in the VRSAVE register.

When this pragma is off, the compiler clears the VRSAVE register and consequently does not save AltiVec registers on the stack.

When this pragma is set to allon, the compiler sets the VRSAVE register to specify that all AltiVec registers should be saved and restored on the stack.

36.4.6  cats

Controls the generation of relative .rela.* sections in the .elf file

Syntax

#pragma cats on | off | reset

Remarks

The default for this pragma is off for Freescale Power Architecture processors.
NOTE

Standard libraries should be rebuild using #pragma cats off in order to get rid of all .rela.* symbols in the internal_FLASH.elf file.

36.4.7 disable_registers

Controls compatibility for the ISO/IEC standard library function setjmp().

Syntax

#pragma disable_registers on | off | reset

Remarks

If this pragma is on, the compiler disables certain optimizations for any function that calls setjmp(). It also disables global optimization and does not store local variables and arguments in registers. These changes ensure that all local variables have correct values when the setjmp() function saves the processor state.

Use this pragma only if you are porting code that relies on this feature because it makes your program much larger and slower. In new code, declare a variable to be volatile if you expect its value to persist across setjmp() calls.

36.4.8 e500_floatingpoint

Generates single-precision floating point instructions for the Power Architecture e500 SPE (Signal Processing Unit) APU (Auxiliary Processing Unit).

Syntax

#pragma e500_floatingpoint on | off | reset

36.4.9 e500v2_floatingpoint

Generates double-precision floating point instructions for the Power Architecture e500v2 SPE (Signal Processing Unit) APU (Auxiliary Processing Unit).
Syntax

#pragma e500v2_floatingpoint on | off | reset

### 36.4.10 function_align

Aligns the executable object code of functions on a specified byte boundary.

**Syntax**

#pragma function_align 4 | 8 | 16 | 32 | 64 | 128 | reset

### 36.4.11 gen_fsel

Controls the use of the floating-point select instruction, `fsel`.

**Syntax**

#pragma gen_fsel on | off |

`number` | `always`

where `number` is a value from 1 to 255.

**Remarks**

The compiler uses this pragma to determine how often it generates the `fsel` instruction. The `number` argument specifies how aggressively the compiler should use this instruction, 1 is equivalent to "rarely" and 255 is equivalent to `always`. The `on` choice is equivalent to a value of 10.

### 36.4.12 gen_isel

Controls the use of the integer select instruction, `isel`.

**Syntax**

#pragma gen_isel on | off |

`number` | `always`
where *number* is a value from 1 to 255.

**Remarks**

The compiler uses this pragma to determine how often it generates the *isel* instruction. The *number* argument specifies how aggressively the compiler should use this instruction, 1 is equivalent to "rarely" and 255 is equivalent to *always*. The *on* choice is equivalent to a value of 10.

### 36.4.13  **gprfloatcopy**

Takes advantage of simpler alignment restrictions for copying floating point data.

**Syntax**

```
#pragma gprfloatcopy on | off | reset
```

**Remarks**

When this pragma is *on*, the compiler uses integer load and store instructions for memory-to-memory assignments for objects of type *double* and *float*, which improves the speed of memory-to-memory assignments of unaligned floating-point data. When this pragma is *off*, the compiler issues floating-point load and store instructions instead.

### 36.4.14  **has8bytebitfields**

Controls the use of bitfields that fit in the *long long* data type.

**Syntax**

```
#pragma has8bytebitfields on | off | reset
```

**Remarks**

When this pragma is *on*, the compiler allows bitfields in the *long long* data type. Such bitfields may occupy up to 64 bits (8 bytes). When this pragma is *off*, the compiler allows bitfields only in integer types of the same size or smaller than the *long* type.

**Listing 36-2. Example for pragma has8bytebitfields**

```
#pragma has8bytebitfields on
struct X {
    long long fielda : 12;
}
```
long long fieldb : 18;
long long fieldc : 32;
long long fieldd : 2;
};

36.4.15 interrupt

Deprecated. To avoid C++ parsing ambiguities and other possible inadvertent errors, use __declspec(interrupt) instead.

36.4.16 legacy_struct_alignment

Avoids the possibility of misaligned load or store instructions caused by promoting the alignment of global and local data objects to a minimum of 4 bytes.

Syntax

#pragma legacy_struct_alignment on | off | reset

Remarks

The default for this pragma is off for Freescale Power Architecture processors as the big endian systems do not crash and misalignment is rare.

36.4.17 merge_float_consts

Each floating point constant is placed in a unique variable such that the linker will merge floating point constants which have the same value. (The variable names are not legal C/C++ and are not accessible by the user). This option works with either small data in TOC on or off. This option minimizes TOC entry usage for programs which frequently use the same floating point constant in many different source files.

Syntax

#pragma merge_float_consts on|off

Remarks

The default for this pragma is off for Freescale Power Architecture processors.
36.4.18  min_struct_align

Increases aggregate alignments for better memory access.

Syntax

#pragma min_struct_align 4 | 8 | 16 | 32 | 64 | 128 | on | off
| reset

Remarks

When this pragma is off, the compiler aligns objects to their regular alignments. The default alignment is 4.

NOTE
This pragma only applies if the optimization level is greater than 0.

36.4.19  misaligned_mem_access

Controls how the compiler copies structures that are not aligned to 4-byte boundaries.

Syntax

#pragma misaligned_mem_access on | off | reset

Remarks

When this pragma is on, the compiler uses 4-byte load and store instructions to copy structures that are not aligned to 4-byte boundaries. By using these misaligned load and store instructions, the compiler improves runtime performance and reduces code size.

When this pragma is off, the compiler uses 1-, 2-, and 4-byte load and store instructions to copy structures that are aligned on corresponding boundaries.

However, misaligned load and store instructions on some Power Architecture processors give poor performance or even generate processor exceptions. For these processors, turn this pragma off. Desktop variants of the Power Architecture processor family do not have this limitation.
Consult the processor manufacturer's documentation for information on the processor's behavior when loading and storing 4-byte values that are not aligned to 4-byte boundaries.

The default for this pragma is on for processors that allow misaligned memory access. The default is off for processors that have limited misaligned memory access performance or generate an exception.

### 36.4.20 no_register_save_helpers

Controls the save and restore registers without calling helper functions

**Syntax**

```
#pragma no_register_save_helpers on | off | reset
```

### 36.4.21 options

Specifies how to align structure and class data.

**Syntax**

```
#pragma options align=
    alignment
```

**Parameter**

alignment

Specifies the boundary on which structure and class data is aligned in memory. Values for `alignment range from 1 to 16, or use one of the` following preset values:

<table>
<thead>
<tr>
<th>Table 36-1. Structs and Classes Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>If alignment is ...</strong></td>
</tr>
<tr>
<td>mac68k</td>
</tr>
<tr>
<td>mac68k4byte</td>
</tr>
<tr>
<td>power</td>
</tr>
</tbody>
</table>
Table 36-1. Structs and Classes Alignment (continued)

<table>
<thead>
<tr>
<th>If alignment is ...</th>
<th>The compiler ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>native</td>
<td>Aligns every field using the standard alignment.</td>
</tr>
<tr>
<td>packed</td>
<td>Aligns every field on a 1-byte boundary. It is not available in any panel. This alignment causes your code to crash or run slowly on many platforms. Use it with caution.</td>
</tr>
<tr>
<td>reset</td>
<td>Resets to the value in the previous #pragma options align statement.</td>
</tr>
</tbody>
</table>

NOTE
There is a space between options and align.

36.4.22 pool_data

Controls whether data larger than the small data threshold is grouped into a single data structure.

Syntax

```c
#pragma pool_data on | off | reset
```

Remarks

When this pragma is on the compiler optimizes pooled data. You must use this pragma before the function to which you apply it.

NOTE
Even if this pragma is on, the compiler will only pool the data if there is a performance improvement.

This pragma corresponds to the CodeWarrior IDE's Pool Data setting in the PowerPC ProcessorProperties > C/C++ Build > Settings > Tool Settings > PowerPC Compiler > Processor panel.

36.4.23 ppc_lvxl_stvxl_errata
Controls the instruction encoding for the lvxl and stvxl instructions on the Power Architecture 745 x processors to correct a bug in the processors.

**Syntax**

```
#pragma ppc_lvxl_stvxl_errata on | off | reset
```

### 36.4.24 profile

Controls the generation of extra object code for use with the CodeWarrior profiler.

**Syntax**

```
#pragma profile on | off | reset
```

**Remarks**

If you enable this pragma, the compiler generates code for each function that lets the CodeWarrior Profiler collect information on it.

This pragma corresponds to the CodeWarrior IDE's **Profiler Information** setting in the **PPC ProcessorProperties > C/C++ Build > Settings > Tool Settings > PowerPC Compiler > Processor** panel.

### 36.4.25 read_only_switch_tables

Controls where tables for switch statements are placed in object code.

**Syntax**

```
#pragma read_only_switch_tables on | off | reset
```

**Remarks**

This option specifies where the compiler places executable code addresses for switch statements. When this option is on, the compiler places these tables in a read-only section (.rodata), allowing the linker to place this object code in a ROM area instead of RAM.

When this option is off, the compiler places these switch tables in an object code section that is readable and writable (.data). Putting these tables in a read/write section allows relocation at runtime. The System V ABI, SuSE, YellowDog, and SDA PIC/PID application binary interfaces (ABIs) allow relocatable object code at runtime.
### 36.4.26 strict_ieee_fp

Controls generation of executable code that conforms to the IEEE floating point standard.

**Syntax**

```plaintext
#pragma strict_ieee_fp on | off | reset
```

**Remarks**

Disabling this option may improve performance but may change the results generated.

- **Use Fused Mult-Add/Sub**
  
  Uses a single instruction to do a multiply accumulate. This runs faster and generates slightly more accurate results than specified by IEEE, as it has an extra rounding bit between the multiply and the add/subtract).

- **Generate `fsel` instruction**
  
  The `fsel` instruction is not accurate for denormalized numbers, and may have issues related to unordered compares, but generally runs faster.

- **Assume Ordered Compares**
  
  Ignore the unordered issues when comparing floating point which allows converting:

  ```plaintext
  if (a <= b)
  into
  if (!(a > b))
  ```

### 36.4.27 read_only_vtable_RTTI

Controls placing vtable and RTTI in a ready-only section.

**Syntax**

```plaintext
#pragma read_only_vtable_RTTI on | off | reset
```

**Remarks**
If you enable this pragma, compiler places vtable and RTTI in a read-only section (.rodata/.sdata2), allowing the linker to place this object code in a ROM area instead of RAM. If you disable this pragma, the compiler places vtable and RTTI in an object code section that is readable and writable (.data/.sdata).

This pragma does not correspond to any panel setting. By default, this pragma is disabled.

36.4.28 switch_tables

Controls the generation of switch tables.

Syntax

#pragma switch_tables on | off | reset

Remarks

When on, the compiler translates switch statements into tables of addresses where each address in the list corresponds to a case statement. Using tables improves the performance of switch statements but may increase the size of the executable code if there are many case statements or if the case label values are not contiguous.

When off, the compiler translates switch statements into a series of comparisons, one comparison for each case statement.

36.4.29 uchar_bool

Controls the size of the _Bool and bool data types.

Syntax

#pragma uchar_bool on | off | reset

Remarks

When on, the compiler translates the _Bool data type in C99 (ISO/IEC 9899-1999) source code and the bool data type in C++ source code to 8 bit values. When off, these data types are 32 bits. Use this pragma only before any declarations.
When this pragma is off (boolean values are 32-bits), use bitfields to ensure that a boolean value is 8 bits. *Listing 36-3 on page 548 shows an example.*

**Listing 36-3. Example of overriding uchar_bool in a structure in C++**

```c
#pragma uchar_bool off /* Boolean values are 32 bits */
typedef struct
{
    bool sockclean:8 /* This value will only occupy 8 bits. */
} sockrec;
```

### 36.4.30 use_lmw_stmw

Controls the use of `lmw` and `stmw` instructions.

**Syntax**

```c
#pragma use_lmw_stmw on | off | reset
```

**Remarks**

Use of `lmw` and `stmw` may be slower on some processors.

### 36.4.31 ushort_wchar_t

Controls the size of `wchar_t`.

**Syntax**

```c
#pragma ushort_wchar_t on | off | reset
```

**Remarks**

When this pragma is `on`, `wchar_t` changes from 4-bytes to 2-bytes.

### 36.4.32 vec2x32float_align_4

Controls the alignment of type `__vec2x32float__`.

**Syntax**

```c
```
#pragma vec2x32float_align_4 on | off | reset

**Remarks**
When the pragma is on, type __vec2x32float__ is aligned on 4byte boundary.
The default value for this pragma is off.

### 36.4.33 z4_16bit_cond_branch_errata_5116

Controls the use of 16-bit conditional instructions.

**Syntax**
```
#pragma z4_16bit_cond_branch_errata_5116 on | off | reset
```

**Remarks**
When the pragma is on, 32-bit conditional instructions are used instead of 16-bit.
The default value for this pragma is off.

### 36.4.34 z4_mtlr_se_rfi_errata_26553

Ensures that there are at least three instructions between the mtlr and the se_rfi.

**Syntax**
```
#pragma z4_mtlr_se_rfi_errata_26553 on | off | reset
```

**Remarks**
When the pragma is on and you are using either __declspec(interrupt) or #pragma interrupt, the compiler ensures that there are at least three instructions between the mtlr and the se_rfi.

If your interrupt handler is written in function level assembler, compiler support only happens if you do not use the nofralloc directive. Standalone assembler does not include this support.
The default value for this pragma is off.
NOTE
Other forms of the `se_rfi` such as `se_rfci` and `se_rfdi` are also supported.

36.5 Optimization Pragmas

This section describes the optimization pragmas.

- aggressive_hoisting
- c9x_alias_by_type
- epilogue_helper_functions
- fp_contract
- fp_contract_aggressive
- ipa_rescopes_globals
- peephole
- peephole_enable_16bit_load_store_inst
- ppc_opt_defuse_mem_limit
- ppc_unroll_instructions_limit
- ppc_unroll_speculative
- processor
- prologue_helper_functions
- remove_frsp_aggressive
- schedule
- schedule_z760
- scheduling
- switch_op
- switch_cases_array
- volatileasm

36.5.1 aggressive_hoisting

Improves the number of variables that get hoisted out of a loop.

Syntax

```
#pragma aggressive_hoisting on | off | reset
```

Remarks
This pragma produces faster code and causes a slight increase in code size, especially when optimizing for size. In some cases, hoisting variables out of a loop when the loop does not have a lot of iterations can make your code slower.

The default value for this pragma is off.

### 36.5.2  c9x_alias_by_type

Allows back-end optimizations to use alias type information.

**Syntax**

```
#pragma c9x_alias_by_type on | off | reset
```

**Remarks**

When this pragma is on, the compiler's back-end optimizations take advantage of type information gathered during alias analysis. Turn this pragma on if your source code follows the type rules specified by the ISO/IEC 9899-1999 C standard ("C99"), section 6.5.

Turn this pragma off if your source code violates type rules. The information collected from source code that violates these rules might lead the compiler to apply its optimizations incorrectly.

This pragma does not have a corresponding IDE panel setting.

### 36.5.3  epilogue_helper_functions

Controls size optimization for function termination instructions.

**Syntax**

```
#pragma epilogue_helper_functions on | off | reset
```

**Remarks**

When this pragma is on, the compiler reduces the size of object code in function terminations. It performs this optimization by replacing several instructions for function termination with fewer calls to special functions that perform the same tasks. This optimization reduces executable code size but also slows the program's performance.
When this pragma is on, the compiler generates instructions that may appear as inconsistent information in a symbolic debugger when stepping through the end of a function.

This pragma does not correspond to any panel setting. By default, this pragma is off. The compiler turns this optimization on implicitly when size optimization is on and optimization level is equal to or greater than 2. For example, these pragma settings will also turn on epilogue helper function generation:

```
#pragma optimization_level 2 #pragma optimize_for_size on
```

### 36.5.4 fp_contract

Controls the SPE additional fused multiply-add instructions codegen, when -spe2_vector is selected.

**Syntax**

```
#pragma fp_contract on|off
```

**Remarks**

Floating point accuracy is maintained by turning off this optimization.

### 36.5.5 fp_contract_aggressive

Enables the peephole pattern to detect and convert the separate instructions.

**Syntax**

```
#pragma fp_contract_aggressive on|off|reset
```

**Remarks**

#pragma fp_contract_aggressive on can be used to further optimize multiply-add opportunities.

**NOTE**

Precision could be lost due to rounding issues.
36.5.6  ipa_rescopes_globals

Rescopes the application global variables, that are only used in one function, to local static. The change to static enables other optimizations that improve alias analysis and load/store optimizations.

**Syntax**

```c
#pragma ipa_rescopes_globals on | off
```

**Remarks**

Ensure that the following requirements are met to rescope the application global variables to local static:

- Program IPA is enabled in all application source files
- use of `#pragma ipa_rescopes_globals on` in all application source files (prefix file or with `-flag ipa_rescopes_globals on the commandline`
- `main()` is defined in one of the application files.
- It is not necessary, or even desirable, to have standard library, runtime or startup code compiled with program IPA and `ipa_rescopes_globals on`. However, it is important to have as many of your application sources as possible compiled with those options enabled.

**NOTE**

As the third party libraries generally do not access the application variables, these libraries can be kept in archive form.

For a simple example, compile/assemble your startup code without program IPA. Compile all of the application code with program IPA, `#pragma ipa_rescopes_globals on` and link the startup objects, your application objects and the library archives (For more details on Program IPA linking procedures, refer "Interprocedural Analysis").

For a complex example where the application sources are put into groups, compiled and then pre-built into several archives or partially linked objects and the build procedure cannot be matched with the simple example, following changes to the build procedure are suggested:

- Try to make the build setup as similar to the simple example as possible. This will help you identify if the code will benefit from `ipa_rescopes Globals` or you will need to
modify your source files to get a successful link. For more details, refer "Generating a successful link".

- All of your functions are not visible to the compiler at once during program IPA. It is possible that a defined global variable in your core files may be used by only one core file but might also be used in one of your application archives that you were unable to build the simple way. If this is true, `ipa_rescopes_globals` will rescope the variable and at link time, your application archive will not be able to find the variable and you will get an undefined symbol link error.

**NOTE**

If you get a successful link you do not have to make any further changes to the build or source.

**Generating a successful link**

Optimization prevents an improper build. If you do not get a successful link or you only get a few such link errors, identify the source file that defines the "undefined" symbol and try one of the following (in decreasing order of general preference):

- Move the definition of the symbol into the application archive. Symbols that are undefined do not get rescoped.
- Force the export of the symbol with `__declspec (force_export)`. Symbols that are exported do not get rescoped.
- Change the symbols to weak with `__declspec (weak)` by inserting before definition. Weak symbols do not get rescoped.
- Change the symbols to volatile. Volatile symbols do not get rescoped.

### 36.5.7 peephole

Controls the use peephole optimization.

**Syntax**

```c
#pragma peephole on | off | reset
```

**Remarks**

If you enable this pragma, the compiler performs *peephole optimizations*. These optimizations replace redundant or commonly occurring groups of instructions with smaller, more efficient groups of instructions.
### 36.5.8 peephole_enable_16bit_load_store_inst


**Syntax**

```c
#pragma peephole_enable_16bit_load_store_inst on | off | reset
```

**Remarks**

This peephole optimization will replace the 32 bit load/store instructions with 16 bit load/store instructions.

This pragma is on by default under size optimization and off under speed optimization.

Please note that this optimization is applicable only when VLE instruction set is enabled.

**Example**

Converts the below pattern

```c
e_stb r0,28(r3)
e_stb r0,32(r3)
e_stb r4,36(r3)
e_stb r4,40(r3)
e_stb r4,41(r3)
e_stb r4,42(r3)
e_stb r4,43(r3)
```

To

```c
e_add16i r3,r6,28
se_stb r0,0(r3)
se_stb r0,4(r3)
```
**Optimization Pragmas**

```
se_stb   r4,8(r3)
se_stb   r4,12(r3)
se_stb   r4,13(r3)
se_stb   r4,14(r3)
se_stb   r4,15(r3)
```

## 36.5.9  ppc_opt_defuse_mem_limit

Controls memory consumed by compiler optimizations on the host computer.

**Syntax**

```
#pragma ppc_opt_defuse_mem_limit on | off | reset | limit
```

**Parameter**

`limit`

Number of megabytes to use on the host computer when optimizing object code. The default value is 150, which specifies 150 megabytes.

**Remarks**

Some optimizations need a lot of memory on the host computer, especially when optimizing large functions that make many function calls or refer to many variables. This pragma controls how much memory these optimizations consume.

If `limit` is set too low, the compiler will not be able to complete some optimizations and will issue an error message.

**NOTE**

This pragma is to be used when users see the compiler error or warning that the compiler needs more memory to be allocated for usedef/defuse chain computation.

## 36.5.10  ppc_unroll_instructions_limit
Limits number of instructions in an unrolled loop to value.

**Syntax**

```plaintext
#pragma ppc_unroll_instructions_limit value | on | off
```

**Parameter**

`value`

Count limit of instructions. The default is 70.

**Remarks**

Use this pragma to specify the maximum number of instructions to place in an unrolled loop. The `opt_unroll_loops` pragma controls loop unrolling optimization.

When this pragma is `on`, the compiler uses the default value.

### 36.5.11 `ppc_unroll_speculative`

Controls speculative unrolling of counting loops which do not have fixed counts.

**Syntax**

```plaintext
#pragma ppc_unroll_speculative on | off
```

**Remarks**

The compiler uses the value specified with the `ppc_unroll_factor_limit` pragma to compute how many times to unroll eligible loops. The compiler adjusts the value specified with `ppc_unroll_factor_limit` so that it is equal to or less than the closest power of 2.

This optimization is only applied when:

- loop unrolling is turned on with the `opt_unroll_loops` pragma
- the loop iterator is a 32-bit value ( `int`, `long`, `unsigned int`, `unsigned long` )
- the loop's body has no conditional statements

If you enable this pragma, the loop unrolling factor is a power of 2, less than or equal to the value specified by the `ppc_unroll_factor_limit` pragma.

The `opt_unroll_loops` pragma controls all loop unrolling optimization. To check this setting, use `__option (ppc_unroll_speculative)` . By default, this pragma is `on` when loop unrolling is enabled.
36.5.12   processor

Specifies the scheduling model used for instruction scheduling optimization.

Syntax

```
#pragma processor
model
```

model

This argument is one of these choices:

```
401 | 403 | 405 | 505 | 5100 | 5200 | 555 | 56x | 601 | 602 | 603 | 603e | 604 | 604e
| 74x | 75x | 801 | 821 | 823 | 85x | 86x | 87x | 88x | 7400 | 744x | 745x | 82xx | 85xx |
| e300v1 | e500v1 | e500v2 | e600 | Zen | generic
```

36.5.13   prologue_helper_functions

Controls size optimization for function initialization instructions.

Syntax

```
#pragma prologue_helper_functions on | off | reset
```

Remarks

When this pragma is on, the compiler reduces the size of object code in function initialization. It performs this optimization by replacing several instructions for function initialization with fewer calls to special functions that perform the same tasks. This optimization reduces executable code size but also reduces the program's performance.

This pragma does not correspond to any panel setting. By default, this pragma is off. The compiler turns this optimization on implicitly when size optimization is on and optimization level is equal to or greater than 2. For example, these pragma settings will also turn on prologue helper function generation:

```
#pragma optimization_level 2 #pragma optimize_for_size on
```

36.5.14   remove_frsp_aggressive
Improves code optimization by transforming Power Architecture LFS and FRSP instructions into the equivalent FMR instructions.

**Syntax**

```
#pragma remove_frsp_aggressive on | off | reset
```

**Remarks**

This pragma allows the compiler more opportunity to use copy propagation optimizations to improve the generated code.

The default value for this pragma is `off`.

### 36.5.15 schedule

Specifies the use of instruction scheduling optimization.

**Syntax**

```
#pragma schedule once | twice | altivec | off
```

**Remarks**

This pragma lets you choose how many times the compiler passes object code through its instruction scheduler.

On highly optimized C code where loops were manually unrolled, running the scheduler once seems to give better results than running it twice, especially in functions that use the `register` specifier.

When the scheduler is run twice, it is run both before and after register colorizing. If it is only run once, it is only run after register colorizing.

This pragma does not correspond to any panel setting. The default value for this pragma is `twice`.

### 36.5.16 schedule z760

Controls the rearrangement of instructions to reduce the effects of instruction latency.

This scheduler is for e200 processors.
Optimization Pragmas

Syntax

#pragma schedule z760

Remarks

This pragma can be activated by:

1. selecting the Zen processor (from the Processor preference panel or by using the proc Zen command line compiler option)
2. specifying #pragma schedule z760 in the prefix file or on the Preprocessor preference panel. You can also pass -pragma "schedule z760" command line to the compiler, to activate the pragma.

This pragma can be used for all e200 dual issue cores.

The pragma does not correspond to any panel setting. By default, this pragma is off.

36.5.17 scheduling

Specifies the scheduling model used for instruction scheduling optimization.

Syntax

#pragma scheduling
   model | off

model

This argument is one of these choices:

401 | 403 | 405 | 505 | 509 | 5100 | 5200 | 555 | 56x | 601 | 602 | 603 | 603e | 604 | 604e | 74x | 75x | 801 | 821 | 823 | 85x | 86x | 87x | 88x | 7400 | 744x | 745x | 82xx | 85xx | e300v1 | e500v1 | e500v2 | e600 | Zen | generic

36.5.18 switch_op

Minimizes comparisons for "if" and "switch" statements.

Syntax

#pragma switch_op on | off
Remarks

When this pragma is on, the comparisons for "if" and "switch" statements are minimized. The compiler performs this optimization by using the constants specified within the switch case (or if) statements and based on internal heuristics that determine the cost.

By default, this pragma is off. The compiler turns this optimization on implicitly when the optimization level is equal to or greater than 2 and the user specifies #pragma switch_op on. For example, these pragma settings will also turn on switch optimization:

```
#pragma optimization_level 2
#pragma switch_op on
```

### 36.5.19 switch_cases_array

Optimizes a certain class of the switch statements so that a memory access to load the address, of the selected case in a switch statement, is eliminated.

**Syntax**

```
#pragma switch_cases_array on | off | reset
```

**Remarks**

Normally, a selected case in a switch statement is chosen by loading the address of the selected case from a table of addresses. In some cases it is desirable to select a case without loading the case address from a memory table of addresses.

This optimization takes a switch statement and creates a contiguous array of code representing each possible selectable case. The code for each case is identical in size. This implies that where no applicable case is specified, there are dummy cases consisting of NOP code. Dummy cases eventually branch to the default case. It also implies that applicable cases may be padded with NOPs to achieve uniform size among the cases. The effect of a contiguous array of case values is that the selected case is chosen by an index calculated by the multiplication of the case value and the uniform case size. No memory loads are required to select the proper case.

This pragma should be used with care since the possibility of huge code growth is possible (since dummy cases are inserted to fill gaps between the user specified cases). Ideally, this optimization could be used on switch statements that have case values closely related such that as few as possible dummy cases are added.

The pragma’s scope is a function, so all switch statements within a function are affected by this pragma.
36.5.20  volatileasm

Controls the optimization of inline assembly statements.

Syntax

#pragma volatileasm on | off | reset

Remarks

When this pragma is off, the compiler applies peephole and scheduling optimizations to inline assembly statements. If the pragma is on, the compiler does not optimize these statements.
Index

__ALTIVEC__ 419
__ASSEMBLER__ 408
__attribute__((deprecated)) 392
__attribute__((force_export)) 393
__attribute__((malloc)) 394
__attribute__((never_inline)) 397
__attribute__((noalias)) 394
__attribute__((noreturn)) 392
__attribute__((returns_twice)) 395
__attribute__((unused)) 395
__attribute__((used)) 396
__COMPILER__ 408
__COUNTER__ 408
__cplusplus 410
__CWBUILD__ 410
__CWCC__ 410
__DATE__ 411
__declspec(do_not_merge) 400
__declspec(final) 400
__declspec(force_export) 401
__declspec(interrupt) 401
__declspec(never_inline) 391
__declspec(no_linker_opts) 403
__declspec(section name) 403
__declspec(vle_off) 405
__declspec(vle_on) 405
__declspec(weak) 406
__declspec(do_not_merge) 400
__declspec(final) 400
__declspec(force_export) 401
__declspec(interrupt) 401
__declspec(never_inline) 391
__declspec(no_linker_opts) 403
__declspec(section name) 403
__FILE__ 412
__func__ 412
__FUNCTION__ 412
__ide_target() 413
__LINE__ 413
__MWERKS__ 414
__optlevel 414
__POWERPC__ 419
__PPCBROADWAY__ 420
__PPCGECKO__ 420
__PRETTY_FUNCTION__ 415
__PRETTY_FUNCTION__ Identifier 298
__profile__ 416
__STDC__ 416
__TARGETARCH__ 409
__TIME__ 417
__VERSION__ 409
__VERSION_NUM__ 409
.(location counter) 225
.ifeq if equal 173
.ifge if greater than or equal 175
.ifgt if greater than 175
.ifle if less than or equal 174
.iflt if less than 174
.ifne if not equal 174
#undef 166
-abi 132
-align 133
-altivec_move_block 133
-ansi 45
-ARM 47
-big 134
-bool 47
-c 91
-char 53
-code_merging 148
-codeaddr 111
-codegen 91
-common 134
-ConvertArchiveToPartialLink 112
-convertpaths 77
cwd 78
-D+ 79
dataaddr 113
defaults 54
define 79
dialect 48
disassemble 63
-E 80
-encoding 54
-enum 92
-EP 80
-ext 92
-far_near_addressing 149
-fatext 134
-flag 55
-fmt 107
-for_scoping 49
-format 107
-fp 135
-fp_contract 137
-fullLicenseSearch 104
-func_align 137
-g 106
-gap_fill 114
-gcc_extensions 56
-gccext 56
-gccincludes 80
-gdwarf-2 106
-genbinary 113
-gen-fsel 138
-heapaddr 114
-heapsize 115
-help 64
-I- 81
-I+ 81
-ide_erre 67
-include 82
-inline 95
-xtables 129

A

ABI Conformance 349
Absolute Value Functions 345
access_errors 433
Activating EC++ 373
Additional Small Data Sections 256
ADDR 226
aggressive_hoisting 550
aggressive_inline 503
AGGRESSIVE_MERGE 276
AGGRESSIVE_MERGE_FILES 276
AGGRESSIVE_MERGE_SECTIONS 277
align 193
ALIGN 227
ALIGNALL 227
Aligning a Variable Declaration 355
Aligning Data 354
alignment_metrowerks 535
Alignment Attribute Syntax 354
Alignment in a Structure Definition 355
Allocating Additional Heap Space 380, 386
altivec_codegen 535
altivec_model 535
altivec_pim_warnings 537
AltIVec™ Data Types 352
always_import 499
always_inline 433
ANSI_strict 427
arg_dep_lookup 434
ARM_conform 434
ARM_scoping 434
array_new_delete 435
ascii 191
asciz 191
asmpoundcomment 459
asmsemicolcomment 459
Assembler 153
Assembler Control Directives 193
Assembler Directives 341
Assembling VLE Instructions 365
Assembly Language Statements 153
Assembly Syntax 329
Associating Input Sections With Output Sections 216
auto_inline 435

B

BIN_FILE_TYPE 229
BIND 228
Bitfields 357
Board Initialization Code 388
bool 435

Boolean Type 350
Branch Prediction 332
bss 177
Buffer Manipulation 348
Building an SDA PIC/PID Application 360
Building a ROM Image 367
byte 188
BYTE 230
Byte-Reversing Functions 346

C

C++ Compiler 297
C++ Compiler Performance 297
C++-style Comments 282
C++-Style Comments 287
C++-Style Digraphs 288
c99 427
C99 Complex Data Types 290
C99 Extensions 284
c9x 428
c9x_alias_by_type 551
Case-Sensitive Identifiers 160
cats 537
C Compiler 281
Character Constants 161
Character Types 350
check_header_flags 489
CheckingPragmaSettings 423
Choosing an MSL Library 378
Choosing a Runtime Library 387
Choosing Which Functions to Inline 323
Closure 259
Code Generation Command-Line Options 131
Code Generation Pragmas 534
Code Merging 327
CodeWarrior Linker Command File 264
Command-Line for Power Architecture Processors 103
Command-Line Options for Diagnostic Messages 63
Command-Line Options for Language Translation 53
Command-Line Options for Library and Linking 89
Command-Line Options for Optimization 95
Command-Line Options for Preprocessing 77
Command-Line Options for Standard C++ Conformance 47
Command-Line Options for Standard C Conformance 45
Commands, Directives, and Keywords 224
Comments 163
Common Subexpression Elimination 315
Common Terms 43
Compatibility Conditional Directives 173
Compiler Architecture 35
Compiling VLE Instructions 365
Compound Literal Values 285
Conditional Preprocessor Directives 167
Configuring Command-Line Tools 39
Configuring Linker File to Several ROM Blocks 273
const_strings 460
Constants 160
Controlling Alignment 217
Controlling C99 Extensions 284
Controlling GCC Extensions 291
Controlling Standard C Conformance 281
Copy Propagation 316
cplusplus 436
cpp_extensions 437
cpp1x 437
Create New RAM Section 272
Create New RAM Segment 271
Create New ROM Section 269
Create New ROM Segment 269
Creating an LCF from Scratch 265
Creating a Precompiled File 306
Creating a Stack Frame 336
Creating Memory Gaps 219
Creating Statement Labels 333
Creating Symbols 220
Creating Unique Labels and Equates 204

data 177
Data Addressing 352
Data Alignment 164
Data Cache Manipulation 347
Data Declaration Directives 187
Data Representation 349
Dead Code Elimination 314
Dead Store Elimination 317
Deadstripping 264
Dead-Stripping 214
debug 178
Debugging Directives 197
Debugging Information Pragmas 527
debuginline 438
Declaration Specifications 391, 399
Declaration Specifications for Power Architecture Code 399
Declaring Variables by Address 283
def_inherited 439
defer_codegen 439
defer_defarg_parsing 439
Defining Macros 200
Defining Sections in the Output File 215
Defining the Target's Memory Map 215
Dependencies Among Source Files 311
Designated Initializers 285
Determining Which Settings Are Saved and Restored 425
Diagnostic Command-Line Options 105
Diagnostic Pragmas 525
Differences Between ISO C++ and EC++ 374
direct_destruction 440
direct_to_sym 440
Directories 164
disable_registers 538
DO_NOT_MERGE 277
DO_NOT_MERGE_FILES 277
DO_NOT_MERGE_SECTIONS 278
dollar_identifiers 460
dont_inline 440
dontreuse_strings 503
double 192
dwarf2lexblockcodeaddrcalc 527
dwarf2typedefchains 527

e500_floatingpoint 538
e500v2_floatingpoint 538
EC++ Specifications 375
ecplusplus 441
else 172
elseif 170
Embedded C++ 373
Embedded Warrior Library 380
Empty Arrays in Structures 288
endian 193
endif 170
endm 165
entry 342
enums_signed 505
enumssalwaysint 504
epilogue_helper_functions 551
equ 186, 344
equal_sign (=) 185
Equates 158
errno_name 506
error 194
EWL Naming Convention 381
EXCEPTION 230
Exception Handling 375
exceptions 441
EXCLUDEFILES 231
explicit_zero_data 506
export 499
EXPORTSTRTAB 231
EXPORTSYMTAB 232
Expressions 162
Expression Simplification 314
extended_errorcheck 467
Extensions to Standard C 281
Extensions to Standard C++ 298
Extensions to the Preprocessor 282
extern 184
EXTERNAL_SYMBOL 233
Extra C99 Keywords 287

F
faster_pch_gen 489
file 197
File-Level Optimization 310
File Name Extensions 44
File Operations 374
fill 188
flat_include 490
float 192
float_constants 507
Floating-Point 351
Floating-Point Constants 161
Floating-Point Directives 192
force_active 528
FORCEACTIVE 233
FORCEFILES 279
Forward Declarations of Static Arrays 294
fp_contract 552
fp_contract_aggressive 552
fralloc 342
ffree 343
fullpath_file 490
fullpath_prepdump 491
function 198
function_align 539
Function and Top-level Variable Declarations 311
Function-Level Inline Assembly 330
Function-Level Optimization 310

G
gcc_extensions 461
GCC Extensions 291, 303
GCC - Style Inline Assembly 331
gen_fsel 539
gen_isel 539
Getting Help 42
global 184
global_optimizer 513
GNU Compatibility 207
GNU Compatible Syntax option 207
gprfloatcopy 540
GROUP 234

H
has8bytebitfields 540
Hexadecimal Floating-Point Constants 288
How to Rebuild the EWL Libraries 383
How to Relocate Code in RAM 275

I
if 168
ifdef 168
ifnc 170
ifndef 169
ignore_oldstyle 428
Implementation-Defined Behavior 301
Implicit Return From main() 286
import 500
IMPORTSTRTAB 235
IMPORTSYMTAB 236
inchin 188
include 194
INCLUDE 236
INCLUDEDWARF 237
incompatible_return_small_structs 525
incompatible_sfpe_double_params 526
INIT 278
Initializing Automatic Arrays and Structures 291
inline_bottom_up 442
inline_bottom_up_once 443
inline_depth 443
inline_max_auto_size 444
inline_max_size 445
inline_max_total_size 445
Inline-Assembly for Power Architecture Build Tools 329
Inlining 323
Inlining Techniques 325
instmgr_file 507
Integer Constants 160
Integer Directives 187
Integer Types 351
Intermediate Optimizations 309, 313
internal 445
INTERNAL_LOAD 237
INTERNAL_SYMBOL 238
Internal and External Addressing Modes 362
Internal and External Segments and References 358
Interprocedural Analysis 309
interrupt 541
Intrinsic Functions 345
Introduction 35
Invalid Pragmas 426
Invoking Command-Line Tools 41
Invoking Interprocedural Analysis 310
Invoking Macros 205
ipa 514
ipa_inline_max_auto_size 514
ipa_not_complete 515
ipa_rescopesGlobals 553
iso_templates 446

K
KEEP 238
keepcomments 491
L

Labels 155
Language Related Issues 375
LCF 264
legacy_struct_alignment 541
lib_export 501
Libraries 374
Libraries and Support Code 377
Library and Linking Command-Line Options 110
Library and Linking Pragmas 528
Library-Related Issues 376
line 199
line_prepdump 491
Linker 213
Linker Architecture 37
Linker Command File Commands 276
Linker Command File Operator Precedence 224
Linker Command File Support 224
Linker Command File Syntax 220
Linker-defined Symbols 359
Linker for Power Architecture Processors 253
Linker Generated Symbols 263
Linker Map File 259
Linking a ROM Image 368
Linking Assembly Files 361
Linking Binary Files 254
Linking VLE Object Code 367
Live Range Splitting 318
LOAD 238
load_store_elimination 516
Localization 374
Local Labels 156, 296
long 189
LONG 239
longlong 508
longlong_enums 508
Loop-Invariant Code Motion 320
Loop Unrolling 322
Low-Level Processor Synchronization 345

M

machine 343
macro 165
macro_prepdump 492
Macro Definition Syntax 201
Macro Directives 164
Macros 200
Main Standard Libraries 377
Manipulating the Contents of a Variable or Register 347
mark 461
Math Functions 348
MAX_BIN_GAP 241
maxerrorcount 468
MEMORY 239
Memory Map 262
merge_float_consts 541
message 469
mexit 165
min_enum_size 509
min_struct_align 542
Minimum and Maximum Operators 296
misaligned_mem_access 542
Modifications to the Section Pragma 362
mpwc_newline 462
mpwc_relax 462
msg_show_lineref 492
msg_show_realfref 492
multibyteaware 463
multibyteaware_preserve_literals 464

N

Naming Conventions 103
new_mangler 447
NEXT 241
no_conststringconv 447
NO_INIT_BSS 241
no_register_save_helpers 543
no_static_dtors 447
NO_TRAILING_BSS_IN_BIN_FILES 245
nofralloc 444
Non-constant Static Data Initialization 286
Non-Local Labels 156
Non-Standard Keywords 283
Normal, Record, and Overflow Forms 333
nosyminline 448
notonce 493

O

OBJECT 245
offset 179
old_friend_lookup 448
old_pods 449
old pragmatic_once 493
old_vtable 449
Omitted Operands in Conditional Expressions 294
once 493
only_std_keywords 429
opt_classresults 450
opt_common_subs 516
opt_dead_assignments 517
opt_dead_code 517
opt_lifetimes 517
opt_loop_invariants 518
opt_propagation 518
opt_strength_reduction 518
opt_strength_reduction_strict 519
opt_unroll_loops 519
opt_vectorize_loops 520
optimization_level 520
Optimization Command-Line Options 148
Optimization Pragmas 550
optimize_for_size 521
optimizewithasm 521
option 196
Option Formats 43
options 543
opword 344
org 195

public 184

R
read_only_switch_tables 545
read_only_vtable_RTTI 546
readonly_strings 510
Redefining Macros 293
REF_INCLUDE 246
Referring to Assembly, C, and C++ Variables 335
Referring to Labels 339
Referring to Registers 338
Referring to the Number of Arguments 205
Referring to Variables in Instruction Operands 337
Referring to Variables in Memory 337
REGISTER 246
Relocatable Labels 157
Relocating Code and Data in External MRAM 272
Relocating Code and Data in Internal RAM 270
Relocating Code in ROM 267
Relocating Function in ROM 268
remove_frsp_aggressive 558
required_prototypes 430
Required Libraries and Source Code Files 385
reverse_bitfields 510
rodata 177
ROMADDR 247
ROM Image Addresses 369
RTTI 451
Runtime Libraries 385

S
Saving and Restoring Pragma Settings 424
sbss 178
schedule 559
schedule 2760 559
scheduling 560
Scope Control Directives 183
sdata 177
sdata2 177
section 180, 528
Section Control Directives 176
Section Layout 261
SECTIONS 248
set 185
Setting CodeWarrior Environment Variables 39
Setting the Floating-Point Environment 346
Setting the PATH Environment Variable 40
short 189
SHORT 249
SHORTEN_NAMES_FOR_TOR_101 279
show_error_filestack 470
showmessagenumber 469
simple_prepdump 496
size 199
SIZEOF 250
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIZEOF_ROM</td>
<td>250</td>
</tr>
<tr>
<td>Small Data Area PIC/PID Support</td>
<td>357</td>
</tr>
<tr>
<td>space 190</td>
<td></td>
</tr>
<tr>
<td>space_prepdump 496</td>
<td></td>
</tr>
<tr>
<td>Specifying A Single ROM Block</td>
<td>369</td>
</tr>
<tr>
<td>Specifying Constant Data Location</td>
<td>373</td>
</tr>
<tr>
<td>Specifying Inline Assembly Statements</td>
<td>330</td>
</tr>
<tr>
<td>Specifying Jump Table Location</td>
<td>372</td>
</tr>
<tr>
<td>Specifying Link Order in the IDE</td>
<td>213</td>
</tr>
<tr>
<td>Specifying Memory Area Locations and Sizes</td>
<td>218</td>
</tr>
<tr>
<td>Specifying RAM addr_mode</td>
<td>364</td>
</tr>
<tr>
<td>Specifying ROM addr_mode</td>
<td>363</td>
</tr>
<tr>
<td>Specifying Several ROM Blocks</td>
<td>370</td>
</tr>
<tr>
<td>Specifying Source File Locations</td>
<td>103</td>
</tr>
<tr>
<td>srcrelincludes 497</td>
<td></td>
</tr>
<tr>
<td>Standard and Non-Standard Template Parsing</td>
<td>299</td>
</tr>
<tr>
<td>Statement-Level Inline Assembly</td>
<td>331</td>
</tr>
<tr>
<td>Statements in Expressions</td>
<td>292</td>
</tr>
<tr>
<td>Statement Syntax</td>
<td>154</td>
</tr>
<tr>
<td>store_object_files 511</td>
<td></td>
</tr>
<tr>
<td>Strength Reduction 321</td>
<td></td>
</tr>
<tr>
<td>strict_ieee_fp 546</td>
<td></td>
</tr>
<tr>
<td>strictheaderchecking 522</td>
<td></td>
</tr>
<tr>
<td>String Directives</td>
<td>190</td>
</tr>
<tr>
<td>Structure Member</td>
<td>356</td>
</tr>
<tr>
<td>Supported Extensions 208</td>
<td></td>
</tr>
<tr>
<td>suppress_init_code 452</td>
<td></td>
</tr>
<tr>
<td>suppress_warnings 470</td>
<td></td>
</tr>
<tr>
<td>switch_cases_array 561</td>
<td></td>
</tr>
<tr>
<td>switch_op 560</td>
<td></td>
</tr>
<tr>
<td>switch_tables 547</td>
<td></td>
</tr>
<tr>
<td>sym 470</td>
<td></td>
</tr>
<tr>
<td>Symbol Definition Directives</td>
<td>185</td>
</tr>
<tr>
<td>Symbols 155</td>
<td></td>
</tr>
<tr>
<td>Syntax 153</td>
<td></td>
</tr>
<tr>
<td>Syntax for Attribute Specifications</td>
<td>392</td>
</tr>
<tr>
<td>Syntax for Declaration Specifications</td>
<td>391, 399</td>
</tr>
<tr>
<td>syspath_once 497</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td></td>
</tr>
<tr>
<td>template_depth 452</td>
<td></td>
</tr>
<tr>
<td>Templates 374</td>
<td></td>
</tr>
<tr>
<td>text 176</td>
<td></td>
</tr>
<tr>
<td>text_encoding 464</td>
<td></td>
</tr>
<tr>
<td>text_vle 178</td>
<td></td>
</tr>
<tr>
<td>textequ 186</td>
<td></td>
</tr>
<tr>
<td>The __builtin_constant_p() Operator</td>
<td>294</td>
</tr>
<tr>
<td>The __builtin_expect() Operator</td>
<td>295</td>
</tr>
<tr>
<td>The sizeof() Operator</td>
<td>292</td>
</tr>
<tr>
<td>The typeof() Operator</td>
<td>293</td>
</tr>
<tr>
<td>Third Party Standard Libraries</td>
<td>380</td>
</tr>
<tr>
<td>thread_safe_init 452</td>
<td></td>
</tr>
<tr>
<td>Trailing Commas in Enumerations</td>
<td>284</td>
</tr>
<tr>
<td>trigraphs 465</td>
<td></td>
</tr>
<tr>
<td>type 200</td>
<td></td>
</tr>
<tr>
<td>Typedef Declaration 356</td>
<td></td>
</tr>
<tr>
<td>Type Definitions</td>
<td>312</td>
</tr>
<tr>
<td>U</td>
<td></td>
</tr>
<tr>
<td>uchar_bool 547</td>
<td></td>
</tr>
<tr>
<td>Unique LCF Examples 273</td>
<td></td>
</tr>
<tr>
<td>Unnamed Arguments 282</td>
<td></td>
</tr>
<tr>
<td>Unnamed Structures and Enumerations in C 313</td>
<td></td>
</tr>
<tr>
<td>unsigned_char 466</td>
<td></td>
</tr>
<tr>
<td>Unsuffixed Decimal Literal Values 290</td>
<td></td>
</tr>
<tr>
<td>Unsupported Extensions 210</td>
<td></td>
</tr>
<tr>
<td>Unsupported Language Features</td>
<td>375</td>
</tr>
<tr>
<td>unused 471</td>
<td></td>
</tr>
<tr>
<td>Updating a Precompiled File Automatically 307</td>
<td></td>
</tr>
<tr>
<td>use_lmw_stmw 548</td>
<td></td>
</tr>
<tr>
<td>Uses for SDA PIC/PID 360</td>
<td></td>
</tr>
<tr>
<td>ushort wchar_t 548</td>
<td></td>
</tr>
<tr>
<td>Using a Precompiled File</td>
<td>306</td>
</tr>
<tr>
<td>Using Build Tools on the Command Line 39</td>
<td></td>
</tr>
<tr>
<td>Using CodeWarrior IDE 254</td>
<td></td>
</tr>
<tr>
<td>Using Command-Line 256</td>
<td></td>
</tr>
<tr>
<td>Using Comments 334</td>
<td></td>
</tr>
<tr>
<td>Using Console I/O 379</td>
<td></td>
</tr>
<tr>
<td>Using Immediate Operands 340</td>
<td></td>
</tr>
<tr>
<td>Using Local Variables and Arguments 335</td>
<td></td>
</tr>
<tr>
<td>Using Macro Arguments 203</td>
<td></td>
</tr>
<tr>
<td>Using Pragmas 423</td>
<td></td>
</tr>
<tr>
<td>Using the .macro Directive 201</td>
<td></td>
</tr>
<tr>
<td>Using the #define Directive 206</td>
<td></td>
</tr>
<tr>
<td>Using the Embedded Warrior Libraries 381</td>
<td></td>
</tr>
<tr>
<td>Using the Instance Manager 298</td>
<td></td>
</tr>
<tr>
<td>Using the Main Standard Libraries 378</td>
<td></td>
</tr>
<tr>
<td>Using the Preprocessor 335</td>
<td></td>
</tr>
<tr>
<td>Using Variable Names as Memory Locations 339</td>
<td></td>
</tr>
<tr>
<td>V</td>
<td></td>
</tr>
<tr>
<td>Variable Argument Macros 286</td>
<td></td>
</tr>
<tr>
<td>Variable-Length Arrays 289</td>
<td></td>
</tr>
<tr>
<td>Variable Length Encoding 364</td>
<td></td>
</tr>
<tr>
<td>vec2x32float_align_4 548</td>
<td></td>
</tr>
<tr>
<td>Void and Function Pointer Arithmetic 294</td>
<td></td>
</tr>
<tr>
<td>Void Return Statements 295</td>
<td></td>
</tr>
<tr>
<td>volatileasm 562</td>
<td></td>
</tr>
<tr>
<td>W</td>
<td></td>
</tr>
<tr>
<td>warn_any_ptr_int_conv 473</td>
<td></td>
</tr>
<tr>
<td>warn_emptydec1 474</td>
<td></td>
</tr>
<tr>
<td>warn_extracomma 474</td>
<td></td>
</tr>
<tr>
<td>warn_filenamecaps 475</td>
<td></td>
</tr>
<tr>
<td>warn_filenamecaps_system 476</td>
<td></td>
</tr>
<tr>
<td>warn_hiddenlocals 476</td>
<td></td>
</tr>
<tr>
<td>warn_hidevirtual 454</td>
<td></td>
</tr>
<tr>
<td>warn_illpragma 477</td>
<td></td>
</tr>
<tr>
<td>warn_illtokenpasting 477</td>
<td></td>
</tr>
<tr>
<td>warn_illunionmembers 478</td>
<td></td>
</tr>
</tbody>
</table>
warn_impl_f2i_conv 478
warn_impl_i2f_conv 479
warn_impl_s2u_conv 479
warn_implicitconv 480
warn_largeargs 481
warn_missingreturn 481
warn_no_explicit_virtual 454
warn_no_side_effect 482
warn_no_typename 455
warn_notinlined 456
warn_padding 482
warn_pch_portability 483
warn_possunwant 483
warn_ptr_int_conv 484
warn_resultnotused 485
warn_structclass 456
warn_undefmacro 485
warn_uninitializedvar 486
warn_unusedarg 486
warn_unusedvar 487
warning 472
warning_errors 473
wchar_type 457
What Can be Precompiled 305
WRITES0COMMENT 251

Z
z4_16bit_cond_branch_errata_5116 549
z4_mtlr_se_rfi_errata_26553 549
z760 559
Information in this document is provided solely to enable system and software implementers to use Freescale products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. Freescale reserves the right to make changes without further notice to any products herein.

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

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

© 2005–2016 Freescale Semiconductor, Inc. All rights reserved.