CodeWarrior Development Studio for StarCore 3900FP DSP Architectures
SC3000 Linker Reference Manual

Document Number: CWSCLINREF
Rev. 10.9.0, 11/2015
## Contents

<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Chapter 1</strong></td>
<td></td>
</tr>
<tr>
<td>1.1</td>
<td>About this document</td>
<td>11</td>
</tr>
<tr>
<td>1.2</td>
<td>Accompanying documentation</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td><strong>Chapter 2</strong></td>
<td></td>
</tr>
<tr>
<td>2.1</td>
<td>Linker configuration tasks</td>
<td>13</td>
</tr>
<tr>
<td>2.1.1</td>
<td>How to create a Linker Command File (LCF)</td>
<td>13</td>
</tr>
<tr>
<td>2.1.2</td>
<td>How to Make LCF Compatible for Flexible Startup</td>
<td>16</td>
</tr>
<tr>
<td>2.1.2.1</td>
<td>Constraints With Flexible Startup Configuration</td>
<td>17</td>
</tr>
<tr>
<td>2.1.3</td>
<td>How to Define and use a Custom set of Tasks</td>
<td>18</td>
</tr>
<tr>
<td>2.1.4</td>
<td>How to Setup Virtual Trace Buffer (VTB) Using LCF</td>
<td>20</td>
</tr>
<tr>
<td>2.1.5</td>
<td>How to Setup Cache</td>
<td>21</td>
</tr>
<tr>
<td>2.1.6</td>
<td>How to Define Physical Memory Address Space of Target Architecture</td>
<td>22</td>
</tr>
<tr>
<td>2.1.7</td>
<td>How to define stack and heap memory area in LCF</td>
<td>23</td>
</tr>
<tr>
<td>2.1.7.1</td>
<td>Example for Multi-Core Architectures</td>
<td>24</td>
</tr>
<tr>
<td>2.1.8</td>
<td>How to Define Physical Memory Layout for a Multi-core Application</td>
<td>26</td>
</tr>
<tr>
<td>2.1.9</td>
<td>How to Modify the LCF When Each Core Runs Different Code</td>
<td>29</td>
</tr>
<tr>
<td>2.1.10</td>
<td>How to Define the Shared Memory</td>
<td>33</td>
</tr>
<tr>
<td>2.1.11</td>
<td>How to Create Virtual Memory for Private Sections</td>
<td>35</td>
</tr>
<tr>
<td>2.1.12</td>
<td>How to Define Virtual Memory for Read-Write-Execute (RWX) Access</td>
<td>38</td>
</tr>
<tr>
<td>2.2</td>
<td>General linker tasks</td>
<td>40</td>
</tr>
<tr>
<td>2.2.1</td>
<td>How to Reserve Physical Memory Area</td>
<td>40</td>
</tr>
<tr>
<td>2.2.2</td>
<td>How to Define Private Data Sections for Multiple Cores</td>
<td>41</td>
</tr>
<tr>
<td>2.2.3</td>
<td>How to troubleshoot linker error messages</td>
<td>43</td>
</tr>
<tr>
<td>2.2.4</td>
<td>How to build expressions in the SC3000 LCF</td>
<td>43</td>
</tr>
<tr>
<td>2.2.5</td>
<td>How to Map Virtual Memory Areas to Physical Memory Address Space</td>
<td>45</td>
</tr>
<tr>
<td>2.2.6</td>
<td>How to specify the content of virtual memory areas</td>
<td>47</td>
</tr>
<tr>
<td>Section number</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>2.2.7</td>
<td>How to Share Code and Data Partially Among Different Cores</td>
<td>48</td>
</tr>
<tr>
<td>2.2.8</td>
<td>How to Limit Code and Data Visibility at Core Level</td>
<td>54</td>
</tr>
<tr>
<td>2.2.9</td>
<td>How to Define Unlikely Block of Code as Private Block of Code in a Multi-core Application</td>
<td>58</td>
</tr>
<tr>
<td>2.2.9.1</td>
<td>Scenario 1: True Private Code Model</td>
<td>58</td>
</tr>
<tr>
<td>2.2.9.2</td>
<td>Scenario 2: Code Partially Shared Among Different Cores</td>
<td>58</td>
</tr>
<tr>
<td>2.2.10</td>
<td>How to Run Multiple Tasks on the Same Core</td>
<td>59</td>
</tr>
<tr>
<td>2.2.11</td>
<td>How to Make Code or Data Sections Visible to a Subset of Cores</td>
<td>62</td>
</tr>
<tr>
<td>2.2.12</td>
<td>How to place a symbol in an another section in LCF</td>
<td>62</td>
</tr>
<tr>
<td>2.2.13</td>
<td>How to Handle C++ Templates in Multi-core Applications</td>
<td>64</td>
</tr>
<tr>
<td>2.2.14</td>
<td>How to Check Local Symbols Addresses</td>
<td>65</td>
</tr>
<tr>
<td>2.2.15</td>
<td>How to use KEEP Directive</td>
<td>66</td>
</tr>
<tr>
<td>2.2.16</td>
<td>How to reserve an MMU descriptor ID</td>
<td>66</td>
</tr>
</tbody>
</table>

**Chapter 3**

**Concepts**

<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Linker configuration concepts</td>
<td>69</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Understanding linker terminology</td>
<td>69</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Understanding SC3000 LCF syntax</td>
<td>70</td>
</tr>
<tr>
<td>3.1.2.1</td>
<td>Using naming conventions</td>
<td>70</td>
</tr>
<tr>
<td>3.1.2.2</td>
<td>Specifying integers</td>
<td>71</td>
</tr>
<tr>
<td>3.1.2.3</td>
<td>Specifying symbol names</td>
<td>71</td>
</tr>
<tr>
<td>3.1.2.4</td>
<td>Specifying global directives</td>
<td>71</td>
</tr>
<tr>
<td>3.1.2.5</td>
<td>Specifying target architecture</td>
<td>72</td>
</tr>
<tr>
<td>3.1.2.6</td>
<td>Defining tasks</td>
<td>72</td>
</tr>
<tr>
<td>3.1.2.7</td>
<td>Defining virtual memory and output sections</td>
<td>73</td>
</tr>
<tr>
<td>3.1.2.8</td>
<td>Configuring the virtual memory</td>
<td>74</td>
</tr>
<tr>
<td>3.1.2.9</td>
<td>Creating an output section</td>
<td>75</td>
</tr>
<tr>
<td>3.1.2.10</td>
<td>Configuring the Physical Memory</td>
<td>77</td>
</tr>
<tr>
<td>3.1.2.11</td>
<td>Specifying Address Translation Construct</td>
<td>78</td>
</tr>
<tr>
<td>3.1.2.12</td>
<td>Linking self-contained libraries</td>
<td>79</td>
</tr>
<tr>
<td>Section number</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>----------------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>3.1.3</td>
<td>Understanding Cache Optimization in SC3000 Linker</td>
<td>79</td>
</tr>
<tr>
<td>3.1.4</td>
<td>Understanding Flexible Startup Configuration</td>
<td>81</td>
</tr>
<tr>
<td>3.1.4.1</td>
<td>Changes Made to Support Flexible Startup Configuration</td>
<td>82</td>
</tr>
<tr>
<td>3.2</td>
<td>General linker concepts</td>
<td>82</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Understanding startup environment</td>
<td>83</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Understanding Flexible Segment Programming Model</td>
<td>84</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Understanding L1 Defense</td>
<td>84</td>
</tr>
<tr>
<td>4.1</td>
<td>Configuration error messages</td>
<td>87</td>
</tr>
<tr>
<td>4.1.1</td>
<td>EID_ARCH_INCOMPATIBLE_WITH_DIRECTIVE</td>
<td>87</td>
</tr>
<tr>
<td>4.1.2</td>
<td>EID_EXPR_SECNAME_CANNOT_EVAL</td>
<td>87</td>
</tr>
<tr>
<td>4.1.3</td>
<td>EID_FAIL_LAYOUT</td>
<td>88</td>
</tr>
<tr>
<td>4.1.4</td>
<td>EID_FAIL_VIRTUAL_LAYOUT</td>
<td>88</td>
</tr>
<tr>
<td>4.1.5</td>
<td>EID_FORCE_VALUE_TO_1</td>
<td>88</td>
</tr>
<tr>
<td>4.1.6</td>
<td>EID_INCONSISTENT_SYMADDR</td>
<td>89</td>
</tr>
<tr>
<td>4.1.7</td>
<td>EID_LAYOUT_UNRESOLVED</td>
<td>89</td>
</tr>
<tr>
<td>4.1.8</td>
<td>EID_LNK_SECTION_TYPE_UNKNOWN</td>
<td>89</td>
</tr>
<tr>
<td>4.1.9</td>
<td>EID_MORE_AUTO_LAYOUT</td>
<td>90</td>
</tr>
<tr>
<td>4.1.10</td>
<td>EID_MEM_CANNOT_FINAL</td>
<td>90</td>
</tr>
<tr>
<td>4.1.11</td>
<td>EID_MEM_INCONSISTENT</td>
<td>91</td>
</tr>
<tr>
<td>4.1.12</td>
<td>EID_MEM_MULTI_AT</td>
<td>91</td>
</tr>
<tr>
<td>4.1.13</td>
<td>EID_MEM_NOT_FULLY_SPEC_RESERVE</td>
<td>91</td>
</tr>
<tr>
<td>4.1.14</td>
<td>EID_MEM_SMALL_MB</td>
<td>92</td>
</tr>
<tr>
<td>4.1.15</td>
<td>EID_MEM_UNDEF</td>
<td>92</td>
</tr>
<tr>
<td>4.1.16</td>
<td>EID_MEM_UNDEF_ADDR</td>
<td>92</td>
</tr>
<tr>
<td>4.1.17</td>
<td>EID_MEM_VIR_NO_PHY</td>
<td>93</td>
</tr>
<tr>
<td>4.1.18</td>
<td>EID_ASSERT_FAIL</td>
<td>93</td>
</tr>
<tr>
<td>4.1.19</td>
<td>EID_ATTMMU_SIZE_UNSPECIFIED</td>
<td>93</td>
</tr>
<tr>
<td>Section number</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>----------------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>4.1.20</td>
<td>EID_MATT_MAP11_ORG</td>
<td>94</td>
</tr>
<tr>
<td>4.1.21</td>
<td>EID_MATT_SPEC</td>
<td>94</td>
</tr>
<tr>
<td>4.1.22</td>
<td>EID_MATT_V1ToPn</td>
<td>94</td>
</tr>
<tr>
<td>4.1.23</td>
<td>EID_MATT_VP_UNMATCH</td>
<td>94</td>
</tr>
<tr>
<td>4.1.24</td>
<td>EID_MATT_WX</td>
<td>95</td>
</tr>
<tr>
<td>4.1.25</td>
<td>EID_MATTTS_OVERNUMBER</td>
<td>95</td>
</tr>
<tr>
<td>4.1.26</td>
<td>EID_MEM_ADDR_SIZE_UNALIGN</td>
<td>95</td>
</tr>
<tr>
<td>4.1.27</td>
<td>EID_MEM_EMPTY</td>
<td>96</td>
</tr>
<tr>
<td>4.1.28</td>
<td>EID_MEM_OVERLAP</td>
<td>96</td>
</tr>
<tr>
<td>4.1.29</td>
<td>EID_MEM_PLACE_INTO_reserve</td>
<td>96</td>
</tr>
<tr>
<td>4.1.30</td>
<td>EID_MEM_REAL_OVERLAP</td>
<td>97</td>
</tr>
<tr>
<td>4.1.31</td>
<td>EID_MEM_REAL_OVERLAP_1</td>
<td>97</td>
</tr>
<tr>
<td>4.1.32</td>
<td>EID_MULTI_ATTMMU_SIZE</td>
<td>97</td>
</tr>
<tr>
<td>4.1.33</td>
<td>EID_PHY_CANNOT_LAYOUT</td>
<td>97</td>
</tr>
<tr>
<td>4.1.34</td>
<td>EID_PHY_MEM_ADDR_SIZE</td>
<td>98</td>
</tr>
<tr>
<td>4.1.35</td>
<td>EID_PHY_MEM_INVALID_reserve_PM</td>
<td>98</td>
</tr>
<tr>
<td>4.1.36</td>
<td>EID_PHY_MEM_MULTI</td>
<td>98</td>
</tr>
<tr>
<td>4.1.37</td>
<td>EID_PHY_MEM_OVERLAP</td>
<td>99</td>
</tr>
<tr>
<td>4.1.38</td>
<td>EID_PHY_MEM_OVERLAPPED</td>
<td>99</td>
</tr>
<tr>
<td>4.1.39</td>
<td>EID_PHY_MEM_PRIVATE</td>
<td>99</td>
</tr>
<tr>
<td>4.1.40</td>
<td>EID_PHY_MEM_RESERVE_OVERLAP_RESERVE</td>
<td>100</td>
</tr>
<tr>
<td>4.1.41</td>
<td>EID_PHY_MEM_UNDEF</td>
<td>100</td>
</tr>
<tr>
<td>4.1.42</td>
<td>EID_PHY_MEM_UNDEF_ADDR</td>
<td>100</td>
</tr>
<tr>
<td>4.1.43</td>
<td>EID_PHY_NO_RESULT</td>
<td>100</td>
</tr>
<tr>
<td>4.1.44</td>
<td>EID_PHY_PROBLEM_OVERSIZE</td>
<td>102</td>
</tr>
<tr>
<td>4.1.45</td>
<td>EID_PHY_SIZE_OVERFLOW</td>
<td>102</td>
</tr>
<tr>
<td>4.1.46</td>
<td>EID_PL_AFTER_CYCLE</td>
<td>102</td>
</tr>
<tr>
<td>4.1.47</td>
<td>EID_PL_MULTI_MAPPING</td>
<td>103</td>
</tr>
<tr>
<td>4.1.48</td>
<td>EID_PL_MMATT</td>
<td>103</td>
</tr>
</tbody>
</table>
4.1.49  EID_PL_PORG.................................................................................................................................103
4.1.50  EID_PROGBIT_AFTER_NOBITS........................................................................................................104
4.1.51  EID_SCL_DIRECTIVE.......................................................................................................................104
4.1.52  EID_SEC_BAD_ATTR.........................................................................................................................105
4.1.53  EID_SEC_MEM_ATTR.........................................................................................................................105
4.1.54  EID_SEC_MEM_SIZE.........................................................................................................................105
4.1.55  EID_SEC_MULTI_DEF......................................................................................................................106
4.1.56  EID_SEC_NO_MEM............................................................................................................................106
4.1.57  EID_SEC_NOT_PLACED....................................................................................................................107
4.1.58  EID_SEC_OSEC_ATTR.....................................................................................................................107
4.1.59  EID_SEC_PC_BACK.........................................................................................................................107
4.1.60  EID_SEC_SIZE_OVERFLOW..............................................................................................................108
4.1.61  EID_SEC_UNDEF_MEM...................................................................................................................108
4.1.62  EID_SEC_UNMATCH_ATTR...............................................................................................................108
4.1.63  EID_SMALL_ATTMMU_SIZE............................................................................................................109
4.1.64  EID_SOME_CORES_WITHOUT_TASKS.............................................................................................109
4.1.65  EID_START_ADDR_EXPR..................................................................................................................110
4.1.66  EID_START_ADDR_MULTI................................................................................................................110
4.1.67  EID_START_ADDR_REDEF...............................................................................................................110
4.1.68  EID_TASK_OVERFLOW....................................................................................................................110
4.1.69  EID_TASK_REDEF_VM.....................................................................................................................111
4.1.70  EID_TASK_UNDEF..........................................................................................................................111
4.1.71  EID_UNRESOLVE_REF......................................................................................................................111
4.1.72  EID_WRONG_AT_ORG.......................................................................................................................112
4.1.73  EID_WRONG_VM_ORG.......................................................................................................................112

4.2  Parser error messages ..........................................................................................................................112
4.2.1  EID_BAD_CORENAME.........................................................................................................................112
4.2.2  EID_COMMENT................................................................................................................................113
4.2.3  EID_DEFINE...................................................................................................................................113

CodeWarrior Development Studio for StarCore 3900FP DSP Architectures SC3000 Linker Reference Manual, Rev. 10.9.0, 11/2015
Freescale Semiconductor, Inc.
<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2.4</td>
<td>EID_EMPTY_EXP</td>
<td>113</td>
</tr>
<tr>
<td>4.2.5</td>
<td>EID_INCLUDE</td>
<td>114</td>
</tr>
<tr>
<td>4.2.6</td>
<td>EID_INCOMPLETE_EXP</td>
<td>114</td>
</tr>
<tr>
<td>4.2.7</td>
<td>EID_MISSING_PAREN_EXP</td>
<td>114</td>
</tr>
<tr>
<td>4.2.8</td>
<td>EID_MATT_WX</td>
<td>115</td>
</tr>
<tr>
<td>4.2.9</td>
<td>EID_OLD_LCF_FORMAT</td>
<td>115</td>
</tr>
<tr>
<td>4.2.10</td>
<td>EID_PREPROCESS</td>
<td>115</td>
</tr>
<tr>
<td>4.2.11</td>
<td>EID_UNDEF_OPER_IN_EXP</td>
<td>116</td>
</tr>
<tr>
<td>4.2.12</td>
<td>EID_UNEXPECTED_TOKEN</td>
<td>116</td>
</tr>
<tr>
<td>4.2.13</td>
<td>EID_UNKNOWN_INTRINSIC</td>
<td>116</td>
</tr>
<tr>
<td>4.2.14</td>
<td>EID_UNKNOWN_PERM_FLAG</td>
<td>116</td>
</tr>
<tr>
<td>4.2.15</td>
<td>EID_UNSUPPORTED_ATTR</td>
<td>117</td>
</tr>
<tr>
<td>4.3</td>
<td>Setup error messages</td>
<td></td>
</tr>
<tr>
<td>4.3.1</td>
<td>EID_ARCH_NOT_SPECIFIED</td>
<td>117</td>
</tr>
<tr>
<td>4.3.2</td>
<td>EID_EXPR_CANNOT_EVAL</td>
<td>117</td>
</tr>
<tr>
<td>4.3.3</td>
<td>EID_LCF_INCOMPLETE</td>
<td>118</td>
</tr>
<tr>
<td>4.3.4</td>
<td>EID_LCF_INCORRECT</td>
<td>118</td>
</tr>
<tr>
<td>4.3.5</td>
<td>EID_NUM_CORES_GT_ARCH</td>
<td>118</td>
</tr>
<tr>
<td>4.3.6</td>
<td>EID_NUM_CORES_LT_ONE</td>
<td>119</td>
</tr>
<tr>
<td>4.3.7</td>
<td>EID_NUM_CORES_NAN</td>
<td>119</td>
</tr>
<tr>
<td>4.3.8</td>
<td>EID_REDEF_LCF_SYM</td>
<td>119</td>
</tr>
<tr>
<td>4.3.9</td>
<td>EID_REDEF_MM_SYM</td>
<td>120</td>
</tr>
<tr>
<td>4.3.10</td>
<td>EID_REPEATED_SECTION_DIFF_OS</td>
<td>120</td>
</tr>
<tr>
<td>4.3.11</td>
<td>EID_REPEATED_SECTIONSAME_OS</td>
<td>120</td>
</tr>
<tr>
<td>4.3.12</td>
<td>EID_TASK_REDEF_VM</td>
<td>121</td>
</tr>
<tr>
<td>4.3.13</td>
<td>EID_TASKS_NOT_SPECIFIED</td>
<td>121</td>
</tr>
<tr>
<td>4.3.14</td>
<td>EID_UNSUPPORTED_ARCH</td>
<td>121</td>
</tr>
</tbody>
</table>
## Chapter 5
### LCF Expression Functions

5.1 Context-dependent intrinsic functions
   - 5.1.1 .............................................................................................................................................................................. 124
   - 5.1.2 align...................................................................................................................................................................... 124
   - 5.1.3 endof.................................................................................................................................................................... 124
   - 5.1.4 originof................................................................................................................................................................ 124
   - 5.1.5 core_id.............................................................................................................................................................. 125
   - 5.1.6 sizeof.................................................................................................................................................................... 125
   - 5.1.7 task_id.............................................................................................................................................................. 125
   - 5.1.8 to_physical........................................................................................................................................................ 125
   - 5.1.9 vmorg............................................................................................................................................................... 126

5.2 Context-independent intrinsic functions
   - 5.2.1 num_task............................................................................................................................................................. 126
   - 5.2.2 physical_address.................................................................................................................................................. 126
   - 5.2.3 num_core.............................................................................................................................................................. 127
   - 5.2.4 defined............................................................................................................................................................... 127
   - 5.2.5 test_arch............................................................................................................................................................. 127

## Chapter 6
### LCF Expression Operators

6.1 LCF expression operators................................................................................................................................................ 129

## Chapter 7
### LCF Preprocessing

7.1 LCF preprocessing .......................................................................................................................................................... 131
   - 7.1.1 Comments........................................................................................................................................................... 131
   - 7.1.2 The include directive............................................................................................................................................ 132
   - 7.1.3 The define directive............................................................................................................................................. 132
   - 7.1.4 Conditional directives........................................................................................................................................... 132
<table>
<thead>
<tr>
<th>Section number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 8</td>
<td>Linker Predefinitions</td>
<td></td>
</tr>
<tr>
<td>8.1</td>
<td>Predefined Symbols for MMU Descriptors</td>
<td>135</td>
</tr>
<tr>
<td>8.2</td>
<td>Predefined Physical Memory Regions</td>
<td>137</td>
</tr>
<tr>
<td>Chapter 9</td>
<td>Command-Line Options</td>
<td></td>
</tr>
<tr>
<td>Chapter 10</td>
<td>Sections in LCF</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 1
Introduction

The StarCore linker is a part of StarCore development tools and generates an executable file for the StarCore family of digital signal processors. In addition, the linker also lets you define a Linker Command File (LCF) that you use to instruct the linker to store different parts of the executable file in different areas of the processor address space.

Currently, StarCore development tools support following linker version:

- SC3000

The SC3000 linker specifically targets SC3900 family of processors. This user guide explains SC3000 linker.

In this chapter:

- About this document
- Accompanying documentation

1.1 About this document

This chapter describes the structure of this manual.

The StarCore SC3000 Linker User Guide is written using a task-based authoring approach. The document consists of:

- tasks chapter that contains linker configuration and general tasks, where:
  - linker configuration tasks help you configure the linker according to your application requirements.
  - general linker tasks help you use the linker during the general development process.
- concepts chapter that contains linker configuration and general concepts, where:
Accompanying documentation

- linker configuration concepts present linker concepts that you might need to comprehend to accomplish linker configuration tasks.
- general linker concepts present linker concepts that you might need to comprehend to accomplish general linker tasks.
- appendices (reference information), such as error/warning messages, linker pre-defined symbols, LCF pre-processing directives, etc. to which you might need to refer to accomplish certain linker tasks.

In addition, each linker task and concept provides a cross reference to:

- related tasks
- related concepts
- related references

Table 1-1 lists each chapter and appendix in this user guide and provides a summary of each.

<table>
<thead>
<tr>
<th>Chapter/Appendix name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tasks</td>
<td>Consists of step-by-step instructions that help you configure and use the linker.</td>
</tr>
<tr>
<td>Concepts</td>
<td>Consists of linker concepts that you might need to comprehend to accomplish the linker tasks.</td>
</tr>
<tr>
<td>Linker Error Messages</td>
<td>Explains linker error/warning messages and how to resolve them.</td>
</tr>
<tr>
<td>LCF Expression Functions</td>
<td>Explains the expressions that you use in the LCF.</td>
</tr>
<tr>
<td>LCF Expression Operators</td>
<td>Explains the operators that you use in the LCF.</td>
</tr>
<tr>
<td>LCF Preprocessing</td>
<td>Explains the LCF pre-processing directives.</td>
</tr>
<tr>
<td>Linker Predefinitions</td>
<td>Explains the linker pre-defined symbols.</td>
</tr>
<tr>
<td>Command-Line Options</td>
<td>Lists the command-line options that sc3000-ld linker supports.</td>
</tr>
<tr>
<td>Sections in LCF</td>
<td>Explains and lists the sections generated by the CodeWarrior linker and compiler.</td>
</tr>
</tbody>
</table>

1.2 Accompanying documentation

The Documentation page describes the documentation included in this version of CodeWarrior Development Studio for StarCore 3900FP DSP Architectures. You can access Documentation page by:

- a shortcut link on the Desktop that the installer creates by default, or
- opening START_HERE.html in CWInstallDir\SC\Help folder.
Chapter 2
Tasks

This chapter consists of step-by-step instructions that help you configure and use the linker during the general development process.

In this chapter:

- Linker configuration tasks
- General linker tasks

2.1 Linker configuration tasks

This chapter describes the linker configuration tasks.

In this section:

- How to create a Linker Command File (LCF)
- How to Make LCF Compatible for Flexible Startup
- How to Define and use a Custom set of Tasks
- How to Setup Virtual Trace Buffer (VTB) Using LCF
- How to Setup Cache
- How to Define Physical Memory Address Space of Target Architecture
- How to define stack and heap memory area in LCF
- How to Define Physical Memory Layout for a Multi-core Application
- How to Modify the LCF When Each Core Runs Different Code
- How to Define the Shared Memory
- How to Create Virtual Memory for Private Sections
- How to Define Virtual Memory for Read-Write-Execute (RWX) Access
2.1.1 How to create a Linker Command File (LCF)

Follow these steps:

1. Create a new file using an editor, such as vi or Notepad, and specify the extension as .l3k.

   The .l3k extension indicates an LCF for StarCore SC3000 linker.

2. Specify the architecture. For example:

   arch(b4860); // for multi-core applications

3. Specify the number of cores:
   a. If developing a single core application, skip this step and see Step 4
   b. If developing a multi-core application and wish to assign all available cores to the application, skip this step and see Step 4
   c. If developing a multi-core application and wish to assign an explicit number of cores to the application, specify:

      number_of_cores(3); // 3 indicates explicit no. of cores

      number_of_cores(1); // 1 indicates explicit no. of cores

4. Set the SR register. This step is mandatory for both single core and multi-core applications. The listing below shows an example for a multi-core application (b4860).

   Listing: Setting the SR register for b4860 application

   // The value to set the Core Status Register (SR) after reset:
   // - Exception mode
   // - Saturation on
   // - Rounding mode: 2's complement rounding
   _SR_Setting = 0xc;

5. Specify the system entry point. For example:

   entry("___crt0_start");

   You can also define a linker symbol and set it as the system entry point. For example:

   _program_start = _VBAddr+0x100;
   entry("_program_start");

6. Optional. Specify a task for the core (or each core in case of a multi-core application).

   The linker by default creates a task for each core. To create a user-defined task, see How to Define and use a Custom set of Tasks.

7. Specify the memory layout:
a. If developing a single core application, skip this step and see Step 8
b. Optional for multi-core applications. See How to Define Physical Memory Layout for a Multi-core Application.

8. Specify the virtual memory. You must instruct the linker about where to place the data and code sections of the user application.
   a. If developing a single core application, see How to Create Virtual Memory for Private Sections.
   b. If developing a multi-core application, see How to Create Virtual Memory for Private Sections, and How to Define the Shared Memory.

9. Specify the address translation entries. You must instruct the linker about how to map the virtual memory area to the physical memory area.
   a. Optional for single core applications. See How to Map Virtual Memory Areas to Physical Memory Address Space.
   b. If developing a multi-core application, see How to Map Virtual Memory Areas to Physical Memory Address Space.

Example

You can access the example LCF files for the multi-core applications from the following directories:

CWInstallDir/SC/StarCore_Support/compiler_3900/etc/b4860/lcf

CWInstallDir/SC/StarCore_Support/compiler_3900/etc/b4420/lcf

For example:

- b4860/lcf/common.l3k: Provides definition common to all cores
- b4860/lcf/mmu_attr.l3k: Provides definition for the MMU and Cache configuration
- b4860/lcf/b4860.l3k: Provides definition for the b4860 core

Notes

- It is a best practice to define the information in separate files, and include the files in the main LCF by using the include directive. For example, if you develop multiple applications for the same architecture, you might define common specifications in one file, and include this file in the multiple LCFs by using the include directive:

  ```
  #include "common_specs.txt"
  ```

- If you want, you can first rename some input sections, and then refer them in the LCF with the new name. You can use the RENAME directive, where the last parameter of the directive represents the new name. You can use renaming to exclude specific sections from specific cores by using the core visibility notation. For example:

  ```
  RENAME "*startup_*.eln", ".text", ".text_boot"
  RENAME "*rtlib_*.elb(target_asm_start.eln)", ".text", ".text_boot"
  unit shared(task0_c2,task0_c3){
  RENAME "c0cl_partial_shared_code.eln", ".text", "c0`exclude"
  }``
2.1.2 How to Make LCF Compatible for Flexible Startup

To make an LCF compatible for the flexible startup configuration, follow the steps given below. However, note that when you continue developing a project, a few changes in LCF are required to have the memory map expose asymmetry with respect to the previously referred sections, such as stack, heap, .att_mmu etc.

In addition, note that with each step given below, you must remove definitions of some specific symbols, while for some other symbols, it is not necessary to remove their definition. You may keep or remove such definitions depending upon the context of the application.

1. Core independent (flexible) placement of the stack and the .att_mmu section.
   a. You may remove the definitions of following symbols because linker can compute them by using the following definitions:

      _StackStart= originof("stack");

      _TopOfStack = (endof("stack")-7)&0xFFFFFFF8;

      If these definitions are not present in the LCF and the stack name is different from "stack", linker cannot compute the stack boundaries.

   b. Symbol definitions that you must remove:

      _LocalData_b

      _LocalData_size
However, note that the stack and the .att_mmu sections still must be placed in the same output_section:

descriptor__xxx__cacheable_wb__sys__private__data__boot {
    LNK_SECTION(att_mmu, "rw", _MMU_TABLES_size, 0x4, ".att_mmu");
    LNK_SECTION(stack, "rw", _StackSize, 0x4, "stack");
} > data_boot_c;

2. Core independent (flexible) placement of heap.
   a. You may remove the definitions of following symbols because linker can compute them by using the following definitions:

      __BottomOfHeap = originof("heap");
      __TopOfHeap = (endof("heap")-7) & 0xFFFFFFF8;

      If these definitions are not present in the LCF and the heap name is different from "heap", linker cannot compute the heap boundaries.

3. Core independent (flexible) placement of exception table and static initialization table.
   a. You may remove the definitions of following symbols because linker can compute them by using the following definitions:

      _cpp_staticinit_start= originof(".staticinit");
      _cpp_staticinit_end= endof(".staticinit");
      __exception_table_start__ = (ENABLE_EXCEPTION) ?originof(".exception_index"):0;
      __exception_table_end__ = (ENABLE_EXCEPTION) ?endof(".exception_index"):0;

      If these symbols definitions still exist, linker computes the new corresponding symbols based on the provided value.

2.1.2.1 Constraints With Flexible Startup Configuration

- The .text_boot sections must be shared and mapped 1-1.
• The `.att_mmu` section and stack must be defined using the same descriptor.
• All exception and staticinit tables need to be consecutive and some of them may need privatization, which can be achieved by renaming to `.exception_index` because of RT binary search.

Related Concept

• Understanding Flexible Startup Configuration

2.1.3 How to Define and use a Custom set of Tasks

A task is a static software unit, which has a unique ID that is recognized by the operating system and the hardware. The linker by default creates one task for each core. However, one or more tasks can be mapped to a single core. The name of the default task is the core number prefixed by the task0_c string.

Follow these steps to define and use a custom set of tasks:

1. Create a new task by using the tasks construct in the LCF. The listing below shows an example.
   **Listing: Creating a new task**

   ```
tasks //core_name: task_name, task_id, pid, did;
   c0 : sys0, 0, 0, 0;
   c0 : sub_task, 2, 2, 2;
   ...
   }
   
   2. Use the application configuration file, or the `__attribute__((section()))` qualifier to place the task specific data and task specific code sections in an input section. In the current StarCore programming model, new tasks created with the tasks construct are not explicitly extended at the C/C++ language level.

   The listing below shows an example of how you specify the task specific data and the task specific code sections in an application configuration file.

   **Listing: Specifying task specific data and code in an application configuration file**

   ```
   program = [
       subtask_program: ".subtask_pgm" //input code section for
       //sub_task
   ]
   ```
data = [
    subtask_data: "\subtask_data" //input data section for
    //sub_task
]

You can also use the \_attribute\_((section())) qualifier, instead of an application
file. For example:

\_attribute\_((section("\subtask_data"))) int data;
\_attribute\_((section("\subtask_pgm"))) void func() {}

3. Specify the virtual memory area and the address translation entries for the task
specific input sections in the LCF. The listing below shows an example.
**Listing: Specifying the virtual memory area and address translation entries**

unit private (sub_task) {
    MEMORY {
        m3__data_nc_wt_sub ("rw"): org = _sub_VIRTUAL_start;
        m3__text_c_sub ("rx"): AFTER(m3__data_nc_wt_sub);
    }
    SECTIONS {
        out_sub_data{
            .subtask_data
        } > m3__data_nc_wt_sub;
        out_sub_text{
            .subtask_pgm
        } > m3__text_c_sub;
    }
    address_translation (sub_task) {
        m3__data_nc_wt_sub(USER_DATA_MMU_DEF_REGA, USER_DATA_MMU_DEF_REGC):M3;
        m3__text_c_sub(USER_PROG_MMU_DEF_REGA, USER_PROG_MMU_DEF_REGC):M3;
    }
}

4. Specify the same start address for the tasks specific code, if the application runs two
or more tasks on the same core. The listing below shows an example, where the code
for sub_task and the sub_task_two tasks start at the same virtual memory address.
**Listing: Specifying same start address for multiple tasks on same core**

unit private (sub_task) {
    MEMORY {
        m3__data_nc_wt_sub ("rw"): org = _sub_VIRTUAL_start;
        m3__text_c_sub ("rx"): AFTER(m3__data_nc_wt_sub);
    }
}
In such cases, the system level task must perform the MMU management so that the associated descriptors are enabled and disabled in accordance with the active task, and also the current program/data ID registers are updated with the task PID/DID.

Notes

• It is a best practice to define the task specific data and task specific code in a separate C/C++ language module. This makes the specifications in the application file easy. For example:

```plaintext
module "b4860_subtask" [  
data = subtask_data  
program = subtask_program  
]
```

Related Concepts

• Understanding linker terminology

2.1.4 How to Setup Virtual Trace Buffer (VTB) Using LCF

Follow these steps:

1. Enable the VTB by using the `_ENABLE_VTB` symbol. The listing below shows an example.

   **Listing: Defining `_ENABLE_VTB` symbol**

   ```plaintext
   // Definitions for the VTB  
   // if 1 = reserve VTB in M2 memory  
   // if 2 = reserve VTB in M3 memory  
   // else VTB will not be configured automatically
   
   _ENABLE_VTB = 1;
   ```
2. Specify the start and end addresses (physical addresses) for the VTB by using the `_VTB_start` and `_VTB_end` symbols. The listing below shows an example.

### Listing: Defining _VTB_start and _VTB_end symbols

```c
// Reserve TRACE_BUFFER in physical memory
// Set TRACE_BUFFER start address and size for DDR
_TRACE_BUFFER_size = (_ENABLE_TB == 1)? 0x20000: //128K for each core
0x0;
_TRACE_BUFFER_start= _ENABLE_TB == 1)? _DDR_PRIVATE_end - _TRACE_BUFFER_size + 1:
0x0;
_TRACE_BUFFER_end = _TRACE_BUFFER_start + _TRACE_BUFFER_size;
```

3. Reserve a physical address space that the VTB can use

See [How to Reserve Physical Memory Area](#)

### Notes

- The VTB does not require virtual address space. Therefore, no address translation entries are required for VTB in the LCF.

### Related Tasks

- [How to Reserve Physical Memory Area](#)

### Related Concepts

- [Understanding linker terminology](#)

### Related References

- [Linker Predefinitions](#)

## 2.1.5 How to Setup Cache

Follow these steps:

1. Enable the cache by using the `ENABLE_CACHE` symbol. For example:

   ```c
   _ENABLE_CACHE = 1;
   ```

   Specifying `-1` as the value disables the cache.
2. Partition the M3/L3 memory by specifying an appropriate size for the M3 memory. You use the _M3_Setting symbol to specify the M2 memory size. Table 2-1 lists the supported values that you can specify for the _M3_Setting symbol.

Table 2-1. Supported Values for _M3_Setting Symbol

<table>
<thead>
<tr>
<th>_M3_Setting Value</th>
<th>M3 Memory Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00 - all memory used as L3Cache</td>
<td>0KB</td>
</tr>
<tr>
<td>0x0f</td>
<td>512KB</td>
</tr>
<tr>
<td>0xff</td>
<td>1024KB</td>
</tr>
</tbody>
</table>

The default value sets the L3/M3 as L3 cache.

3. Specify the cacheable attributes by using the MMU descriptors.

When you map the virtual memory area to the physical memory area, you specify the MMU descriptors. A few descriptors refer to caching. The listing below shows an example.

Listing: Using MMU Descriptors for Caching

```c
SYSTEM_DATA_MMU_DEF_REGA = MMU_DATA_CACHEABLE | MMU_DATA_PREFETCH_ANY | MMU_DATA_DEF_WPERM | MMU_DATA_DEF_RPERM;
SYSTEM_DATA_MMU_DEF_REGC = MMU_DATA_COHERENCY_MODE;
address_translation (*) {
    data_boot_c (SYSTEM_DATA_MMU_DEF_REGA, SYSTEM_DATA_MMU_DEF_REGC): DDR, org = _PRIVATE_DATA_BOOT_start;
}
```

Related Tasks

- How to define stack and heap memory area in LCF

2.1.6 How to Define Physical Memory Address Space of Target Architecture

The physical memory address space of the target architecture is automatically defined when you specify the architecture in the LCF. However, by using the physical_memory construct you can explicitly define:

- boundaries for each physical memory on the target architecture
- attributes for each physical memory on the target architecture
Follow one of these steps to explicitly define the physical memory address space:

- Use the `physical_memory` construct to define a physical memory address space that overlaps with the default physical memory address space
- Use the `physical_memory` construct to define a physical memory address space that has the same name as the default physical memory address space

The listing below shows default physical memory definitions for the b4860 architecture.

**Listing: Default physical memory definitions for b4860 architecture**

```c
physical_memory shared (*) {
    M3: org = _M3_start, len = _M3_size;
    DDR: org = _DDR_start, len = _DDR_size;
}
```

**Notes**

- If the default physical memory address space is `private`, then the explicitly defined physical memory address space must also be `private`
- If the default physical memory address space is `shared`, then the explicitly defined physical memory address space must also be `shared`. In addition, the core list must also be identical.

**Related Tasks**

- How to Define Physical Memory Layout for a Multi-core Application
- How to Reserve Physical Memory Area
- How to Map Virtual Memory Areas to Physical Memory Address Space

**Related Concepts**

- Understanding SC3000 LCF syntax

**Related References**

- Linker Predefinitions

### 2.1.7 How to define stack and heap memory area in LCF

Follow these steps:

1. Define the stack and heap memory size. For example, you might use user-defined symbols:
2. Declare the stack and heap memory area by using the `LNK_SECTION` construct. The listing below shows an example.

**Listing: Declaring stack and heap memory area**

```c
LNK_SECTION (stack, //section type
    "rw", //flags
    app_StackSize, //length
    0x8, //alignment
    "app_stack"); //name

LNK_SECTION (heap, "rw", app_HeapSize, 0x8, "app_heap");
```

The stack and heap memory alignment value for the StarCore architectures is eight bytes (0x8).

The `LNK_SECTION` construct defines an input section of one of the following predefined types: stack, heap, stack_and_heap, .att_mmu or .bss of the given size and alignment.

3. Define the following symbols for CodeWarrior runtime support routines:

```c
__StackStart = originof ("app_stack");
__TopOfStack = (endof ("app_stack") - 7) & 0xFFFFFFF8;
__BottomOfHeap = originof ("app_heap");
__TopOfHeap = (endof ("app_heap") - 7) & 0xFFFFFFF8;
```

The `originof` and `endof` functions are intrinsic functions that return the address of the specified input section. The input section can be a regular input section (identified by its name), or a special input section (defined and identified by using the `LNK_SECTION` construct).

4. Define the common memory area, if required, for the stack and heap memory by using the predefined stack_and_heap section. The listing below shows an example.

**Listing: Defining the common memory area for stack and heap memory**

```c
LNK_SECTION (stack_and_heap, "rw", StackHeapSize, 0x8, 
app_stack_and_heap");

__StackStart = originof ("app_stack_and_heap ");
__TopOfStack = (endof ("app_stack_and_heap ") - 7) & 0xFFFFFFF8;
__BottomOfHeap = originof ("app_stack_and_heap ");
__TopOfHeap = (endof ("app_stack_and_heap ") - 7) & 0xFFFFFFF8;
```

### 2.1.7.1 Example for Multi-Core Architectures
For multi-core architectures that have MMU support, for example b4860, the stack must be placed in the same construct as the .att_mmu section. The size of this construct (generally, the size of stack + the size reserved for the .att_mmu section) must be a power of two, and must be aligned to the size.

The listing below shows an example of `private__data__boot` construct (output section) of a multi-core application.

**Listing: Example output section of a multi-core application**

```c
private__data__boot {
    LNK_SECTION(att_mmu, "rw", _MMU_TABLES_size, 0x4, ".att_mmu");
    LNK_SECTION(stack, "rw", _StackSize, 0x4,"stack");
} > data_boot_vmemory;
```

The listing below shows the definition of the `data_boot_vmemory` memory area.

**Listing: Definition of `data_boot_vmemory` memory area**

```c
MEMORY {
    data_boot_vmemory ("rw"): org = _VIRTUAL_DATA_BOOT_start, len = _DATA_BOOT_size;
    ...
}
_DATA_BOOT_size = _StackSize + _MMU_TABLES_size;
```

In the listing above, `_VIRTUAL_DATA_BOOT_start` is a multiple of `_DATA_BOOT_size`, and `_DATA_BOOT_size` is a power of two.

You must define the following symbols because of specific constraints in the runtime library:

- **_LocalData_b**
  
  Specifies the virtual memory address of the construct, where the .att_mmu section and the stack are placed

- **_LocalData_size**
  
  Specifies the memory size of the construct, where the .att_mmu section and the stack are placed

- **_LocalData_Phys_b**
  
  Specifies the physical memory address of the construct on the first core, where the .att_mmu section and the stack are placed

For the construct defined in the listing, Example output section of a multi-core application, the values of _LocalData_b, _LocalData_size, and _LocalData_Phys_b symbols are:
LocalData_b = _VIRTUAL_DATA_BOOT_start;
LocalData_size = _DATA_BOOT_size;
LocalData_Phys_b = _PRIVATE_DATA_BOOT_start - (core_id() \* 0x01000000); // first descriptor is placed in the M2 memory

The _PRIVATE_DATA_BOOT_start address is an address in the M2 memory, and is also aligned to _DATA_BOOT_size.

Related Tasks

• How to Setup Cache

Related References

• Linker Predefinitions

2.1.8 How to Define Physical Memory Layout for a Multi-core Application

As a prerequisite, it is recommended that you see Appendix E on Linker Predefinitions before continuing with this task.

In case of single core applications, defining a memory layout is not required as the linker predefined physical memory regions are sufficient in most of the cases.

However, in case of multi-core applications, the physical memory address space is shared by all the cores on the platform. As a result, you might need to classify the shared physical memory address space into the private and shared memory regions. You specify the private and shared memory regions by creating a set of symbols for the start address, the end address, and the size of these regions.

Follow these steps to specify a private memory region in the shared physical memory space:

1. Define the size of the private memory region in the shared physical memory space. For example:

   // Define the size for private data to stay in M3
   _PRIVATE_M3_DATA_size = 0x1000;

   The _PRIVATE_M3_DATA_size symbol defines the size of private memory region for each core on the platform. If the application uses \( n \) cores, the size of M3 private memory region will be \( n \times _PRIVATE_M3_DATA_size \).
2. Define the start address of the private memory region. The listing below shows an example.

**Listing: Defining the start address of private memory region**

```c
// Define the physical memory for M3 private data
// The private space is placed at the beginning of M3 if the size of
// private space is bigger than size of shared space; otherwise at the
// end of M3.
_PRIVATE_M3_DATA_start = (_PRIVATE_M3_DATA_size <
   _M3_size-(core_num()*_PRIVATE_M3_DATA_size)) ?
   _M3_start+_PRIVATE_M3_DATA_size -
   (core_id()*_PRIVATE_M3_DATA_size):
   _M3_start + (core_id() * _PRIVATE_M3_DATA_size);
```

The listing above specifies two linker intrinsic functions:

- num_core() - returns the number of cores on the platform. You can specify a smaller number of cores per the application requirements.
- core_id() - returns the core ID. Core ID begin from 0.

Following Step 1 and Step 2, you have specified a private M3 memory region that you can use to place the private data for each core. The core_id() intrinsic function specifies the location of private memory region for each core preventing overlapping memory regions.

3. Define the end address of the private memory region. For example:

```c
:'/Define the end of M3 private data.
_PRIVATE_M3_DATA_end = _PRIVATE_M3_DATA_start + _PRIVATE_M3_DATA_size - 1;
```

**Example**

The listing below shows an example of how the private memory region created in the M3 memory is used in the virtual memory specification and address translation constructs.

**Listing: Using private memory region**

```c
/*
// This is linker predefined from the architecture specification
_PRIVATE_M3_DATA_start = 0x30000000;
*/

// Define the size for private data to stay in M3
_PRIVATE_M3_DATA_size = 0x10000;

// Define the physical memory for M3 private data
// The private space is placed at the beginning of M3
// if the size of private space is bigger than size of shared space.
```

Freescale Semiconductor, Inc.
M3_PRIVATE_start = (_PRIVATE_M3_DATA_size <
    _M3_size-(core_num() * _PRIVATE_M3_DATA_size)) ?
    _M3_start+_M3_size-(core_num() * _PRIVATE_M3_DATA_size)
    + (core_id() * _PRIVATE_M3_DATA_size):
    _M3_start + (core_id() * _PRIVATE_M3_DATA_size);

M3_PRIVATE_end = M3_PRIVATE_start + _PRIVATE_M3_DATA_size -1;

// Define the physical memory for M3 shared data and code
// The shared space is placed at the beginning of M3
// if the size of shared space is bigger than size of private space.
M3_SHARED_start = (_PRIVATE_M3_DATA_size < _M3_size - (core_num() * 
    _PRIVATE_M3_DATA_size))?
    _M3_start:
    _M3_start + (core_num() * _PRIVATE_M3_DATA_size);
// Defines for virtual memory map placement
VIRTUAL_PRIVATE_MEM_DATA_start = 0x70000000;
/* creating the private virtual memories*/
unit private (*){
    MEMORY {
    ddr_private_data_c_wb ("rw"): org = _VIRTUAL_PRIVATE_MEM_DATA_start;
    m3_private_data_c_wb ("rw"): AFTER(m2_private_data_c_wb);
    }
SECTIONS {
    descriptor_m3__cacheable_wb__sys__private_data {
    m3__cacheable_wb__sys__private__data
    reserved_crt_tls
    .data
    .m3__cacheable_wb__sys__private__rom
    .bsstab
    .init_table
    .rom_init
    .rom_init_tables
    .exception
    .exception_index
    .staticinit

CodeWarrior Development Studio for StarCore 3900FP DSP Architectures SC3000 Linker Reference Manual, Rev. 10.9.0, 11/2015
Freescale Semiconductor, Inc.
\m3\_cacheable\_wb\_sys\_private\_bss
\bss
}\ > m3\_private\_data\_c\_wb;
\}
\}
address\_translation\ (\*)\ {
\ ddr\_private\_data\_c\_wb (SYSTEM\_DATA\_MMU\_DEF\_REGA, SYSTEM\_DATA\_MMU\_DEF\_REGC): M2, org = \_DDR\_PRIVATE\_start;
\ m3\_private\_data\_c\_wb (SYSTEM\_DATA\_MMU\_DEF\_REGA, SYSTEM\_DATA\_MMU\_DEF\_REGC): M3, org = \_M3\_PRIVATE\_start;
}\}

Related Tasks

- How to Define Physical Memory Address Space of Target Architecture
- How to Reserve Physical Memory Area

Related References

- Linker Predefinitions

2.1.9 How to Modify the LCF When Each Core Runs Different Code

Follow these steps:

1. Consider an LCF for a multi-core application where all the cores share code. Further, consider a shared\_func function as a start point where all the cores begin to run different code. The listing below shows an example where the shared\_func function is placed in the .text section.

Listing: Example unit construct

```
unit shared (\*) {  
MEMORY {
  m3\_shared\_text\_c ("rx"): org = \_M3\_SHARED\_start;
}
SECTIONS {
  descriptor\_m3\_cacheable\_sys\_shared\_text { 
    .m3\_cacheable\_sys\_shared\_text
```
2. The `shared_func` functions calls a private function, `private_entry_point` that has different code on each core but it has to be placed at the same virtual address on all cores. You specify the `private_entry_point` function in a C source file as:

```c
extern void private_entry_point (void);
void shared_func(void){
    private_entry_point();
}
```

3. Create different C source code files for different private code that runs on each core. For example, create `c0_private_text.c` and `c1_private_text.c` files for cores `c0` and `c1` respectively. In each of such C source code files, specify an entry function, `private_entry_point();`.

When specifying the entry function, you must follow these steps:

a. Define core specific code sections in the application configuration file so that the resolution of `_private_entry_point` symbol is correct. For example:

```plaintext
Pgm0_shared_to_private :  ".text" core="c0 "/* private section c0`.text for core 0 */ ,
Pgm1_shared_to_private :  ".text" core="c1 "/* private section c1`.text for core 1 */
```

b. Set the code sections at the function level. The listing below shows an example.

```
Listing: Setting code sections at function level

module "c0_private_text" [ 

    function _private_entry_point [ 
        program = Pgm0_shared_to_private
    ]
]

module "c1_private_text" [ 

    ... 
    function _private_entry_point [ 
        program = Pgm1_shared_to_private
    ]
]
```
c. Place the functions at the same address in tasks' virtual memory space on all the cores by using the LCF. This is required because the functions are private, but called from shared code. The listing below shows an example.

**Listing: Placing functions at the same address in virtual memory space**

```c
unit private (task0_c0) {
  MEMORY {
    ddr_shared_to_private_text_c0 ("rx"): org = _VIRTUAL_DDR_PRIVATE_text_start;
  }
  SECTIONS{
    outsec_ {
      "c0`.text" // input section with function _private_entry_point for core 0
    } > ddr_shared_to_private_text_c0;
  }
}

unit private (task0_c1) {
  MEMORY {
    ddr_shared_to_private_text_c1 ("rx"): org = _VIRTUAL_DDR_PRIVATE_text_start;
  }
  SECTIONS{
    outsec_ {
      "c1`.text" // input section with function _private_entry_point for core 1
    } > ddr_shared_to_private_text_c1;
  }
}
```

4. Place all the functions that are called from the **private_entry_point** function in separate input sections, so that the functions are further easily placed in the private memory areas defined in the LCF. Follow these steps:

a. Define the code input sections for private code in the application configuration file. For example:

```c
program = [...
  Pgm0:"private_code_c0" /* private code for c0 */ ,
  Pgm1:"private_code_c1" /* private code for c1 */
]
```

b. Specify the defined input sections at the module level in the application file. For example:

```c
module "c0_private_text" [...
  program = Pgm0
```
c. Place the defined input sections separately in the task virtual memory space. The listing below shows an example.

**Listing: Placing the input section in task virtual memory space**

```plaintext
unit private (task0_c0) {
  MEMORY {
    ddr_private_text_c0 ("rx"): AFTER(ddr_shared_to_private_text_c0);
  }
  SECTIONS{
    outsec__{
      " private_code_c0"
    } > ddr_private_text_c0;
  }
}
unit private (task0_c1) {
  MEMORY {
    ddr_private_text_c1 ("rx"): AFTER(ddr_shared_to_private_text_c1);
  }
  SECTIONS{
    outsec__{
      " private_code_c1"
    } > ddr_private_text_c1;
  }
}
```

5. Specify the virtual to physical memory mapping using the `address_translation` construct. The listing below shows an example.

**Listing: Specifying the virtual to physical memory mapping**

```plaintext
address_translation (task0_c0) {
  ddr_shared_to_private_text_c0 (SYSTEM_PROG_MMU_DEF_REGA, SYSTEM_PROG_MMU_DEF_REGC): DDR, org = _DDR_PRIVATE_start;
  ddr_private_text_c0 (SYSTEM_PROG_MMU_DEF_REGA, SYSTEM_PROG_MMU_DEF_REGC): DDR;
}
address_translation (task0_c1) {
  ddr_shared_to_private_text_c1 (SYSTEM_PROG_MMU_DEF_REGA, SYSTEM_PROG_MMU_DEF_REGC): DDR, org = _DDR_PRIVATE_start;
```
2.1.10 How to Define the Shared Memory

Follow these steps:

1. Define the `unit` construct for specifying the memory areas and output sections. The listing below shows an example.

**Listing: Defining the unit statement for shared memory area**

```c
unit shared (task1, task2) {
    memory {
        // typical virtual memory definitions
        virtual_shared_data_memory ("rw") : org = _VirtualData_b;
        virtual_shared_const_memory ("r") : AFTER (virtual_shared_data_memory), len=0x200;
        virtual_shared_code_memory ("rx") : org = 0xC0000000, len=0x1000;
        virtual_shared_libs_memory ("rx") : AFTER (virtual_shared_code_memory);
        // minimal memory specification
        virtual_shared_memory_1;
        // other examples
        virtual_shared_memory_2 : org = _SomeSharedVirtualAddress;
        virtual_shared_memory_3 : len = 0x100;
    }
}
```

In the listing below:

- `task1` and `task2` indicate the tasks that share the memory; the wildcard (*) can be used to specify all tasks on the platform
- `virtual_shared_data_memory` specifies the shared virtual memory
- the `rw` keyword sets the Read and Write flags for the memory
• the `org` keyword sets the memory start address in the virtual memory space; the start address specified must adhere to architecture constraints
• the `len` keyword specifies the maximum memory size in bytes; if not specified, the linker uses the minimum size
• the `after` keyword instructs the linker to start placing the memory after the end address of the specified virtual memory area

NOTE

The `after` keyword does not mean right after. The memory can be placed starting at an address that is beyond the end address of the specified virtual memory.

2. Place the data/code in the virtual memory areas by using the output sections. The listing below shows an example.

Listing: Placing the data/code in the virtual memory area

```c
unit shared (task1, task2) {
    MEMORY {
        // typical virtual memory definitions
        virtual_shared_data_memory ("rw") : org = _VirtualData_b;
        virtual_shared_const_memory ("r") : AFTER (virtual_shared_data_memory), len=0x200;
        virtual_shared_code_memory ("rx") : org = 0xC0000000, len=0x1000;
        virtual_shared_libs_memory ("rx") : AFTER (virtual_shared_code_memory);
        // minimal memory specification
        virtual_shared_memory_1;
        // other examples
        virtual_shared_memory_2 : org = _SomeSharedVirtualAddress;
        virtual_shared_memory_3 : len = 0x100;
        virtual_shared_memory_4 DATA;
    }
    SECTIONS {
        // create an output section that uses a selection of data sections to be placed into
        // the virtual memory.
        output_section__shared__data {
            .m3__non_cacheable_wt__sys__shared__data
            reserved_crt_mutex
            .m3__non_cacheable_wt__sys__shared__rom
        }
    }
}
```
.m3__non_cacheable_wt__sys__shared__bss
   } > virtual_shared_data_memory;
   // some read only data to place into read only shared virtual
   // memory
   read_only_output_section {
      .rom
   } > virtual_shared_const_memory;
}

3. Specify the address translation entries to map the virtual memory to the physical memory areas. See How to Map Virtual Memory Areas to Physical Memory Address Space.

Notes

- Using the after keyword to instruct the linker that memory A starts after memory B, does not prohibit placing other memory regions in-between memory A and memory B.
- The access flags for the shared memory must be specified according to the type of data that the memory holds. A section that contains executable code cannot be placed into the virtual_shared_memory memory as declared in Listing: Defining the unit statement for shared memory area because this memory does not specify the x flag.
- Make sure that when you specify the tasks for the shared memory, the target memory space for placing the shared memory area is accessible to all the tasks.
- Reduce the usage of the AFTER keyword and provide precise placement using org and len keywords in order to reduce the problem search space and the linking time.

Related Tasks

- How to Map Virtual Memory Areas to Physical Memory Address Space
- How to define stack and heap memory area in LCF

Related Concepts

- Understanding SC3000 LCF syntax

2.1.11 How to Create Virtual Memory for Private Sections

Follow these steps:

1. Define the unit construct. The listing below shows an example.
   Listing: Defining the unit statement for virtual memory area
unit private (*) {  
/* 1.1 A MEMORY statement is needed to define the virtual memory areas.  
These virtual memory areas act like containers for the data/code  
section. Four memory areas are defined in four typical different  
ways:*/  
MEMORY {  
    local_data_descriptor_1 ("rw"): org = _VirtLocalDataDDR_b;  
    local_data_descriptor_2 ("rw"): AFTER(local_data_descriptor_1);  
    local_data_descriptor_3 ("rw"): org = _VirtLocalDataM3_b,  
        LEN=0x1000;  
    local_data_descriptor_4 ("rw"): AFTER(local_data_descriptor_3),  
        LEN=0x1000;  
}  
}  

In the above listing:  

- the unit scope is private, i.e. all data/code specified by this unit is private to the  
specified tasks list (the wildcard indicates all the tasks on the platform; you can  
specify a comma separated task list)  
- the MEMORY construct lets you create virtual memory areas and map them to the virtual  
memory space. Except the name, all other information required to create a virtual  
memory area is optional.  
- the rw keyword sets the Read and Write flags for the memory  
- the org keyword sets the memory start address in the virtual memory space; the start  
address specified must adhere to architecture constraints  
- the len keyword specifies the maximum memory size in bytes; if not specified, the  
linker uses the minimum size  
- the after keyword instructs the linker to start placing the memory after the end  
adress of the specified virtual memory area  
- local_data_descriptor_1 is a virtual memory that starts at _VirtLocalDataDDR_b. The  
access rights are set for Read and Write, so it is illegal to place the executable code in  
this memory area. The length is not specified. The linker determines the length from  
the sections placed into this memory area.  
- local_data_descriptor_2 is similar to local_data_descriptor_1, except the memory is  
placed to start after the local_data_descriptor_1, without fixed origin or length  
- local_data_descriptor_3 is similar to local_data_descriptor_1, except  
local_data_descriptor_3 has fixed length, in addition to fixed origin  
- local_data_descriptor_4 is similar to local_data_descriptor_2, except  
local_data_descriptor_4 has a fixed length  
- Specify the input sections by creating output sections and associate them with the  
specified virtual memory areas. The listing below shows an example.
Listing: Creating output sections

unit private (*) {
/* 1.1 A MEMORY statement is needed to define the virtual memory areas. These virtual memory areas act like containers for the data/code section. Four memory areas are defined in four typical different ways:*/

MEMORY {
    local_data_descriptor_1 ("rw"): org = _VirtLocalDataDDR_b;
    local_data_descriptor_2 ("rw"): AFTER(local_data_descriptor_1);
    local_data_descriptor_3 ("rw"): org = _VirtLocalDataM3_b,
        LEN=0x1000;
    local_data_descriptor_4 ("rw"): AFTER(local_data_descriptor_3),
        LEN=0x1000;
}
/* 2. A SECTIONS statement is needed to indicate a mapping between data/code sections and virtual memories. */

SECTIONS {
    /*2.1. The descriptor_local_data is an output section that gathers a list of input sections and assigns a virtual memory to hold them. In this example the descriptor_local_data uses the local_data_descriptor_1 virtual memory for placing the input sections.*/
    descriptor_local_data {
        .oskernel_local_data
        .data
        .ramsp_0
        .oskernel_rom
        .rom
        .exception_index
        .ramsp_0
        .init_table
        .rom_init
        .bsstab
        .rom_init_tables
        .staticinit
        LNK_SECTION(att_mmu, "rw", 0x200, 4, ".att_mmu");
        .oskernel_local_data_bss
    }
2.1.11 Linker configuration tasks

. bss

} > local_data_descriptor_1;

}

• Specify the address translation entries to map the virtual memory to the physical memory areas. See How to Map Virtual Memory Areas to Physical Memory Address Space.

Notes

• Using the after keyword to instruct the linker that memory A starts after memory B, does not prohibit placing other memory regions in-between memory A and memory B.

• The lesser the information you provide when creating a virtual memory space, the larger is the search space and the linking time.

Related Tasks

• How to Map Virtual Memory Areas to Physical Memory Address Space
• How to define stack and heap memory area in LCF

Related Concepts

• Understanding SC3000 LCF syntax

2.1.12 How to Define Virtual Memory for Read-Write-Execute (RWX) Access

You define virtual memory for RWX only on multi-core architectures that have explicit MMU support, and visible physical memory address space. Note that the linker does not check whether the memory space, that you want to assign RWX attributes, is cacheable. In addition, you must be aware of the hardware memory map to prevent tricky errors when selecting the target physical memory space for the RWX memory.

Follow these steps to define virtual memory for RWX access:

1. Define the virtual memory by using the unit construct. The listing below shows an example.

   Listing: Defining a virtual memory

   unit shared (*) {
      MEMORY {
         m3_rwx_memory : org = 0xC0004300;
      }
   }
Make sure that you do not specify any RWX flags for the virtual memory itself. The linker prevents you from accidently stamping a virtual memory as RWX without any code mapped to the respective memory.

2. Define an output section. The listing below shows an example.

**Listing: Defining an output section**

```c
unit shared (*) {
    MEMORY {
        m3_rwx_memory : org = 0xC0004300;
    }

    SECTIONS {
        m3__non_cacheable_wt__sys__shared__text ("rwx") {
            tracepoint_handler
            tracepoints_hash
        }> m3_rwx_memory;
    }
}
```

Make sure that you specify the RWX attributes for the output section. In addition, note that you can use only one output section to specify the RWX attributes for a virtual memory.

3. Specify the address translation entries for the virtual memory. The listing below shows an example.

**Listing: Specifying address translation entries**

```c
address_translation (*) map11{
    m3_rwx_memory (SHARED_NON_CACHEABLE_PROG_MMU_DEF_REGA,
    SHARED_NON_CACHEABLE_PROG_MMU_DEF_REGC) "rx":M3;

    m3_rwx_memory (SHARED_NON_CACHEABLE_DATA_MMU_DEF_REGA,
    SHARED_NON_CACHEABLE_DATA_MMU_DEF_REGC) "rw":M3;
}
```

Make sure that:

- you specify two address translation entries; one for data, and another for program
- you specify the access attributes for each entry; one must be RW, and another must be RX

It is the only case when you specify access attributes for address translation entries.
Related Tasks

- How to Map Virtual Memory Areas to Physical Memory Address Space
- How to Create Virtual Memory for Private Sections
- How to Define Private Data Sections for Multiple Cores

Related Concepts

- Understanding SC3000 LCF syntax

2.2 General linker tasks

This chapter describes the general linker tasks.

In this section:

- How to Reserve Physical Memory Area
- How to Define Private Data Sections for Multiple Cores
- How to troubleshoot linker error messages
- How to build expressions in the SC3000 LCF
- How to Map Virtual Memory Areas to Physical Memory Address Space
- How to specify the content of virtual memory areas
- How to Share Code and Data Partially Among Different Cores
- How to Limit Code and Data Visibility at Core Level
- How to Define Unlikely Block of Code as Private Block of Code in a Multi-core Application
- How to Run Multiple Tasks on the Same Core
- How to Make Code or Data Sections Visible to a Subset of Cores
- How to place a symbol in an another section in LCF
- How to Handle C++ Templates in Multi-core Applications
- How to Check Local Symbols Addresses
- How to use KEEP Directive
- How to reserve an MMU descriptor ID

2.2.1 How to Reserve Physical Memory Area

Follow these steps:

1. Identify following details of the memory area that you need to reserve.
• Beginning address
• Size
• Type; can be shared or private

2. Specify the identified details using the reserve, org, len, and private keywords. The listing below shows an example.

Listing: Reserving a physical memory area

```c
physical_memory private (*) {
  ...
  reserve: org = _DDR_start, len = _DDR_size;
  ...
}
```

Notes

• The linker does not occupy the reserved physical memory area
• A reserved physical memory must overlap with a defined physical memory. Since a reserved area does not have a name, the overlap must occur using the addresses that the reserved and physical memory areas occupy.

Related Tasks

• How to Define Physical Memory Address Space of Target Architecture
• How to Define Physical Memory Layout for a Multi-core Application

Related References

• Linker Predefinitions

2.2.2 How to Define Private Data Sections for Multiple Cores

This task covers the following Single Instruction Multiple Data (SIMD) scenario:

• Source code is shared among the cores
• Some data is private to the cores, and is initialized differently at the compile time

Follow these steps:

1. Place the private data, which is initialized differently at the compile time, in separate modules for each core. You must use same symbol in all modules. For example, to store the data for two tasks running on different cores, define a private_data_c0.c module and a private_data_c1.c module:

   ```c
   private_data_c0.c:
   int my_private_channel_info[] = {0,1,2,3,4};
   ```
2. Specify the defined modules in the application configuration file. This step binds the private data/modules to the associated cores. The listing below shows an example.

**Listing: Binding private data to associated cores**

```c
section /* private section c0`.data for core 0 */
data = [Data0 : ".data" core="c0", Data1 : ".data" core="c1"]
end section

/* place the data section from private_data_c0 into core's c0 private data */
module "private_data_c0" [ data = Data0 ]

/* place the data section from private_data_c1 into core's c1 private data */
module "private_data_c1" [ data = Data1 ]
```

3. Define a private unit in the LCF to create a private virtual memory area to store the private data. The listing below shows an example.

**Listing: Defining a private unit**

```c
unit private (task0_c0, task0_c1) {
    memory {
        private_init ("rw"): org = _VIRTUAL_PRIVATE_MEM_DATA_start;
    }
    sections{
        someoutsec {
            c*`.data
            } > private_init ;
    }
}
```

4. Map the private virtual memory area to a physical memory area by using the address_translation construct. For example:

```c
address_translation (*) {
    private_init (SYSTEM_DATA_MMU_DEF_REGA, SYSTEM_DATA_MMU_DEF_REGC) : DDR ,org = 
    DDR_PRIVATE_start;
}
```

**Example**

Consider the my_private_channel_info array (defined in step 1) placed at the same address in the M2 memory for cores c0 and c1.
The listing below shows an example function that manipulates the data from the `my_private_channel_info` array.

**Listing: Example function**

```c
int get_avg()
{
    int accumulator=0;
    for(int i=0; i< 5 ; i++)
        accumulator+= my_private_channel_info[i];
    return ( accumulator/5 );
}
```

For core c0, the `get_avg()` function returns 2, and for core c1, the function returns 12.

**Related Tasks**

- How to Define Physical Memory Address Space of Target Architecture

**2.2.3 How to troubleshoot linker error messages**

See appendix Linker Error Messages.

**2.2.4 How to build expressions in the SC3000 LCF**

An expression in the StarCore linker can consists of:

- symbols, including location counter
- user defined labels and associated integer values
- combination of integers and the linker expression functions

Use the `=` assignment operator to:

- create symbols
- assign memory addresses

If a symbol is defined in an input file in addition to the LCF, the definition in the LCF takes precedence. Use the `defined` linker expression function to cancel the precedence order, and to use the definition from the input file.

The listing below shows examples of various expressions in an LCF.
General linker tasks

Listing: Linker expressions

_DataStart = 0x1000;
_CodeStart = 0x10000;
_ROMStart = 0x7f000;
_StackStart = 0x28000;
_TopOfStack = 0x7eff0;
_SR_Setting = 0xe4000c;
arch(b4860);
UNIT private {*}{

MEMORY{
    mem1 ("rx"): org = 0, len = _DataStart;
    mem2 ("rw"): org = _DataStart, len = _CodeStart - _DataStart;
    mem3 ("rx"): org = _CodeStart, len = _ROMStart - _CodeStart;
    mem4 ("rw"): org = _ROMStart, len = 0x20000;
}

SECTIONS{
    INTVEC("rx"){
        .intvec;
    } > mem1 ;
    ...
    TEXT("rx"){ 
        .text;
        . += 100;
        .default;
    } > mem3;
    ...
    ROM("r"){ 
        .init_table;
        . = align(16);
        .rom_init;
        .rom;
    } > mem4;
}

_text_start = originof(TEXT);
Related Tasks

- How to create a Linker Command File (LCF)

Related Concepts

- Understanding linker terminology
- Understanding SC3000 LCF syntax

Related References

- LCF Expression Functions
- LCF Expression Operators

2.2.5 How to Map Virtual Memory Areas to Physical Memory Address Space

Once the virtual memory layout is defined, you must define the mapping of individual virtual memory areas to the physical memory address space by using the `address_translation` construct.

The listing below shows an example of the `address_translation` construct.

Listing: Example `address_translation` construct

```c
address_translation (sys0, sys1, sys2, sys3) map11
{
    m_shared_m3_x (SYSTEM_PROG_MMU_DEF_REGA, SYSTEM_PROG_MMU_DEF_REGC) : M3;
    m_shared_m3_d (SHARED_DATA_MMU_DEF_REGA, SHARED_DATA_MMU_DEF_REGC) : M3 descriptor(1);
    m_shared_ddr_d (SHARED_DATA_MMU_DEF_REGA, SHARED_DATA_MMU_DEF_REGC) map11 : DDR;
}
```

It is mandatory to specify the list of tasks that relate to the respective virtual memory areas. The `sys0`, `sys1`, `sys2`, `sys3` tasks in the listing above indicate the tasks.

When the virtual memory area is a shared, it will be mapped only once to the shared physical memory. Still, an entry in the address translation table gets created for each of the hosting cores. In the process of providing information for the MMU programming model, the MMU attributes for each descriptor must be captured using the `SYSTEM_PROG_MMU_DEF_REGA`, `SYSTEM_PROG_MMU_DEF_REGC` and `SHARED_DATA_MMU_DEF_REGA`, `SHARED_DATA_MMU_DEF_REGC` attributes.
SHARED_DATA.MMU_DEF_REGC symbols respectively. X_REGA and X_REGC represent the values used to program registers M_PSDAx / M_DSDAx and M_PSDCx / M_DSDCx respectively.

If the physical memory address is same as the virtual memory address, the map11 keyword can be specified either associated to each virtual memory area, or at the address_translation level for all the enclosed virtual memory areas. In such cases, if the org and the len keywords specified for the virtual memory area do not conform to the target architecture restrictions, the linking process stops with an error message.

In you want to map a virtual memory to a certain descriptor, the descriptpro(ID) parameter can be used.

The listing below shows an example of the address_translation construct that uses a wildcard.

Listing: Example address_translation construct using wildcard

```plaintext
address_translation (*)
{
  data_boot_c (SYSTEM_DATA.MMU_DEF_REGA, SYSTEM_DATA.MMU_DEF_REGC): DDR,
    org = _PRIVATE_DATA_BOOT_start, len = _DATA_BOOT_size;
  ddr_private_text_c (SYSTEM_PROG.MMU_DEF_REGA, SYSTEM_PROG.MMU_DEF_REGC): DDR;
  ddr_private_data_c_wb (SYSTEM_DATA.MMU_DEF_REGA, SYSTEM_DATA.MMU_DEF_REGC): after
    (ddr_private_text_c);
}
```

In the listing above, the wildcard * specifies the set of all tasks that are defined in the LCF for virtual-to-physical address translation. For the tasks that do not have associated virtual memory, the linker stops with an LCF configuration error message.

It is recommended that the origin of physical placement is defined by the org keyword as it reduces the search space and the linking time. You can further reduce the search space and the linking time by specifying the exact length of the mapped region with the len keyword.

Both the org and len keywords are optional when you specify the memory mapping, but provide the highest flexibility in memory specification.

The listing below shows an example of the address_translation construct that defines the content required to reserve an MMU entry.

Listing: Reserving an MMU entry

```plaintext
address_translation (*)
{
  reserve (SYSTEM_DATA.MMU_DEF_REGA, SYSTEM_DATA.MMU_DEF_REGC): paddr, _vaddr, size, "rwx";
```
The reservations for the address translation table entries must be fully specified with:

- physical and virtual start addresses
- size
- attributes

Generally, the access rights are \textit{rw} for data and \textit{rx} for code.

In the above listing, one address translation table entry is generated on the host core of each of the tasks in the list (specified with the wild card). The linker evaluates the reservation parameters (e.g. \texttt{paddr}, \texttt{vaddr}, \texttt{size}) in the task context.

As a result of the reservation entry specified in Listing:

- for each task, the virtual memory region \([vaddr, vaddr+size-1]\) is reserved
- the physical memory regions \([paddr, paddr+size-1]\) are reserved

For such a case, it is likely that the virtual address start symbol needs to be referenced at the C source code level for further content processing. For example:

```c
extern unsigned long int vaddr;
unsigned long int *ptr;
ptr = &vaddr;
```

Related Tasks

- How to Define Physical Memory Layout for a Multi-core Application
- How to Reserve Physical Memory Area
- How to Define Private Data Sections for Multiple Cores

Related References

- Linker Predefinitions

2.2.6 How to specify the content of virtual memory areas

Use output sections to specify the content of virtual memory areas. Each output section contains a list of input sections that are placed in the specified order in the respective virtual memory area. In addition to input sections, the output sections can also contain assignments and special input sections. The listing below shows an example.

**Listing: Example output section**

```c
unit shared (*) {
  MEMORY {
```
The my_output_section describes the content of the m3_shared_textboot_c virtual memory area. The output section contains two assignments and two input section specifications. The use of the location counter (".".) is permitted inside the output sections.

The input section names, such as .text_boot are user-defined at the compiler level. You can instruct the compiler to place some code or data in the specific user defined sections by using the:

- application configuration file; the application file gives flexibility without changing the C/C++ source files, or
- pragma options, such as pragma sgm_seg_name, data_seg_name, and bss_seg_name that redefine the default section names as they are used by the compiler, or
- __attribute__ qualifier for specific variable/function only. For example:

  __attribute__((section("mydata"))) int data;
  __attribute__((section("mytext"))) void func () {}

**Related Tasks**

- How to Define Private Data Sections for Multiple Cores

**Related References**

- Linker Predefinitions

**2.2.7 How to Share Code and Data Partially Among Different Cores**
Follow the steps described in this task when you need to partially share the code or the data among different cores. For example, let's assume that you need to share some data in the M3 memory between core 0 and core 3, and at the same time you need to share some other data in the M3 memory among cores 1, 2, 4 and core 5. In addition, let's assume that you are using the default tasks. Note that specifying a list of all tasks that run on two or more cores implies that the unit is shared by the respective two or more cores.

**NOTE**

In the StarCore linker, core numbering starts with 0.

**NOTE**

If an application has a shared code between core 0 and 3, then, for the application to build successfully, the shared sections must be excluded from private tasks and other shared tasks of cores 2, 1, 4, and 5.

A few more points that complete the example scenario cited above:

- The data that core 0 and core 3 share is placed in the M3 memory in `array_c0c3`
- Another segment of data that core 1, core 2, core 4 and core 5 share is placed in the M3 memory in `array_c1c2c4c5`
- A private function, `foo()`, that is partially shared by core 0 and core 3, and another private function, `foo()`, that is partially shared by core 1, core 2, core 4 and core 5
- A single `main` function shared by all cores

The steps to implement the example scenario are:

a. Split the data and code into separate files

This step is required because the compiler application configuration file and the LCF deal with the linker sections differently. Table 2-2 lists more details.

**Table 2-2. List of Files That Store Partially Shared Data and Code**

<table>
<thead>
<tr>
<th>File Name</th>
<th>Code Listing</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>shared_c0c3.c</td>
<td>Listing: Contents of shared_c0c3.c</td>
<td>Definition of <code>array_c0c3</code></td>
</tr>
<tr>
<td>shared_c1c2c4c5.c</td>
<td>Listing: Contents of shared_c1c2c4c5.c</td>
<td>Definition of <code>array_c1c2c4c5</code></td>
</tr>
<tr>
<td>shared_text_c0c3.c</td>
<td>Listing: Contents of shared_text_c0c3.c</td>
<td>Implementation of <code>foo</code> function for core 0 and core 3</td>
</tr>
<tr>
<td>shared_text_c1c2c4c5.c</td>
<td>Listing: Contents of shared_text_c1c2c4c5.c</td>
<td>Implementation of <code>foo</code> function for core 1, core 2, core 4 and core 5</td>
</tr>
</tbody>
</table>
Listing: Contents of shared_c0c3.c

//
int array_c0c3[] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
//

Listing: Contents of shared_c1c2c4c5.c

//
int array_c1c2c4c5[] = {1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000};
//

Listing: Contents of shared_text_c0c3.c

extern int array_c0c3[];
int foo()
{
    if (array_c0c3[0] || array_c0c3[1])
        return (array_c0c3[0] + array_c0c3[1]);
    return 0;
}

Listing: Contents of shared_text_c1c2c4c5.c

extern int array_c1c2c4c5[];
int foo()
{
    if (array_c1c2c4c5[0] || array_c1c2c4c5[1])
        return (array_c1c2c4c5[0] + array_c1c2c4c5[1]);
    return 0;
}

b. Modify the application configuration file

Each file listed in Table 2-2 must be described in the application configuration file in order to define the section name mappings as the listing below shows.

Listing: Modified application configuration file

program = [
    M3_shared_prog_c0c3 : ".m3_shared_text_c0c3",
    M3_shared_prog_c1c2c4c5 : ".m3_shared_text_c1c2c4c5",
    DDR_cacheable_sys_private_text_c1c2c4c5 : ".ddr_private_c1c2c4c5",
    DDR_cacheable_sys_private_text_c0c3 : ".ddr_private_c0c3"
]
c. Append the shared data or code definition in the LCF
   1. Append the definition of data shared by core 0 and core 3 as the listing below shows
Listing: Adding the definition of data shared by core 0 and core 3

```c
unit shared(task0_c0, task0_c3) {
    RENAME "*shared_clc2c4c5.eln","","c1\`.excluded"

    MEMORY {
        m3_shared_c0_c3 ("rw") : org = _M3_SHARED_start+0x200;
        m3_shared_text_c0c3 ("rx") : AFTER(m3_shared_c0_c3);
    }

    SECTIONS {
        m3_part_shared_os_03 {
            .m3.shared_data_c0_c3
            .m3.shared_rom_c0_c3
            .m3.shared_bss_c0_c3
        } > m3_shared_c0_c3;
        m3_part_shared_text {
            .m3.shared_text_c0c3
        } > m3_shared_text_c0c3;
    }
}

address_translation(task0_c0, task0_c3) {
    m3_shared_c0_c3 (SHARED_DATA_MMU_DEF_REGA,
                    SHARED_DATA_MMU_DEF_REGC):M3;
    m3_shared_text_c0c3 (SYSTEM_PROG_MMU_DEF_REGA,
                         SYSTEM_PROG_MMU_DEF_REGC):M3;
}
```

NOTE

Using the `RENAME` directive is mandatory because the same function name, `foo`, is used for the two shared spaces. The `EXCLUDE` directive cannot be used as it excludes the `foo` name itself, and generates a linking error.

2. Append the definition of data shared by core 1, core 2, core 4 and core 5 as the listing below shows

Listing: Adding the definition of data shared by core 1, core 2, core 4 and core 5

```c
unit shared (task0_c1, task0_c2, task0_c4, task0_c5) {
    RENAME "*shared_c0c3.eln","","c0\`.excluded"

    MEMORY {
```
m3_shared_c1_c2_c4_c5 ("rw") : org = _M3_SHARED_start+0x400;
m3_shared_text_c1c2c4c5 ("rx") : AFTER(m3_shared_c1_c2_c4_c5);
}

SECTIONS {
  m3_part_shared_os_1245 {
    .m3_shared_data_c1_c2_c4_c5
    .m3_shared_rom_c1_c2_c4_c5
    .m3_shared_bss_c1_c2_c4_c5
} > m3_shared_c1_c2_c4_c5;
  m3_part_shared_text_1245 {
    .m3_shared_text_c1c2c4c5
} > m3_shared_text_c1c2c4c5;
}

address_translation(task0_c1, task0_c2, task0_c4, task0_c5) {
  m3_shared_c1_c2_c4_c5 (SHARED_DATA_MMU_DEF_REGA, SHARED_DATA_MMU_DEF_REGC):M3;
  m3_shared_text_c1c2c4c5 (SYSTEM_PROG_MMU_DEF_REGA, SYSTEM_PROG_MMU_DEF_REGC):M3;
}

3. Make sure that the section .default is defined private in the LCF. The listing below shows an example.

**Listing: Example of private .default section**

```c
unit private (*){
  MEMORY {
    ddr_private_text_c (*): org = _VIRTUAL_DDR_PRIVATE_text_start;
  }

  SECTIONS {
    descriptor__ddr__cacheable__sys__private__text {
      .ddr__cacheable__sys__private__text
      .default
    } > ddr_private_text_c;
  }

  address_translation (*){
    ddr_private_text_c (SYSTEM_PROG_MMU_DEF_REGA, SYSTEM_PROG_MMU_DEF_REGC): DDR, org =
```
NOTE
The .default section is created for each file and contains references to data or code defined in the file. Because of partial sharing, and in order to avoid exclusions, the .default section must be private.

Related Tasks
- How to Define Private Data Sections for Multiple Cores
- How to Limit Code and Data Visibility at Core Level

Related References
- Linker Predefinitions

2.2.8 How to Limit Code and Data Visibility at Core Level

Follow the steps described in this task when you need to limit the visibility of the code and/or data sections to specific set of cores. For example, lets assume that for a symbol, you need one definition on core 0, 1 and 2, and another definition on core 3, 4 and 5.

Follow these steps to implement the example scenario:

1. Create two source files

   This step is required because the compiler application configuration file and the LCF deal with the linker sections differently. Table 2-3 lists more details.

<table>
<thead>
<tr>
<th>File Name</th>
<th>Code Listing</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>shared_c0c1c2.c</td>
<td>Listing: Contents of shared_c0c1c2.c</td>
<td>Data visible to cores c0, c1 and c2</td>
</tr>
<tr>
<td>shared_c3c4c5.c</td>
<td>Listing: Contents of shared_c3c4c5.c</td>
<td>Data visible to cores c3, c4 and c5</td>
</tr>
</tbody>
</table>

Listing: Contents of shared_c0c1c2.c
extern char buffer[] = "shared c0,c1,c2";

Listing: Contents of shared_c3c4c5.c

extern char buffer[] = "shared c3,c4,c5";

2. Modify the application configuration file

Each file listed in Table 2-3 must be described in the application configuration file in order to define the section name and visibility as the below listing shows.

Listing: Modified application configuration file

section
  program = [
    shared_text_c0c1c2 : ".text" core = "c0\`c1\`c2",
    shared_text_c3c4c5 : ".text" core = "c3\`c4\`c5"
  ]

data = [
  shared_data_c0c1c2 : ".data" core = "c0\`c1\`c2",
  shared_data_c3c4c5 : ".data" core = "c3\`c4\`c5"
]

rom = [
  shared_rom_c0c1c2 : ".rom" core = "c0\`c1\`c2",
  shared_rom_c3c4c5 : ".rom" core = "c3\`c4\`c5"
]

bss = [
  shared_bss_c0c1c2 : ".bss" core = "c0\`c1\`c2",
  shared_bss_c3c4c5 : ".bss" core = "c3\`c4\`c5"
]
end section

module "shared_c0c1c2" [
  program = shared_text_c0c1c2
  data = shared_data_c0c1c2
  rom = shared_rom_c0c1c2
  bss = shared_bss_c0c1c2
]

module "shared_c3c4c5" [
You can obtain the same result by using pragmas or attribute directly in source files. See Listing: Contents of shared_c0c1c2.c using attributes and Listing: Contents of shared_c3c4c5.c using pragmas.

Listing: Contents of shared_c0c1c2.c using attributes

```c
// __attribute__((section("c0`c1`c2`.data"))) extern char buffer[]= "shared c0,c1,c2";
```

Listing: Contents of shared_c3c4c5.c using pragmas

```c
// #pragma data_seg_name "c0`c1`c2`.data"
extern char buffer[]= "shared c0,c1,c2";
```

3. Append the shared data or code definition in the LCF as the listing below shows

Listing: Definition of shared data or code in the LCF

```c
/**
 * Shared c0,c1,c2
 */
unit shared (task0_c0,task0_c1, task0_c2) {
    MEMORY {
        mem_shared_data_c0c1c2 ("rw"): org = _VIRT_SHARED_cores_start;
        mem_shared_text_c0c1c2 ("rx"): AFTER(mem_shared_data_c0c1c2);
    }
}
SECTIONS {
    descriptor__shared_data_c0c1c2 {
        "c0`c1`c2`.data"
        "c0`c1`c2`.rom"
        "c0`c1`c2`.bss"
    } > mem_shared_data_c0c1c2;
    descriptor__shared_text_c0c1c2 {
        "c0`c1`c2`.text"
    } > mem_shared_text_c0c1c2;
```
address_translation (task0_c0, task0_c1, task0_c2) {
    mem_shared_data_c0c1c2 (SYSTEM_DATA_MMU_DEF_REGA, SYSTEM_DATA_MMU_DEF_REGC): M3;
    mem_shared_text_c0c1c2 (SYSTEM_PROG_MMU_DEF_REGA, SYSTEM_PROG_MMU_DEF_REGC): M3;
}

/**
* Shared c3, c4, c5
*/

unit shared (task0_c3, task0_c4, task0_c5) {
    MEMORY {
        mem_shared_data_c3c4c5 ("rw"): org = _VIRT_SHARED_cores_start;
        mem_shared_text_c3c4c5 ("rx"): AFTER(mem_shared_data_c3c4c5);
    }
    SECTIONS {
        descriptor__shared_data_c3c4c5 {
            "c3`c4`c5`.data"
            "c3`c4`c5`.rom"
            "c3`c4`c5`.bss"
        } > mem_shared_data_c3c4c5;
        descriptor__shared_text_c3c4c5 {
            "c3`c4`c5`.text"
        } > mem_shared_text_c3c4c5;
    }
}

address_translation (task0_c3, task0_c4, task0_c5) {
    mem_shared_data_c3c4c5 (SYSTEM_DATA_MMU_DEF_REGA, SYSTEM_DATA_MMU_DEF_REGC): M3;
    mem_shared_text_c3c4c5 (SYSTEM_PROG_MMU_DEF_REGA, SYSTEM_PROG_MMU_DEF_REGC): M3;
}

Related Tasks

- How to Define Private Data Sections for Multiple Cores
- How to Share Code and Data Partially Among Different Cores

Related References
2.2.9 How to Define Unlikely Block of Code as Private Block of Code in a Multi-core Application

When you specify the unlikely keyword for a block of code in your application, the compiler moves that block of code to the .unlikely section in the LCF.

In a multi-core application, however, specifying the unlikely keyword for private block of code on multiple cores leads to linking errors because, by default, unlikely blocks of code are not considered as private blocks of code.

In order to avoid such linking errors, follow these steps to explicitly define unlikely blocks of code as private blocks of code in a multi-core application.

2.2.9.1 Scenario 1: True Private Code Model

1. Rename the relevant .unlikely section to make it specific to each of the cores either by using core="c0" in the application configuration file, or by using the RENAME directive in the LCF.

   For example, for C0 private functions defined in the module COM2function.c, use the following linker directive:

   RENAME "*COM2function.eln", ".unlikely", "c0`.unlikely_COM2function"

2. Place the renamed section under output section definition that belongs to the private memory areas specific to the hosting core. For example, under the UNIT PRIVATE (task0_c0) construct.

2.2.9.2 Scenario 2: Code Partially Shared Among Different Cores

If a subset of cores share the code, or the code is partially shared among different cores (function code is shared only between core 0 and core 1), then rename the unlikely section on the non-sharing cores set.

The section so renamed becomes hidden for the non-sharing cores set.
For example, let's consider a scenario where partial code is shared between core C0 and core C1, and the functions are defined in module C0C1_DDRfunction.c. You use the following linker directive for renaming the unlikely section on the non-sharing cores set:

```c
UNIT PRIVATE (task0_c2, task0_c3, task0_c4, task0_c5){
    RENAME "*C0C1_DDRfunction.eln", ".unlikely", "c0`.remove"
    ...
}
```

Related Tasks

- How to Define Private Data Sections for Multiple Cores
- How to Share Code and Data Partially Among Different Cores

Related References

- Linker Predefinitions

2.2.10 How to Run Multiple Tasks on the Same Core

Follow the steps given below in order to run multiple tasks on the same core.

1. Specify only user-defined tasks. For example, the listing below shows two tasks, task1 and task2, both running on core c0.

   **Listing: Specifying only user-defined tasks**

   ```c
   TASKS {
     //  core_name : task_name task_id prog_id  data_id
     c0        : task1,    1,      1,       1;
     c0        : task2,    3,      3,       3;
   }
   ```

2. Make sure that the tasks sharing the same core must also share the .att_mmu section. The listing below shows an example.

   **Listing: Multiple tasks sharing the .att_mmu section**

   ```c
   unit shared(*) { 
     MEMORY {
       local_data_descriptor ("rw"): org = _VirtLocalDataDDR_b;
     }
     SECTIONS {
       descriptor_local_data {
         LNK_SECTION(att_mmu, "rw", 0x200, 4, ".att_mmu");
     }
3. Place the task-specific private data/code, if any, under task-specific units.

**NOTE**
It is not supported to have private units that refer to multiple tasks running on the same core.

The listing below shows an example where the linker does not duplicate the task-specific private code/data sections because the tasks are hosted on the same core.

**Listing: Incorrect approach to specify task-specific private code/data**

```
// !!!
////////// Incorrect approach to have private code in two tasks
////////// running on the same core
// !!!
unit private(*) {
MEMORY {
    Task_pgm ("rx"): org = 0x20000000;
    Swi_data ("rw"): org = 0x20000000;
}
SECTIONS {
    task_pgm_output_section {
        .task_pgm
    } > Task_pgm;
    swi_data_output_section {
        .swi_data
    } > Swi_data;
}
```

The listing below shows the right approach.

**Listing: Correct approach to specify task-specific private code/data**
unit private (task1) {
  MEMORY {
    Task1_pgm ("rx"): org = 0x20000000;
    Swi1_data ("rw"): org = 0x20000000;
  }

  SECTIONS {
    task1_pgm_output_section {
      .task1_pgm
    } > Task1_pgm;

    swi1_data_output_section {
      .swi1_data
    } > Swi1_data;
  }
}

unit private (task2) {
  MEMORY {
    Task2_pgm ("rx"): org = 0x20000000;
    Swi2_data ("rw"): org = 0x20000000;
  }

  SECTIONS {
    task2_pgm_output_section {
      .task2_pgm
    } > Task2_pgm;

    swi2_data_output_section {
      .swi2_data
    } > Swi2_data;
  }
}

address_translation (task1) {
  Task1_pgm (SYSTEM_PROG_MMU_DEF_REGA, SYSTEM_PROG_MMU_DEF_REGC): LOCAL_M2;
  Swi1_data (SYSTEM_DATA_MMU_DEF_REGA, SYSTEM_DATA_MMU_DEF_REGC): LOCAL_M2;
}

address_translation (task2) {
2.2.11 How to Make Code or Data Sections Visible to a Subset of Cores

In order to make specific code or data sections visible to a subset of cores, it is recommended that you use the `rename` directive, and follow the naming convention to restrict the section visibility per core.

For example, sections with name prefixed by `c0` are visible only to core 0, sections with name prefixed by `c1` are visible only to core 1, and so on.

If you need to make a compiler generated data section, for example, the `.data` section, visible only to core `c4`, then specify the following command in the LCF file. Note that this setting is applicable at the global level.

```
rename ".testfile.eln", ".data", "c4\..local_data";
```

In order to set the visibility to a subset of cores, for example, core `c2`, core `c4`, and core `c5`, you must rename the section for exclusion in the cores that do not need that section. For example:

```
unit private (task0_c0, task0_c1, task0_c3) {
    rename ".testfile.eln", ".data", "c2\..exclude";
}
```

However, note that the same section is placed with its original name in the cores where it is visible.

Related Tasks

- How to Define Private Data Sections for Multiple Cores
- How to Share Code and Data Partially Among Different Cores

Related References

- Linker Predefinitions

2.2.12 How to place a symbol in an another section in LCF
Follow these steps:

1. Make sure that you build your application or library with the `-Xllt --one_symb_per_sect` option. Note that `-Xllt` in the preceding command represents compiler's low-level optimizer.
2. Place a `SYMBOL` command in the appropriate unit using following syntax. Table 2-4 provides description of the syntax.

   `<descName> { 
   SYMBOL "<moduleName>" ("<sectionName>") ("<symbolName>") 
} > <outputSection>`

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;descName&gt;</code></td>
<td>Name of the descriptor where you want to allocate the <code>&lt;outputSection&gt;</code> section</td>
</tr>
<tr>
<td><code>&lt;outputSection&gt;</code></td>
<td>Name of the section where you want to place the object</td>
</tr>
<tr>
<td><code>&lt;moduleName&gt;</code></td>
<td>Name of the module where the symbol is defined. Specify the full path to the <code>.elx</code> file name. Note that you can use wildcard characters if you want to.</td>
</tr>
<tr>
<td><code>&lt;sectionName&gt;</code></td>
<td>Name of the section where the object is allocated in the <code>&lt;moduleName&gt;</code> file</td>
</tr>
<tr>
<td><code>&lt;symbolname&gt;</code></td>
<td>Name of the symbol you want to allocate in the <code>&lt;outputSection&gt;</code> section. Specify the mangled name.</td>
</tr>
</tbody>
</table>

**Example**

The listing below shows an example of how a function, say `myFunc` that is implemented in the `myFile.c` module and placed in the `.text` section, is allocated in the `mySection` section in shared DDR memory. The code in the listing below, belongs to the LCF, the `.l3k` file.

**Listing: Example - How to place a function in another section in the LCF**

```c
unit_shared (*) { 
    MEMORY {
        ...
        ddr_shared_data_nc_wt ("rw"): org = _DDR_SHARED_start;
        ddr_shared_data_c_wb ("rw"): AFTER(ddr_shared_data_nc_wt);
        ddr_shared_text_c ("rx"): AFTER(ddr_shared_data_c_wb);
        mySection ("rx"): AFTER(ddr_shared_text_c);
    }
    SECTIONS {
```

Freescale Semiconductor, Inc. 63
2.2.13 How to Handle C++ Templates in Multi-core Applications

This task explains how to workaround the problems that might occur when compiling and linking the multi-core applications that contain the C++ templates.

When you use C++ templates in a multi-core application, the StarCore compiler generates specific functions in each of the modules/source code files where the template is used.

Consider the following C++ template in a header file:

```cpp
template <class T> class mycontainer {
    T element;

    public:
        mycontainer (T arg) {element=arg;}
        T increase () {return ++element;}
};
```

Further, consider that two source code files, file1.cpp and file2.cpp, define objects for this class as:

```cpp```

Related Tasks

- How to specify the content of virtual memory areas
- How to Make Code or Data Sections Visible to a Subset of Cores

Related References

- Linker Predefinitions
mycontainer<int> myint (7);

Now, the object files for file1.cpp and file2.cpp will have the __ZN11mycontainerIiEC1Ei constructor function. Note that the __ZN11mycontainerIiEC1Ei function is actually a C++ mangled name for mycontainer<int>::mycontainer(int).

The symbols, such as __ZN11mycontainerIiEC1Ei, are attributed by compiler as being MULTIDEF, which means that the linker does not exit with an error message about multiple definitions of same symbol. Instead, the linker picks one of the symbol definitions and continue the processing.

However, the linker might not select an appropriate symbol when linking a multi-core application because of various types of symbol sharing, such as, private, fully shared, partially shared etc.

For example, consider that:

- there are references to template instance code from private space of each core
- there are references to template instance code from shared space of all cores (for example, from library code, which is generally shared)

In this case, the current StarCore linker tries to resolve the references by keeping the definitions in private space. If the definitions are not placed at the same virtual address, the references from shared space are not correctly resolved and the linker exits with the following error message:

[LNK,2,6999,-1]: Error(E2005): in core c0: Symbol resolution: found inconsistent address for symbol '__ZN11mycontainerIiEC1Ei' which is defined in section 'c1`.text_entry'.

As a workaround, place the full library or at least the modules using the C++ templates in each private space.

Related Tasks

- How to Define the Shared Memory
- How to troubleshoot linker error messages

Related References

- Linker Predefinitions

2.2.14 How to Check Local Symbols Addresses
Linker can check if the private local symbols accessed from the shared space have same address on all cores. In order for the linker to check local symbol address, use the `--check-locals` switch.

**NOTE**

Local symbols checking is time consuming and is not enabled by default.

If the linker finds private local symbols that are accessed from the shared space and have different addresses on different cores, the linker throws an error message and terminates the process.

### 2.2.15 How to use KEEP Directive

The KEEP directive is used to prevent from stripping the selected symbols, a section or the entire contents of an object file.

1. The syntax for using the KEEP directive to avoid stripping a selected symbol is as follows:

   ```
   KEEP (SYMBOL "object_file_name_pattern" (section_name) (symbol_name));
   ```

   **Example**

   To avoid stripping of the unused `foo` function defined in the file `my_keep_file.c` part of the `.text` section, use the following construct:

   ```
   unit shared (*)
   {
      KEEP (SYMBOL "*my_keep_file.eln" (.text) (_foo));
   }
   ```

2. The syntax for using the KEEP directive to avoid stripping the symbols from a section is as follows:

   ```
   KEEP ("object_file_name_pattern" (section_name));
   ```

   In this case all symbols containing in the section `section_name` are prevented from stripping.

3. The syntax for using the KEEP directive to avoid stripping the symbols from an entire object file is as follows:

   ```
   KEEP ("object_file_name_pattern");
   ```

   In this case all symbols defined in the `object_file_name` regardless of the containing section are prevented from stripping.
2.2.16 How to reserve an MMU descriptor ID

In order to reserve a certain descriptor ID, the following directive can be used:

```c
reserve_descriptor_id(TYPE, ID);
```

The directive has the following two parameters:

* **TYPE**: represents the type of the descriptor, with two possible values: “data” or “code”;
* **ID**: the id of the descriptor to be reserved. The ID can be then used in the application code.

The directive can be used at UNIT level. For example,

```c
unit shared(*)
{
    reserve_descriptor_id("data", 1);
}
```
General linker tasks
Chapter 3
Concepts

This chapter consists of linker concepts that you might need to comprehend to accomplish linker tasks.

In this chapter:

• Linker configuration concepts
• General linker concepts

3.1 Linker configuration concepts

This chapter describes the linker configuration concepts.

In this section:

• Understanding linker terminology
• Understanding SC3000 LCF syntax
• Understanding Cache Optimization in SC3000 Linker
• Understanding Flexible Startup Configuration

3.1.1 Understanding linker terminology

Table 3-1 describes the linker terminology.

<table>
<thead>
<tr>
<th>Term</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>A core is a hardware processing unit. The linker defines code, data, sections etc. for the core.</td>
</tr>
</tbody>
</table>

Table continues on the next page...
Table 3-1. Linker terminology (continued)

<table>
<thead>
<tr>
<th>Term</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task</td>
<td>A task is a static software unit. Each task has a unique ID that the OS and/or the hardware recognize. One or more tasks can be mapped to a single core.</td>
</tr>
<tr>
<td>Input Section</td>
<td>An input section is a fragment of a user or data code in object files.</td>
</tr>
<tr>
<td>Output Section</td>
<td>An output section is a collection of input sections. The output section is a part of the final executable file.</td>
</tr>
<tr>
<td>Unit</td>
<td>A unit is an LCF language construct that is used to enclose memory and section definitions for an explicit or implicit set of tasks.</td>
</tr>
<tr>
<td>Private Virtual Memory</td>
<td>Private virtual memory is a memory area that is accessible to a single task.</td>
</tr>
<tr>
<td>Private Physical Memory</td>
<td>Private physical memory is a memory area that is accessible to a single core.</td>
</tr>
<tr>
<td>Shared Virtual Memory</td>
<td>Shared virtual memory is a memory area that is accessible by a set of tasks.</td>
</tr>
<tr>
<td>Shared Physical Memory</td>
<td>Shared physical memory is a memory area that is accessible to a set of cores.</td>
</tr>
<tr>
<td>Address Translation</td>
<td>Address translation specifies the mapping from a virtual memory range to the physical address or a physical memory range.</td>
</tr>
</tbody>
</table>

Related Concepts

- Understanding SC3000 LCF syntax

Related References

- LCF Expression Functions
- LCF Expression Operators

3.1.2 Understanding SC3000 LCF syntax

This section helps you understand SC3000 LCF syntax.

3.1.2.1 Using naming conventions

- Non-terminal entity is represented in lower-case
- Terminal entity is represented in UPPER-CASE
- String enclosed in quotes is interpreted as is
3.1.2.2 Specifying integers

The SC3000 parser supports following types of integers:

- an octal integer, which is a zero followed by zero or more octal digits. An octal integer uses one or more of these digits: 0,1,2,3,4,5,6,7.
- a decimal integer, which starts with a non-zero digit followed by zero or more digits. A decimal integer uses one or more of these digits: 0,1,2,3,4,5,6,7,8,9.
- a hexadecimal integer, which starts with a '0x' or '0X' followed by one or more hexadecimal digits. A hexadecimal integer uses one or more of these digits: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F.

3.1.2.3 Specifying symbol names

Symbol names in SC3000 LCF can be:

- defined without quotes - use when the symbol name does not conflict with any linker keyword. A symbol name defined without double quotes must start with:
  - a letter
  - an underscore
  - a point
- defined with double quotes (""') - use when the symbol name matches one of the linker keywords. A symbol name defined with double quotes can include odd characters.

A symbol name can include any letters, underscores, digits and points.

3.1.2.4 Specifying global directives

At the global level, you specify only global settings, self-contained directives, and top level containers, such as:

- arch
- unit
- physical_memory
- address_translation
- task_definition
3.1.2.5 Specifying target architecture

A predefined variable, number_of_cores, specifies the number of cores available during the linking operation. Cores are referred by predefined names, such as c0, c1, c2, etc.

NOTE
You cannot change the predefined core names.

You specify the target architecture using the arch directive. For example:

```
arch = ARCH '{' arch_name '}'
arch_name = STRING;
end = ';' | ',';
```

The arch_name parameter accepts these values:

- B4860

3.1.2.6 Defining tasks

Tasks are defined for each core.

NOTE
If you do not define a task explicitly, a core runs one task by default.

The syntax for defining a task is:

```
task_definition = TASKS '{'
( core_name ':' task_name ',' task_id ','
  prog_id ',' data_id ';' )*
'}';
```

where,

- core_name is a STRING, and can be c0, c1, c2 etc.
• task_name is a STRING
• task_id is an INTEGER
• prog_id is an INTEGER
• data_id is an INTEGER

3.1.2.7 Defining virtual memory and output sections

The unit directive defines the virtual memory and the output sections for a task or a list of tasks. The syntax for defining a unit directive is:

```
unit = UNIT (PRIVATE|SHARED) task_list
    '{'
    unit_statements;
    '}'
```

The PRIVATE attribute specifies private content for multiple tasks. The output sections or the virtual memory blocks that the unit directive defines, are private to these tasks.

The SHARED attribute specifies a shared memory or an output configuration for multiple tasks. The symbols defined are shared among tasks.

You specify a task in the task list using:

• its name, or
• using a task name pattern

The listing below shows the syntax for specifying a task in the task list.

Listing: Specifying a Task in the Task List

```
task_list = '{' task_list_b '};
task_list_b = task_sp ['',' task_list_b];
    |task_name
    |pattern
```

In addition to the virtual memory blocks and the output sections, a unit directive also includes assignments and statements, such as entry, assert, keep, rename, and exclude. The listing below shows the syntax.

Listing: The unit directive syntax

```
unit_statements = (unit_statement)*
unit_statement = sections
    |memory_specifications
```
3.1.2.8 Configuring the virtual memory

You configure the memory with virtual memory blocks. For embedded systems, the linker generates several non-consecutive virtual memory blocks. The blocks are non-consecutive because there are multiple physical memory peripherals, and the linker generates physical to virtual mapping schemes, such as 1:1 mapping.

You configure the memory using the MEMORY function. The listing below shows the syntax.

Listing: Configuring the memory using the MEMORY function

```c
memory_specifications = MEMORY '{' mem_specs '}';
mem_specs = (mem_spec)*;
mem_spec = memory_name
    [RESERVE]
    [attributes]
    [page]
    [':' org_and_len]
    [;]
;```

memory_name = STRING;
org_and_len = org | len | (org, len);
org = ORG '=' exp
     | AFTER '(' memory ')';
len = LEN '=' exp;
attributes = '()' STRING ');
page = DATA | PROGRAM;
entry = ENTRY '(' symbol_name ')';
symbol_name = STRING;
keep = KEEP '(' input_section_spec_no_keep ')';
assert = ASSERT '(' exp ',' message_string ')';
message_string = STRING;

NOTE
If you do not specify the len parameter, the linker calculates the virtual memory size by default. If the size that the linker calculates is larger than what the len parameter specifies, a fatal link error occurs.

The reserve qualifier specifies that the virtual memory block is reserved for the dynamic sections.

You can optionally specify attributes, which can be used for automatic memory layout and verification. You may also specify attributes that are not built-in in the hardware.

Table 3-2 lists the currently supported attributes.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'R'</td>
<td>Read-only</td>
</tr>
<tr>
<td>'W'</td>
<td>Read/Write</td>
</tr>
<tr>
<td>'X'</td>
<td>Sections that contain executable code</td>
</tr>
<tr>
<td>'A'</td>
<td>Allocated</td>
</tr>
<tr>
<td>'I'</td>
<td>Initialized</td>
</tr>
<tr>
<td>'!'</td>
<td>Inverts the meaning of the corresponding attribute</td>
</tr>
</tbody>
</table>

3.1.2.9 Creating an output section

To create an output section, you use the linker directives to specify:
• the memory (and file) layout of the output section
• how an input section is placed in the output section
• how to interpret the symbols related to the section placement
• other definitions, such as ENTRY

The listing below shows the syntax for creating an output section.

**Listing: Creating an output section**

```c
sections: SECTIONS '{' sec_or_group_p1 '}'
sec_or_group_p1 = sec_or_group_p1 section
| sec_or_group_p1 statement_anywhere;
section: STRING | 'NONAME'
| [q_accessibility] [q_attribute] [q_align]
| '{' statement_list_opt '}' '>' memory_name ';'
| [fill_opt];
q_accessibility = '(' [ 'r' | 'w' | 'x' ] ')'
q_attribute = ATTRIBUTE '(' '+' 'z' ')'
fill_opt = FILL '(' STRING ')';
```

• The `q_accessibility` parameter specifies the accessibility of the output section. The accessibility defines the default placement of:
  • the corresponding input sections
  • the output section in the memory

• The `statement_list_opt` parameter contains a list of statements. A statement can be an assignment, an assertion, or generally an `input_section_spec`. The `input_section_spec` parameter is a wildcard to match the input sections.

• The `fill_opt` parameter specifies the fill pattern for the output section.

The listing below shows the syntax for creating an output section.

**Listing: Statement Definition**

```c
statement_list_opt = (statement)*;
statement = assignment ';'
| assert ';'
| special_sections [end]
| input_section_spec
| lnk_section_spec
| unmatch_spec [end]
| LEN '(' exp ')'
| FILL '(' STRING ')'
;
pattern = STRING;
input_section_spec:
| input_section_spec_no_keep
| KEEP '{' input_section_spec_no_keep '}'
;
input_section_spec_no_keep = file_or_sec_name [section_name] [ATTR attributes] [end]
| SYMBOL file_name [section_name] symbol [end]
file_or_sec_name = STRING | '::*';
section_name = '('symbol_name ')';
symbol = '(' 'symbol_name ')';
lnk_section_spec =
```

**CodeWarrior Development Studio for StarCore 3900FP DSP Architectures SC3000 Linker Reference Manual, Rev. 10.9.0, 11/2015**

Freescale Semiconductor, Inc.
The length parameter fills the raw material, such as, BYTE(0X33).

The input_section_spec parameter places the input sections. The parameter consists of a number of wildcard specifications to match input contents by the file, section, and symbol name. You can specify the filename as * to includes all the files. When you do not use the wildcards to specify the input section name, the linker places the section per section's attributes.

The file_or_sec_name parameter indicates:

- a file name, if section_name also exists, e.g. "my_file (.text)" indicates all sections named ".text" from a file named my_file
- a section name, if no section_name exists, e.g. ".data" indicates all sections named ".data" from all input modules

The LNK_SECTION parameter defines the bss, heap, stack, and att_mmu sections.

The unmatch_spec parameter lets you include the unmatched sections in an output section by using one of the following values:

- UNMATCH_PGM
  Includes all the program sections that are not defined in any other directive, and which match the specified pattern

- UNMATCH_DATA
  Includes all the data sections that are not defined in any other directive, and which match the specified pattern

- UNMATCH_BSS
  Includes all the uninitialized data sections that are not defined in any other directive, and which match the specified pattern

- UNMATCH_ROM
  Includes all the read-only data sections that are not defined in any other directive, and which match the specified pattern

3.1.2.10 Configuring the Physical Memory
You define the physical memory as:

- private to a core, or
- symmetric to several cores, or
- shared with other cores

You may also define some reserved area in the physical memory. The listing below shows the syntax for configuring physical memory.

**Listing: Configuring Physical Memory**

```plaintext
physical_memory = PHYSICAL_MEMORY
               ( private [core_list]
               | shared [core_list])
               '{'
               (NAME ':' [attributes] d_org_and_len ';' |
               RESERVE ':' d_org_and_len ';')*
               '}'
               ;

core_list = '( ' core_name (',' core_name)* ' )';
d_org_and_len = ORG '=' exp ',' LEN '=' exp;
```

The `core_name` parameter is obtained by prefixing the core id with the letter 'c'. Therefore, the valid values are `c0`, `c1`, `c2`, etc. The linker also supports wildcards for specifying the core names.

### 3.1.2.11 Specifying Address Translation Construct

The `address_translation` construct is an array of address translation entries. Every entry specifies how a virtual memory block is mapped to a physical memory fragment.

The listing below shows the syntax for creating an address translation entry.

**Listing: The address_translation Directive Syntax**

```plaintext
address_translation = ADDRESS_TRANSLATION task_list [MAP11]
               '{'
               (address_translation_entries | reserve_entries)*
               '}'
               ;

address_translation_entry = VIRTUAL_MEMORY_NAME [mmu_attributes]
               [MAP11] physical_memory_range [descriptor(ID)][''];

mmu_attributes = '('{{ STRING, STRING '})';
```
In the above listing, flags is a set of memory attributes such as rw, rx.

### 3.1.2.12 Linking self-contained libraries

If you enable the self-contained library linking, you cannot use any other directives except a few, which are related to the self contained library linking.

The listing below shows the syntax for creating self-contained library linking.

**Listing: Creating Self-contained library linking**

```
.library_entry "entry_symbol" (',' "entry_symbol")*
.undefined_function "function" (',' "function")*
.library_prefix "prefix_name"
.library_public_symbols "public_symbol"
    (',' "public_symbol")*
.library_concatenate_sections "name" ','
    "section_name_pattern"
    (',' "section_name_pattern")*
```

**Related Tasks**

- How to create a Linker Command File (LCF)

**Related References**

- Command-Line Options
- Sections in LCF

### 3.1.3 Understanding Cache Optimization in SC3000 Linker

The goal of cache optimization is to place routines near their callers in the virtual memory in order to reduce paging traffic, and to place frequently used and related routines in a way such that they are less likely to collide with each other in the I/D-cache.

To enable cache optimization in the linker, use -set-cache1 option. For example:
You must use PROFILE_INFO construct in the LCF to instruct the linker to perform the cache optimization. The listing below shows the syntax:

**Listing: PROFILE_INFO Syntax**

PROFILE_INFO (core_name_list)
{

FUNCTIONS {

  fname TIME(time1) [ MODULE(module_name) [IS_MANGLED(+/-)]]{
    CALLS(f2name, times_x);
    CALLS(f3name, times_y);
    ...
  }

  f2name TIME(time2) [...] {
    CALLS(f3name, times_z);
    ...
  }

}

In the above listing:

- the attributes MODULE and IS_MANGLED are optional; however, when defined, the value of MODULE must be the definition file name
- fname, f2name, and f3name represent the name of the functions. The function name is mangled or un-mangled, based on the value of the IS_MANGLED attribute; a + sign indicates a mangled name while a - sign represents an un-mangled name. By default, the function name is considered as mangled.
- time1 and time2 represent the average cycle count spent in the function
- the CALLS intrinsic specifies which the children functions are, and how often they are called during a parent-function call

The listing below shows an example.

**Listing: PROFILE_INFO Example**

profile_info (c0) {
  functions {
    _main TIME(3/2) IS_MANGLED(+) {
      CALLS(_printf,2);
    }

    _printf TIME (1123) IS_MANGLED(+) {

  }

CodeWarrior Development Studio for StarCore 3900FP DSP Architectures SC3000 Linker Reference Manual, Rev. 10.9.0, 11/2015
In case the frequency information is not provided in the LCF, the linker generates the required frequency information automatically by counting the call number; excluding the loops.

The linker lets you optimize only specific output sections. To do this, you must add the cache_optimized attribute for the relevant section by adding ATTRIBUTE (+c) when defining the output section. The listing below shows an example.

**Listing: Optimizing Only Specific Output Sections**

```plaintext
unit private (*){
    MEMORY {
        ...
        memory_code ("rwx"):org = _CodeStart;
        ....
    }
    SECTIONS {
        ...
        .text attribute (+c) {
            .text
            .default
        } > memory_code;
        ...
    }
}
```

3.1.4 Understanding Flexible Startup Configuration

The Flexible Startup configuration lets you place the following sections independent of any core:

- `.att_mmu` section
- Stack
- Heap
3.1.4.1 Changes Made to Support Flexible Startup Configuration

The startup code is modified so that it behaves as SIMD configuration code for stack, heap, .att_mmu, C++ exceptions etc. The linker is modified to support this behavior and to automatically define the following symbols relevant to each core.

**CAUTION**

The following symbols must not be redefined, either by LCF or by the C/C++ source.

- __LNK_Local_StackStart_cX
- __LNK_Local_TopOfStack_cX
- __LNK_TopOfHeap_cX
- __LNK_BottomOfHeap_cX

Following are the C++ specific symbols:

- __LNK_cpp_Static_init_start_cX
- __LNK_cpp_Static_init_end_cX
- __LNK_ExceptionTable_start_cX
- __LNK_ExceptionTable_end_cX

The X in the symbols name represents the core number that can be any number between 0 and 5.

**Related Tasks**

- How to Make LCF Compatible for Flexible Startup

3.2 General linker concepts

This chapter describes the general linker concepts.

In this section:

- Understanding startup environment
- Understanding Flexible Segment Programming Model
- Understanding L1 Defense
3.2.1 Understanding startup environment

The linker startup code consists of these steps:

1. Initialize SR with the default settings
2. Initialize the temp stack pointer
3. Initialize the exception registers to the exception handlers
4. Disable translation and protection (for MMU enabled architectures)
5. Initialize C variables (zero .bss sections)
6. First HOOK (function __target_asm_start)

   This hook will program the MMU exception handlers (M_DESRA0, M_DESRA1, M_PESRA0, M_PESRA1 registers). For more information about the way the descriptor is defined, refer to the topic How to Make LCF Compatible for Flexible Startup.

Table 3-3 shows the First HOOK code in LCF.

<table>
<thead>
<tr>
<th>Code in LCF</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_ENABLE_MMU_TRANSLATION=1;</td>
<td>Enables the MMU translation. If the value is 1, the Address Translation Enable (ATE) bit in the MMU Control Register will be set by the __target_c_start function.</td>
</tr>
<tr>
<td>SYSTEM_DATA_MMU_DEF</td>
<td>Specifies different attributes of MMU</td>
</tr>
</tbody>
</table>

The ATE bit enables or disables the address translation mechanism. If disabled, addresses are not translated (for example, from a virtual address to a physical address). The reset value is configured according to external DSP subsystem plug.

7. Initialize the stack pointer

   The code in LCF for initializing the stack pointer is:

   _StackStart

8. Second HOOK (function __target_c_start())

   The default implementation of this hook is an empty function.

9. Third HOOK (function __target_setting())

   The default implementation of this function is an empty function.
10. Set the `argv` and `argc` arguments
11. Enable the interrupts
12. Jump to `main()` function

### 3.2.2 Understanding Flexible Segment Programming Model

The flexible segment programming model supports the segment sizes of 4 Kbytes up to 16MB - 256 Kbytes, and enables the definition of more flexible data and program segments as compared to Aligned Segment Programming Model.

The b4860 architectures support the flexible segment programming model.

For applications that target b4860 architecture, the linker by default attempts to use the flexible segment model when constructing the MMU descriptors. If all the values of virtual address, physical address, size and boundary of the MMU descriptor conform to the hardware constraints, the flexible segment model is set for that particular descriptor.

See *SC3900 Flexible Vector Processor (FVP) Core Reference Manual* for the hardware constraints that the MMU descriptor must conform to.

If the MMU descriptor does not conform to the hardware constraints, the linker uses the Aligned Segment Programming Model.

You can change the default linker behavior, and explicitly specify the Aligned Segment Programming Model by using the `--disable-flexible-segment-model` command-line option.

### 3.2.3 Understanding L1 Defense

The linker specifies each segment the sharing space:

- If the segment is private, the linker encodes the core id using the cluster id (0,1 or 2) and the core id in cluster (0 or 1).

or

- If the segment is shared, in what cluster (0,1 or 2) the segment is visible.

This is performed using processor specific flags in the `p_flags` member of the segment header. The flags are masked by the `PF_MASKPROC`.
In case one segment is shared between 2 cores of different clusters, the linker considers the segment to be shared between the 2 clusters.

There are two different encodings for each case, as listed below:

1. Segment is private:
   - Bit 28 = 1 (segment is private)
   - Bits 29 and 30 = cluster id
   - Bit 31 = core ID in cluster

2. Segment is shared:
   - Bit 28 = 0 (segment is shared)
   - Bit 29 = if segment is visible in cluster 0
   - Bit 30 = if segment is visible in cluster 1
   - Bit 31 = if segment is visible in cluster 2

<table>
<thead>
<tr>
<th>Sharing \ Bit</th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared</td>
<td>Cluster 2</td>
<td>Cluster 1</td>
<td>Cluster 0</td>
<td>0</td>
</tr>
<tr>
<td>Private</td>
<td>Core id in cluster (0 or 1)</td>
<td>Cluster ID (0, 1 or 2)</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

You can disable this linker behavior using the following command line option:

```
-disable-flag-segment-sharing
```
Chapter 4
Linker Error Messages

This chapter describes the error messages that the StarCore linker may display at various stages of linking operation.

In this appendix:

- Configuration error messages
- Parser error messages
- Setup error messages

4.1 Configuration error messages

The linker may display configuration error messages if the LCF is not configured properly.

4.1.1 EID_ARCH_INCOMPATIBLE_WITH_DIRECTIVE

This error indicates that the linker found a directive incompatible with the selected architecture. For example, for the architectures without MMU, such as sc140e, sc3400, sc3850, only the virtual space related directives are supported. In case an address_translation directive is detected in a project for sc3400, this error is displayed.

(E1079)

To resolve EID_ARCH_INCOMPATIBLE_WITH_DIRECTIVE error

Make sure that the directives you specify in your project are supported for the target architecture.
4.1.2 **EID_EXPR_SECNAME_CANNOT_EVAL**

Indicates the intrinsic function specified in the error message cannot evaluate the intrinsic expression.

```
[LNK,1,6999,-1]: Warning(W1062): LCF configuration: in core c0, task
task0_c0: Intrinsic 'originof' parameter error: Section
'data_boot_c' missing or empty. Intrinsic expression evaluates to 0x0
```

To resolve **EID_EXPR_SECNAME_CANNOT_EVAL** error

Verify that the intrinsic parameter does not depend upon memory layout, or is not an undefined symbol.

4.1.3 **EID_FAIL_LAYOUT**

Indicates the linker has failed in building a memory layout; all allowed number of trials expired without converging to a stable memory layout.

```
[LNK,3,6999,-1]: Fatal(F2007): cannot layout all the sections, possibly it is a LCF issue.
```

To resolve **EID_FAIL_LAYOUT** error

Provide origin and length for memory and address translation constructs to reduce/eliminate the search for establishing a memory layout.

4.1.4 **EID_FAIL_VIRTUAL_LAYOUT**

Indicates that the virtual layout algorithm is not able to find a valid layout for the output sections of the specified virtual memory area.

```
Fatal (F2009)
```

To resolve **EID_FAIL_VIRTUAL_LAYOUT** error

Consider adding the ORG directive for the specified memory area, or reducing the content of the associated output sections.
4.1.5  **EID_FORCE_VALUE_TO_1**

This warning indicates that one of the linker-defined sections (defined using the LNK_SECTION directive) receives as parameter an alignment value that is less than 0x1. Linker by default sets the alignment to 0x1 in such cases.

```
[LNK,1,6999,173,../b4860.l3k]: Warning(W1077): LCF configuration: the alignment expression of LNK_SECTION(heap) is forced to value 0x1.
```

To resolve EID_FORCE_VALUE_TO_1 error

Specify a valid alignment value, bigger than or equal to 0x1, in the specified linker-defined section.

4.1.6  **EID_INCONSISTENT_SYMADDR**

Indicates a symbol has different virtual address in private memory areas and is referenced from a shared memory area.

```
[LNK,2,6999,-1]: Error(E2005): Symbol resolution: found inconsistent address for symbol '_my_private_channel_info' which is defined in section 'c*'.data'.
  Value 0x70004000 from unit c0 in ./Source/private_data_c0.eln
  Value 0x70004100 from unit c1 in ./Source/private_data_c1.eln
```

To resolve EID_INCONSISTENT_SYMADDR error

Symbol must be placed at the same fixed location for all cores private memory.

4.1.7  **EID_LAYOUT_UNRESOLVED**

Indicates the linker was unable to generate a virtual memory layout.

```
[cf/b4860.l3k]: Fatal(F2007): in core c0, task task0_c0: order of section placement in virtual memory m3_shared_data_nc_wt' can not be determined. The following sections can not be placed:    ddr_data_shared
```

To resolve EID_LAYOUT_UNRESOLVED error

Provide the origin and length for memory and address translation statements to reduce/eliminate the search for establishing a memory layout.
4.1.8  EID_LNK_SECTION_TYPE_UNKNOWN

Indicates an unknown type of section that the linker does not recognize.


To resolve EID_LNK_SECTION_TYPE_UNKNOWN error
Make sure that the specified section type is valid.

4.1.9  EID_MORE_AUTO_LAYOUT

Indicates a number of situations, such as:

• exceeded number of trials for reaching a virtual layout
• unspecified virtual memory for an output section
• one virtual memory corresponds more than one physical memory and automatic memory map is disabled (caused by EID_PL_MULTI_MAPPING)
• address translation missing or address translation does not mention org keyword, and automating original address translation is disabled (caused by EID_PL_PORG)
• one virtual memory needs more than one MATT entry, and automatic splitting of virtual memory areas across more than 1 MATT entry is disabled (caused by EID_PL_MMATT)
• one output section is placed into more than one virtual memory, which is currently not supported

[LNK,3,6999,-1]: Error(E2008): higher memory layout automation level needed, not enabled by command line option or not supported yet.

To resolve EID_MORE_AUTO_LAYOUT error
• If the error message is not caused by EID_PL_MULTI_MAPPING, EID_PL_PORG, or EID_PL_MMATT, verify that:
  • there is no output section that has more than one target virtual memory
  • there is no output section without a target virtual memory
• If the error message is caused by EID_PL_MULTI_MAPPING, EID_PL_PORG, or EID_PL_MMATT, see the respective descriptions.

4.1.10  EID_MEM_CANNOT_FINAL

Indicates a problem with the virtual memory definition.

Error(1037)

**To resolve EID_MEM_CANNOT_FINAL error**

Verify the definition for virtual memory, and make sure that org or len expressions could be evaluated, if specified.

### 4.1.11 EID_MEM_INCONSISTENT

Indicates that a shared section is placed in multiple virtual memory locations. Such a placement is invalid.

Error(E1036)

**To resolve EID_MEM_INCONSISTENT error**

Place the shared section in only one virtual memory.

### 4.1.12 EID_MEM_MULTI_AT

Indicates that one virtual memory has multiple entries in the address translation.

[LNK,3,6999,228,../b4860/LCF/b4860.l3k]: Fatal(1034): LCF configuration: multiple address translation entries found for virtual memory 'data_boot_c'.

**To resolve EID_MEM_MULTI_AT error**

Verify the address translation entries so that each virtual memory has only one associated entry. For more information, see How to Define Virtual Memory for Read-Write-Execute (RWX) Access.

### 4.1.13 EID_MEM_NOT_FULLY_SPEC_RESERVE

Indicates that a reserved area in virtual memory space is not fully specified. In other words, either reserved area's origin or its length information is missing or cannot be determined prior to start of the memory layout algorithm.

[LNK,3,6999,-1]: Fatal(1038): LCF configuration: memory local_reserve is reserved and not fully specified with original address and length.

---

**CodeWarrior Development Studio for StarCore 3900FP DSP Architectures SC3000 Linker Reference Manual, Rev. 10.9.0, 11/2015**

Freescale Semiconductor, Inc.
To resolve EID_MEM_NOT_FULLY_SPEC_RESERVE error

Make sure that you specify origin and length of the reserved virtual area as values that do not depend on the outcome of the memory layout algorithm.

4.1.14 EID_MEM_SMALL_MB

Indicates that the specified virtual memory cannot be placed.

[LNK,3,6003,2,.../b4860/LCF/b4860.l3k]: Fatal(F1030): LCF configuration: there is not enough space to place virtual memory 'vml' (org=0xC0000000, size=0x10000). Memory 'vm2' (org=0xC0001000, size=0x400) should be placed further to allow enough space for 'vml'.

To resolve EID_MEM_INCONSISTENT error

Use the ORG or the AFTER directive to place 'vm2' at a higher address in the virtual space to allow successful placement of vm1 in a lower address range.

4.1.15 EID_MEM_UNDEF

Indicates that the virtual memory specified in the error message is referenced in the LCF but is not defined.

[LNK,2,6999,229,newbad.lcf]: Error(E1033): LCF configuration: virtual memory 'm_shared_ddr_x' is not defined.

To resolve EID_MEM_UNDEF error

Define the virtual memory specified in the error message, or remove the reference in the LCF.

4.1.16 EID_MEM_UNDEF_ADDR

This error occurs when the following conditions are true:

1. a virtual memory is defined
2. a section is allocated to the defined virtual memory
3. an address reference in the section allocated in step 2, is out-of-range of the virtual memory defined in step 1
To resolve EID_MEM_UNDEF_ADDR error
Extend the virtual memory definition to include the out-of-range address reference, or change the address reference.

4.1.17  EID_MEM_VIR_NO_PHY

Indicates that linker cannot find a physical memory for the mapped virtual memory from the LCF configuration.

[LNK,2,6999,-1]: Error(E1040): LCF configuration: cannot find physical memory for virtual memory 'vm_data'

To resolve EID_MEM_VIR_NO_PHY error
Map the virtual memory to a physical memory explicitly in the LCF making use of the ORG directive.

4.1.18  EID_ASSERT_FAIL

Indicates that the ASSERT directive in the LCF evaluates to FALSE.

Error(E1019)

To resolve EID_ASSERT_FAIL error
Adjust the condition expression of the assert directive or other related definitions in the LCF.

4.1.19  EID_ATTMMU_SIZE_UNSPECIFIED

Indicates that size for the .att_mmu section is not specified in the LCF. The linker may also display this error message when no command-line option is specified for the linker to automatically allocate size for the .att_mmu section.

[LNK,3,6999,-1]: Fatal(F1017): LCF configuration: att_mmu size specification not found in LCF and automatic att_mmu size is not enabled by command line or supported.

To resolve EID_ATTMMU_SIZE_UNSPECIFIED error
Specify size for the .att_mmu section in the LCF. For example: `lnk_section(att_mmu,"rw", 0x130, 256 );`

4.1.20  **EID_MATT_MAP11_ORG**

Indicates that the starting address of the virtual memory, which is mapped to a physical memory fragment in map11 style, is not consistent with the starting address of the physical memory fragment.

**Error(E1059)**

**To resolve EID_MATT_MAP11_ORG error**

Adjust the address translation in the LCF, so that the virtual memory and the physical memory have the same address spaces.

4.1.21  **EID_MATT_SPEC**

Indicates an error in the address translation.

**Error(E1046)**

**To resolve EID_MATT_SPEC error**

Specify correct address translation.

4.1.22  **EID_MATT_V1ToPn**

Indicates that the virtual memory is mapped to more than one physical memory device in the LCF.

[LNK,3,6999,19,linker_control_files\newlcf\aal.cmd]: Fatal(F1043): LCF configuration: in core c0, task task_1_0: map virtual memory 'm3_share_data' to multiple physical memory.

**To resolve EID_MATT_V1ToPn error**

Specify 1:1 virtual memory to physical memory mapping.
4.1.23 **EID_MATT_VP_UNMATCH**

Indicates that the virtual memory does not properly map to a physical memory fragment.

*Error (E1044)*

**To resolve EID_MATT_VP_UNMATCH error**

Specify 1:1 virtual memory to physical memory mapping.

4.1.24 **EID_MATT_WX**

Indicates that an address_translation entry is set for both data and program sections. An address_translation entry must be set for either data or program sections, but not both.

*Error (E1042)*

**To resolve EID_MATT_WX error**

Specify the address_translation entry separately for data and program sections.

4.1.25 **EID_MATTS_OVERNUMBER**

Indicates that the number of address_translation entries specified for the MMU has crossed the maximum limit.

*Error (E1018)*

**To resolve EID_MATTS_OVERNUMBER error**

- Combine multiple address_translation entries into a single entry
- For more information on maximum number of data and program descriptors, refer to the *DSP Core Reference Manual* of the respective core

4.1.26 **EID_MEM_ADDR_SIZE_UNALIGN**

Indicates that the starting address of the mapped virtual memory does not align with its size.
Configuration error messages

[LNK,1,6999,23,D:\CodeWarrior\Stationery\StarCore\ISS\linker_control_f
iles\newlcf\aa.cmd]: Error(E1031): LCF configuration: in core c0,
task task_1_0: virtual memory 'm3_share_text'(org=0xd0000201,
size=0x00000100) is not aligned to MATT/MMU constraints on address
and size.

To resolve EID_MEM_ADDR_SIZE_UNALIGN error
Make sure that the original address % size of the virtual memory must be equal to zero.

4.1.27 EID_MEM_EMPTY

Indicates an empty section placed in the virtual memory specified in the warning
message.

NOTE
The EID_MEM_EMPTY message is a warning message.

[LNK,1,6999,16,linker_control_files\newlcf\aa3.cmd]: Warning(W1026):
LCF configuration: in core c0, task task_1_0: virtual memory
'm2_share_data_nc' is empty.

To resolve EID_MEM_EMPTY warning
Remove the respective empty section, virtual memory and address translation entries.

4.1.28 EID_MEM_OVERLAP

Indicates an overlap of virtual memory with other virtual memory or reserved memory
fragments.

Error(E1027)

To resolve EID_MEM_OVERLAP error
Verify the memory allocations and make sure that no virtual memory or reserved
memory fragments are overlapped.

4.1.29 EID_MEM_PLACE_INTO_RESERVE

Indicates an attempt to place an output section into a reserved virtual memory.
To resolve EID_MEM_PLACE_INTO_RESERVE error
Use non-reserved virtual memory for placing the output sections.

4.1.30 EID_MEM_REAL_OVERLAP

Indicates overlapped virtual memory space.

To resolve EID_MEM_REAL_OVERLAP error
Modify the org address or size of the virtual memory.

4.1.31 EID_MEM_REAL_OVERLAP_1

Same as EID_MEM_REAL_OVERLAP.

4.1.32 EID_MULTI_ATTMMU_SIZE

Indicates more than one size declaration for the .att_mmu section in the LCF.

To resolve EID_MULTI_ATTMMU_SIZE error
Specify only one size declaration for the .att_mmu section in the LCF.
Indicates that linker was not able to find a valid memory layout for both the virtual and the physical spaces.

Error (E1049)

To resolve EID_PHY_CANNOT_LAYOUT error
Revisit the recent project changes or the changes you have made in the LCF after the previous successful linking process. It could be that one or more virtual memory areas need explicit specification of their mapping in physical memory space. If that is the case, use the ORG directive in the address_translation construct.

4.1.34  EID_PHY_MEM_ADDR_SIZE

Indicates that a physical memory fragment, which is mapped to the MMU, violates the alignment rule: org/len=0.

Error (E1043)

To resolve EID_PHY_MEM_ADDR_SIZE error
Make sure that original address / length of the physical memory = 0.

4.1.35  EID_PHY_MEM_INVALID_RESERVE_PM

Indicates that a reservation of physical memory region is invalid. For example, whole or part of the reserved area does not exist, or some shared reservation is defined, but not at the identical address for all cores.

Error (E1047)

To resolve EID_PHY_MEM_INVALID_RESERVE_PM error
Verify the reservations in the physical_memory constructs, and the corresponding specifications for the memory sharing, start address and the size definitions.

4.1.36  EID_PHY_MEM_MULTI

Indicates more than one physical memory definition in the LCF with the same name.
Error (E1046)

To resolve EID_PHY_MEM_MULTI error
Make sure that all physical memory definition in the LCF have unique names.

4.1.37  EID_PHY_MEM_OVERLAP

Indicates overlapped physical memory fragments.

Error (E1037)

To resolve EID_PHY_MEM_OVERLAP error
Modify the physical memory fragments.

4.1.38  EID_PHY_MEM_OVERLAPPED

Indicates that the physical memory description contains memory fragments that are overlapping as address ranges.

Error (E1045)

To resolve EID_PHY_MEM_OVERLAPPED error
Check the physical_memory constructs in the LCF and the corresponding specifications for the memory size and the start address.

4.1.39  EID_PHY_MEM_PRIVATE

Indicates that a physical memory private area is defined in the address range corresponding to a shared memory on the selected architecture.

[LNK,1,6999,62,/common.l3k]: Warning (W1081): LCF configuration:
Physical memory DDR1 is considered as shared physical memory.

To resolve EID_PHY_MEM_PRIVATE warning
Verify that all physical_memoryprivate (c0, c1,...) directives are defined only in the address ranges of private memories on the target architecture.
4.1.40  **EID_PHY_MEM_RESERVE_OVERLAP_RESERVE**

Indicates that a couple of physical memory reservations have overlapping ranges.

*Warning*(E1054)

**To resolve EID_PHY_MEM_RESERVE_OVERLAP_RESERVE error**

Verify the expressions for start and size of the reserved memory areas in physical memory space. In the LCF, begin with looking for the reserves in the `physical_memory` constructs.

4.1.41  **EID_PHY_MEM_UNDEF**

Indicates that the referenced physical memory fragment is not defined.

```
[LNK,2,6999,305,foo.lcf]: Error(E1041): LCF configuration: physical memory 'M2' is not defined.
```

**To resolve EID_PHY_MEM_UNDEF error**

Define the physical memory fragment specified in the error message.

4.1.42  **EID_PHY_MEM_UNDEF_ADDR**

Indicates that the address used in the virtual memory is not defined in the physical memory. This error generally occurs when the virtual memory is mapped to physical memory in map11 style.

```
[LNK,3,6999,260,newUninitResultReturn.lcf]: Fatal(F1042): LCF configuration: undefined physical memory is used: address 0xc0040000.
```

**To resolve EID_PHY_MEM_UNDEF_ADDR error**

Extend definition of the physical memory specified in the error message, or map the respective virtual memory to another physical memory fragment.
4.1.43  EID_PHY_NO_RESULT

Indicates an error with the physical memory configuration. This error can occur when one or more of the following conditions are true.

- The starting address and size of the physical memory do not meet the restrictions specified for the MMU hardware. For instance, for the b4860 target, the origin address should be multiple of size of the memory area.
- The physical memory map contains overlapping physical memory regions. One common case when this happen is when:
  - address_translation specification is specified for multiple tasks in the same language construct
  - private virtual memory areas are mapped to a shared physical memory
  - the physical start address is a fixed value that does not depend on the core_id() intrinsic function
- The physical memory device is not sufficient to hold the specified virtual memory entities.

[LNK,3,6999,-1]: Fatal(F1050): LCF configuration: no physical layout result for physical memory M3.

Tentative memory map for physical memory 'M3'(org=0xd0000000, len=0x00a00000):

- c0: |porg=0xd0000000, plen= AUTOMATIC| m3_share_data(org=0xd0000000, len= AUTOMATIC)
- c0: |porg=0xd0000000, plen= AUTOMATIC| m3_share_text_boot(org=0xd0000000, len= AUTOMATIC)
- c0: |porg=0xd0000000, plen= AUTOMATIC| Data_heap_private_mmu(org=0x30000000, len=0x00001000)
- c1: |porg=0xd0000000, plen= AUTOMATIC| Data_heap_private_mmu(org=0x30000000, len=0x00001000)
- c2: |porg=0xd0000000, plen= AUTOMATIC| Data_heap_private_mmu(org=0x30000000, len=0x00001000)
- c3: |porg=0xd0000000, plen= AUTOMATIC| Data_heap_private_mmu(org=0x30000000, len=0x00001000)
- c0: |porg=0xd0000000, plen= AUTOMATIC| private_data_boot(org=0x20000000, len=0x00008000)
- c1: |porg=0xd0000000, plen= AUTOMATIC| private_data_boot(org=0x20000000, len=0x00008000)
- c2: |porg=0xd0000000, plen= AUTOMATIC| private_data_boot(org=0x20000000, len=0x00008000)
- c3: |porg=0xd0000000, plen= AUTOMATIC| private_data_boot(org=0x20000000, len=0x00008000)

To resolve EID_PHY_NO_RESULT error
Modify the physical memory configuration and make sure that:

- The starting address and size of the physical memory meet the restrictions specified for the MMU hardware.
- The physical memory map does not contain overlapping physical memory addresses.
- The physical memory device is sufficient to hold the specified virtual memory entities.

### 4.1.44 EID_PHY_PROBLEM_OVERSIZE

Indicates that the linker requires more information to automatically configure the address_translation entries.

**Error (E1052)**

**To resolve EID_PHY_PROBLEM_OVERSIZE error**

Refer to the information specified in the .dmp file, and provide more starting addresses and size information for the address_translation entries.

### 4.1.45 EID_PHY_SIZE_OVERFLOW

Indicates that the linker was not able to find a valid physical memory layout because the content size placed in physical memory is larger than its actual storage capacity.

**Error (E1051)**

**To resolve EID_PHY_SIZE_OVERFLOW error**

Consider splitting the large data/code sections content over more physical memories. Another solution could be to specify higher size-optimization levels during compilation, or to *share* more of the data and/or code.

### 4.1.46 EID_PL_AFTER_CYCLE

Indicates a cycle of virtual memory areas. It is not possible to place virtual memory A after B, and B after A at the same time.
Fatal(F2013): there is one or more cycle of "after" in address-translation specification, part of the cycle are virtual memories data_boot_c and ddr_private_data_c_wb.

To resolve EID_PL_AFTER_CYCLE error

Break the cycle by using one of the following options:

• choosing a fixed AFTER target virtual memory
• choosing a virtual memory area for AFTER specification, which has independent placement of the memory that was part of the cycle

4.1.47 EID_PL_MULTI_MAPPING

Indicates the mapped physical memory of the virtual memory is not defined, and the automatic mapping is not enabled by the command-line interface, or not supported.

[E2010]: in core c0, task task0_c0: the mapped physical memory of virtual memory data_default_init_tbl is not defined and automatic mapping is not enabled by command line or supported.

To resolve EID_PL_MULTI_MAPPING error

Provide a physical memory map for the respective virtual memory.

4.1.48 EID_PL_MMATT

Indicates that the size of the virtual memory is impossible to be described by a single MATT, and multiple MATT for single memory is not enabled by the command-line interface.

[E2012]: in core c0, task task0_c0: the size (0x00006200) of virtual memory data_default_init_tbl is impossible to be described by single MATT and multiple MATT for single memory is not enabled by command line.

To resolve EID_PL_MMATT error

Enable the multiple MATT for single memory, or reduce the virtual memory size.

4.1.49 EID_PL_PORG
Indicates the original physical address of the virtual memory is not specified, and automatic original address is not enabled by command-line interface.

[LNK,2,6999,-1]: Error(E2011): in core c0, task task0_c0: the original physical address of virtual memory data_default_init_tbl is not specified and automatic original address is not enabled by command line.

To resolve EID_PL_PORG error
Provide the address translation construct for the respective virtual memory, and specify the origin in physical memory address space.

4.1.50 EID_PROGBIT_AFTER_NOBITS

Indicates that a bss section is placed before a data section. The bss section will be transformed into a data section that contains zeros.

NOTE
The EID_PROGBIT_AFTER_NOBITS message is a warning message.

[LNK,1,6999,127,W:/bcu3_modem_phy/prod/build_lte/linker_command/local_map_link.l3k]: Warning(W1078): LCF configuration: The file image of the section will be increased. SHT_PROGBITS section (non-BSS section) is placed after SHT_NOBITS section (BSS section).

To resolve EID_PROGBIT_AFTER_NOBITS warning
Place the bss section at the end of the output section in the LCF.
To suppress this warning message, use the -disable-warn-progbits-after-nobits command-line option.
To convert this warning message to an error message, use the -enable-error-progbits-after-nobits option.

4.1.51 EID_SCL_DIRECTIVE

Indicates that the linker found at least one directive specific to the self-contained library operation mode, but could not find the related option.

Warning (W2023)

To resolve EID_SCL_DIRECTIVE warning
Add the -self-contained-library linker option and make sure that the linker command files contain only directives specific to the self-contained library output operation mode.

4.1.52 EID_SEC_BAD_ATTR

Indicates incompatible input sections in an output section. For example,

```plaintext
sec ("rx"){
  .exception_index;
  .text;
} >mx;
```

The .exception_index is a read-only input section and the .text section is an input section with attribute "rx".

[LNK,2,6999,23,align1.cmd]: Error(E1000): LCF configuration: in core c0: section '.exception_index'(attr='r') is not compatible with other inputs(attr='rx').

To resolve EID_SEC_BAD_ATTR error

Move the incompatible input section to a compatible output section.

4.1.53 EID_SEC_MEM_ATTR

Indicates that an output section is placed into an incompatible virtual memory block. The attributes of the output section do not match with that of the virtual memory.

[LNK,3,6999,22,align2_EID_SEC_MEM_ATTR.cmd]: Fatal(F1002): LCF configuration: SECTION 'INTVEC'(attr='rx') is not compatible with virtual memory 'mem1'(attr='rw').

To resolve EID_SEC_MEM_ATTR error

Place the output section in a compatible virtual memory block. Alternatively, modify the attributes of the corresponding virtual memory block so that it is compatible with the output section.

4.1.54 EID_SEC_MEM_SIZE
Indicates that an output section is placed into an incompatible virtual memory block. The size of the virtual memory block is smaller than that of the output section.

```
[LNK,3,6999,23,align2_EID_SEC_MEM_SIZE.cmd]: Fatal(F1003): LCF configuration: SECTION 'INTVEC'(size=0x00001000) is too large to fit into remaining space of virtual memory 'mem1'.
```

Memory map for virtual memory 'mem1':
```
| org=0x00000000, len=0x00000064 | free |
```

**To resolve EID_SEC_MEM_SIZE error**

Extended the size of the corresponding virtual memory block. Alternatively, move the output section to a larger virtual memory block.

### 4.1.55 EID_SEC_MULTI_DEF

Indicates that the LNK_SECTION directive defines a linker section with a name that is already in use by other input sections placed for the same task.

```
[LNK,3,6999,148,../b4860/LCF/b4860.l3k]: Fatal(F1009): LCF configuration: in core c5, task task0_c5: redefine input section 'stack' by LNK_SECTION(...).
```

**To resolve EID_SEC_MULTI_DEF error**

Make sure that the custom name of the linker-defined section does not match the name of other input sections placed for a particular task.

### 4.1.56 EID_SEC_NO_MEM

Indicates that the output section specified in the warning message is defined in the LCF, but is not placed in a virtual memory block.

**NOTE**

The EID_SEC_NO_MEM message is a warning message.

```
[LNK,1,6999,22,align_SEC_NO_MEM.cmd]: Warning(W1005): LCF configuration: SECTION 'INTVEC' is not placed into any virtual memory.
```

**To resolve EID_SEC_NO_MEM error**

If this warning message is followed by the error message "higher memory layout automation level needed, ...", place the corresponding section in a virtual memory block in the LCF.
4.1.57  **EID_SEC_NOT_PLACED**

Indicates that a section is loaded from the input modules, but is not placed in any memory block.

```
[LNK,3,6999,-1]: Fatal(F1008): LCF configuration: SECTION '.text' is not placed into any memory.
```

**To resolve EID_SEC_NOT_PLACED error**

Place the section specified in the error message in a virtual memory block.

**NOTE**

It is recommended that you explicitly exclude the sections (using the **EXCLUDE** directive) that are not intended to be linked to a specific task. Alternatively, you can rename such sections in the LCF to enforce limited core visibility. For example, `.text_sec`, when renamed to `c1`.text_sec, limits text_sec's visibility to core 1 (c1).

4.1.58  **EID_SEC_OSEC_ATTR**

Indicates that the attributes specified for an output section are not consistent with the compiler/assembler generated input section attributes.

```
[LNK,3,6999,22,align2_EID_SEC_BAD_ATTR.cmd]: Fatal(F1001): LCF configuration: SECTION 'INTVEC'(attr='rw') is not compatible with inputs(attr='rwx').
```

**To resolve EID_SEC_OSEC_ATTR error**

Modify the attributes for the output section so that they are consistent with the input section attributes.

4.1.59  **EID_SEC_PC_BACK**

Indicates a wrong assignment to the PC value. The value assigned can only be incremented. Decrementing the value results in an error. For example,
The second assignment, . = 0x0, results in an error.

[LNK,3,6999,27,align_SEC_PC_BACK.cmd]: Fatal(F1010): LCF configuration: set location counter value backward.

To resolve EID_SEC_PC_BACK error
Do not decrement the PC value.

4.1.60  EID_SEC_SIZE_OVERFLOW

Indicates that the memory of a section has crossed the maximum limit.

Error(E1011)

To resolve EID_SEC_SIZE_OVERFLOW error
Split the corresponding section. The maximum memory limit for a section, in the current linker, is unsigned 0xffffffff.

4.1.61  EID_SEC_UNDEF_MEM

Indicates that no definition exists in the LCF for the virtual memory specified in the error message.

[LNK,3,6999,18,linker_control_files\newlcf\aa1.cmd]: Fatal(F1004): LCF configuration: in core c0, task task_1_0: SECTION 'm2_share_text' is placed into unavailable virtual memory 'm2_share-data'.

To resolve EID_SEC_UNDEF_MEM error
Verify that the virtual memory specified in the error message is defined in the LCF.

4.1.62  EID_SEC_UNMATCH_ATTR
Indicates that section components with the same name, belonging to different modules, differ in attributes.

**NOTE**

The section components with the same name, belonging to different modules, are merged into one input section.

**Error(E1007)**

**To resolve EID_SEC_UNMATCH_ATTR error**

Use RENAME directive to change the corresponding section names so that they are not merged together.

### 4.1.63 EID_SMALL_ATTMMU_SIZE

Indicates that the memory size specified in the LCF for the .att_mmu section cannot include all the required address_translation entries.

[LNK,3,6999,-1]: Fatal(F1016): LCF configuration: att_mmu size specification is too small, actual=16, expect=248.

**To resolve EID_SMALL_ATTMMU_SIZE error**

You use the `lnk_section` directive to specify the memory size:

```
lnk_section(att_mmu,"rw", 0x16, 256 );
```

Modify the size parameter in the `lnk_section` directive to specify a larger memory size.

### 4.1.64 EID_SOME_CORES_WITHOUT_TASKS

This error message occurs in case the `tasks` directive is used to define tasks for each active core, where the number of active cores is specified by the `number_of_cores` directive, and not each of these receives description of at least one hosted task.

[LNK,3,6999,-1]: Fatal(F1080): LCF configuration: There is no task specified for core c1.

**To resolve EID_SOME_CORES_WITHOUT_TASKS error**

Make sure that each active core, c0, c1, ..., cn-1, where n is used in the `number_of_cores(n)` directive, hosts at least one task.
4.1.65  **EID_START_ADDR_EXPR**

Indicates an invalid value for the start_address parameter.

```
[LNK,3,6999,36,new3.lcf]: Fatal(F1012): LCF configuration: only symbol or constant is allowed for START_ADDRESS.
```

**To resolve EID_START_ADDR_EXPR error**

Specify only symbol or constant for the `start_address` parameter.

4.1.66  **EID_START_ADDR_MULTI**

Indicates conflicting definitions for the start_address parameter among multiple cores.

```
Error(E1014)
```

**To resolve EID_START_ADDR_MULTI error**

Define the `start_address` parameter in a shared memory area.

4.1.67  **EID_START_ADDR_REDEF**

Indicates more than one definition for the start_address parameter in a single core.

```
Error(E1013)
```

**To resolve EID_START_ADDR_REDEF error**

Specify only one definition for the `start_address` parameter in a single core.

4.1.68  **EID_TASK_OVERFLOW**

Indicates that the task PID or DID was set beyond 255.

```
[LNK,2,6999,-1]: Error(E1024): LCF configuration: TASK(name=sys0, coreid=1, taskid=256) is overflow, taskid cannot be greater than 255.
```
To resolve EID_TASK_OVERFLOW error
Make sure that no PID or DID is greater than 255.

4.1.69 EID_TASK_REDEF_VM

Indicates more than one definition for a single virtual memory.

[LNK,3,6999,70,linker_control_files\newlcf\aa4.cmd]: Fatal(F1025): LCF configuration in core c0, task task_1_0: redefine virtual memory 'private_data_boot'.

  linker_control_files\newlcf\aa4.cmd(65): private_data_boot("rw"): org = _VIRTUAL_DATA_BOOT_start;

  linker_control_files\newlcf\aa4.cmd(70): private_data_boot("rw"): org = _VIRTUAL_DATA_BOOT_start+0x600;

To resolve EID_TASK_REDEF_VM error
Specify only one definition for the corresponding virtual memory.

4.1.70 EID_TASK_UNDEF

Indicates that the task specified in the error message is referenced (e.g. in the unit or the address_translation directive), but is not defined.

Error(E1020)

To resolve EID_TASK_UNDEF error
Add the definition for the task specified in the error message, or remove its reference.

4.1.71 EID_UNRESOLVE_REF

Indicates the symbol specified in the error message was not found either in the input files or in the LCF file.

[LNK,1,6999,-1]: Warning(W2003): in core c0: Symbol resolution: found undefined input symbol '_DataInR', referenced from:

./Source/msc8156_main.eln

To resolve EID_UNRESOLVE_REF error
Check source files for the missing declaration of the symbol specified in the error message, or according to the case, if the symbol is meant for the LCF statements, provide a value for it.

### 4.1.72 EID_WRONG_AT_ORG

Indicates that the origin address in the address translation construct for the virtual memory specified in the error message is an undefined symbol, or is a symbol depended of the memory layout.

```
[LNK,3,6999,-1]: Fatal(F1070): LCF configuration: org for address translation entry for virtual memory 'data_boot_c' was specified using unknown symbols or layout dependent intrinsics.
```

**To resolve EID_WRONG_AT_ORG error**

Choose an origin for the address translation construct of the specified virtual memory that is depended on defined symbols, or is using intrinsics that don't require completion of the memory layout phase.

### 4.1.73 EID_WRONG_VM_ORG

Indicates that the origin of the virtual memory specified in the error message is an undefined symbol or is a symbol depended of the memory layout.

```
[LNK,3,6999,-1]: Fatal(F1069): LCF configuration: org for virtual memory 'ddr_private_text_c' was specified using unknown symbols or layout dependent intrinsics.
```

**To resolve EID_WRONG_VM_ORG error**

Choose an origin for the specified virtual memory that is depended on defined symbols, or is using intrinsics that don't require completion of memory layout phase.

### 4.2 Parser error messages

The linker may display parser error messages while parsing the LCF.
4.2.1 **EID_BAD_CORENAME**

Indicates a syntax issue with one of the core names used in a syntax context, such as a physical_memory or tasks construct.

[LNK,3,6999,-1]: Fatal(F0016): LCF syntax: bad core name 'core1', expect 'c%d', like 'c0'.

**To resolve EID_COMMENT error**

Make sure that the cores are referred by names with the string pattern "c%d". For example, the cores for b4860 architecture are named c0,c1,c2,c3,c4,c5.

4.2.2 **EID_COMMENT**

Indicates that a multiline comment is not closed properly.

[LNK,2,6003,-1]: Error(E0002): LCF syntax: comment error - unclosed multiline comment.

**To resolve EID_COMMENT error**

Close all the multiline comments.

4.2.3 **EID_DEFINE**

Indicates an incorrect syntax for the define directive.

[LNK,2,6003,-1]: Error(E0004): LCF syntax: define - illegal symbol name '#'.

[LNK,2,6003,-1]: Error(E0004): LCF syntax: define - illegal expression '#'.

**To resolve EID_DEFINE error**

Specify correct syntax for the define directive.

4.2.4 **EID_EMPTY_EXP**

Indicates that an #if construct does not have an expression to evaluate.

[LNK,3,6003,14,8156_lcf.cmd]: Fatal(F0012): LCF syntax: empty #if expression.
To resolve EID_EMPTY_EXP error
Specify an expression for the #if construct.

4.2.5 EID_INCLUDE

Indicates an issue with linker command file inclusion; be it inclusion recursion, high nesting or a file reading error.

[LNK,3,6003,2,../b4860/LCF/b4860.l3k]: Fatal(F0003): LCF syntax: include error - includes nested too deep.

[LNK,3,6003,4,../b4860/LCF/b4860.l3k]: Error(F0003): LCF syntax: include error - unclosed quote. Correct syntax: #include "file_name".

To resolve EID_INCLUDE error
Verify the linker command file inclusion structure, file names, and paths.

4.2.6 EID_INCOMPLETE_EXP

Indicates that an #if construct has an incomplete expression.

[LNK,3,6003,13,8156_lcf.cmd]: Fatal(F0009): LCF syntax: incomplete #if expression.

To resolve EID_INCOMPLETE_EXP error
Specify complete expression for the #if construct.

4.2.7 EID_MISSING_PAREN_EXP

Indicates that the parenthesis in an #if construct are not properly closed.

[LNK,3,6003,13,8156_lcf.cmd]: Fatal(F0010): LCF syntax: missing paren in #if expression.

To resolve EID_MISSING_PAREN_EXP error
Make sure that the number of opening parenthesis equals the number of closing parenthesis.
4.2.8 EID_MATT_WX

Indicates that one entry in MATT received both WRITE and EXECUTE accessibility attributes, which is not valid given the architectural separation of the data and program spaces.

Error(E1055)

To resolve EID_MATT_WX error

Verify the address translation entries and verify that there is no forcing of both the WRITE and EXEC attributes on the same virtual memory entry. If the intention is to create RWX areas, see How to Define Virtual Memory for Read-Write-Execute (RWX) Access.

4.2.9 EID_OLD_LCF_FORMAT

Indicates a syntax issue that is likely to occur if you use an older unsupported linker (sc100-ld) lcf file as an input to the sc3000-ld linker.

[LNK,2,6003,5,/b4860/LCF/b4860.cmd]: Error(E0006): LCF syntax: directive '.provide' found. You may be using the old lcf format.

To resolve EID_OLD_LCF_FORMAT error

Make sure that you use sc3000-ld directives and syntax.

4.2.10 EID_PREPROCESS

Indicates an issue with the preprocessing directives.

[LNK,2,6003,6,/b4860/LCF/b4860.l3k]: Error(E0005): LCF syntax: too many ENDEFs.

[LNK,3,6003,7,/b4860/LCF/b4860.l3k]: Fatal(F0005): LCF syntax: illegal symbol name.

[LNK,3,6003,10,/b4860/LCF/b4860.l3k]: Fatal(F0005): LCF syntax: missing #if, #ifdef or ifndef directive.

[LNK,3,6003,5,/b4860/LCF/b4860.l3k]: Fatal(F0005): LCF syntax: IFDEF expression must end with newline.

To resolve EID_PREPROCESS error
Use the information given in the message description to correct the preprocessing directive description/definition.

4.2.11  EID_UNDEF_OPER_IN_EXP

Indicates a syntax issue with one of the operators used in the #if expression.

[LNK,3,6003,8,b4860.l3k]: Fatal(F0014): LCF syntax: undefined operator '=' in #if expression.

To resolve EID_UNDEF_OPER_IN_EXP error
Verify the #if expression in the specified linker command file, and make sure that all the operators are amongst the supported ones for the #if expression, such as <=, >=, >, <, !=, ==, ||, &&.

4.2.12  EID_UNEXPECTED_TOKEN

Indicates that a token is found at an invalid position.

[LNK,2,6003,-1]: Error(E0001): LCF syntax: unexpected token in top_level<>.

To resolve EID_UNEXPECTED_TOKEN error
Specify correct syntax for the LCF directives.

4.2.13  EID_UNKNOWN_INTRINSIC

Indicates a syntax issue with one of the intrinsics used in an expression.

[LNK,2,6003,6,../b4860/LCF/b4860.l3k]: Error(E0015): LCF syntax: unknown intrinsic '_originof'.

To resolve EID_UNKNOWN_INTRINSIC error
Verify the expression in the specified linker command file, and make sure all the intrinsic names are amongst the supported ones. The supported intrinsics are documented in LCF Expression Functions.
4.2.14  **EID_UNKNOWN_PERM_FLAG**

Indicates an invalid flag.

```
[LNK,2,6003,25,b4860_lcf.cmd]: Error(E0007): LCF syntax: unknown permission flag 'Z'.
```

**To resolve EID_UNKNOWN_PERM_FLAG error**

Remove the invalid flag specified in the error message.

4.2.15  **EID_UNSUPPORTED_ATTR**

Indicates an invalid attribute for a section. The supported attribute is only '+z'.

```
[LNK,2,6003,39,b4860_lcf.cmd]: Error(E0008): LCF syntax: unsupported attribute '+Y'.
```

**To resolve EID_UNSUPPORTED_ATTR error**

Replace the invalid attribute specified in the error message with '+z', or remove the invalid attribute.

4.3  **Setup error messages**

The linker may display setup error messages if the linker is not setup properly.

4.3.1  **EID_ARCH_NOT_SPECIFIED**

Indicates that the target architecture is not specified.

```
[LNK,3,6999,-1]: Error(E1068): LCF configuration: the architecture was not specified.
```

**To resolve EID_ARCH_NOT_SPECIFIED error**

Use the arch directive to specify the target architecture.
4.3.2 EID_EXPR_CANNOT_EVAL

Indicates that the linker cannot evaluate the specified expression.

Error(E1061)

To resolve EID_EXPR_CANNOT_EVAL error
Make sure that all the symbols in the expression are defined.

4.3.3 EID_LCF_INCOMPLETE

This message occurs when the linker detects that insufficient information is passed through the command files.

[LNK,2,6999,-1]: Error(E2020): linker command file is not complete: no virtual memory specified.

To remove EID_LCF_INCOMPLETE message
Add the descriptions for the specified missing entities, such as sections, virtual memories, physical memories, as specified in the message description.

4.3.4 EID_LCF_INCORRECT

Indicates a syntax or a setup phase error in the LCF.

[LNK,3,6999,-1]: Fatal(F2021): linker command file is not correct, linker stops.

To resolve EID_LCF_INCORRECT error
Make sure that all syntax and setup phase errors are resolved. Check other error messages that accompany this error message for more information on syntax and setup phase errors.

4.3.5 EID_NUM_CORES_GT_ARCH

Indicates that the number of cores specified is greater than the number of cores target architecture supports.
[LNK,3,6999,-1]: Error(E1064): LCF configuration: 7 cores were specified; the architecture only supports 6.

To resolve EID_NUM_CORES_GT_ARCH error
Make sure that the number of cores specified is less than or equal to the number of cores target architecture supports.

4.3.6  EID_NUM_CORES_LT_ONE

Indicates that the number of cores specified is less than one.

[LNK,3,6999,3,D:/Profiles/b12260/workspace/test/LCF/new_lcf/8156_lcf.cmd]: Error (E1065): LCF configuration: 0 cores were specified; the number of cores cannot be less than 1.

To resolve EID_NUM_CORES_LT_ONE error
Make sure that you specify at least one core in the LCF.

4.3.7  EID_NUM_CORES_NAN

Indicates that an expression is used to define the number of cores.

[LNK,3,6999,3,8156_lcf.cmd]: Error (E1066): LCF configuration: number_of_cores must be a number.

To resolve EID_NUM_CORES_NAN error
Use only a positive integer to specify the number of cores.

4.3.8  EID_REDEF_LCF_SYM

Indicates that a user symbol is defined multiple times in the LCF file.

[LNK,2,6999,65,../b4860/LCF/common.l3k]: Error(E1075): LCF configuration: In core c3: LCF Redefinition of symbol 'OneLcfSymbol' found. First definition found in ../b4860/LCF/common.l3k line 64.

To resolve EID_REDEF_LCF_SYM error
Keep only one of the user symbol definitions, and delete or temporarily comment out the other ones.

4.3.9 **EID_REDEF_MM_SYM**

Indicates that one of the linker predefined symbols, based on architecture selection was redefined in the LCF file.

```
[LNK,1,6999,62,../b4860/LCF/common.l3k]: Warning(W1076): LCF configuration: In core c3: Redefinition of linker predefined symbol '_M3_Setting' found. User's definition will be used.
```

**To resolve EID_REDEF_MM_SYM warning**

Use one of the following options:

- Use the symbol value as it is provided by default by the machine model for the selected architecture.
- Disable the usage of predefined definitions from the machine model, using the linker option `-disable-emit-machine-model-lcf`. Using this option means that the set of memory definitions and symbols are fully in user's control.
- Continue with the machine model predefinitions, the symbol definition as set by the user in the LCF, and making use of the linker option `-disable-warn-redef-linker-sym` that just inhibits this type of warning message.

4.3.10 **EID_REPEATED_SECTION_DIFF_OS**

Indicates that one input section was placed more than one times in a couple of output sections.

```
[LNK,2,6999,-1]: Error(E1071): LCF configuration: section '.intvec' was placed more than once on task 'task0_c5' in output section 'descriptor__m3__cacheable__sys__shared__text_boot'.
```

**To resolve EID_REPEATED_SECTION_DIFF_OS error**

Decide which of the occurrences of the input section should remain in which output section, and then delete the other occurrence.

4.3.11 **EID_REPEATED_SECTIONSAME_OS**

Setup error messages
Indicates that one input section was placed more than one times in the same output section.

[LNK, 2, 6999, -1]: Error (E1071): LCF configuration: section '.intvec' was placed more than once on task 'task0_c5' in output section 'descriptor_m3_cacheable_sys_shared_text_boot'.

To resolve EID_REPEATED_SECTIONSAME_OS error

Decide which of the occurrences of the input section should remain in the final layout of the output section, and then delete the other occurrence.

4.3.12 EID_TASK_REDEF_VM

Indicates that the specified virtual memory is already defined.

To resolve EID_TASK_REDEF_VM error

Remove the already existing definition.

4.3.13 EID_TASKS_NOT_SPECIFIED

Indicates that the set of LCF does not provide a tasks directive that defines all system tasks each associated with its hosting core. In such cases, the linker implicitly assumes one task per each active core.

[LNK, 0, 6999, -1]: Information (I2019): tasks not specified; default tasks will be generated.

To remove EID_TASKS_NOT_SPECIFIED message

Define the system tasks by explicitly using the tasks directive in one of the linker command files.

4.3.14 EID_UNSUPPORTED_ARCH

Indicates that the linker does not support the specified architecture.

[LNK, 3, 6999, 2, lcf.cmd]: Error (E1067): LCF configuration: architecture 'msc8156' is not supported.

To resolve EID_UNSUPPORTED_ARCH error
Specify a supported architecture.
Chapter 5
LCF Expression Functions

This chapter describes the intrinsic functions that you use to build expressions in the LCF.

In this appendix:

- Context-dependent intrinsic functions
- Context-independent intrinsic functions

5.1 Context-dependent intrinsic functions

This chapter describes the context-dependent intrinsic functions.

A context-dependent intrinsic function has a specific scope. For example, a context-dependent intrinsic function is valid only in a specific section.

Table 5-1 lists the context-dependent intrinsic functions that the StarCore linker supports.

<table>
<thead>
<tr>
<th>Scope</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>SECTIONS</td>
<td>align</td>
</tr>
<tr>
<td>UNIT</td>
<td>endof</td>
</tr>
<tr>
<td>UNIT, ADDRESS_TRANSLATION</td>
<td>originof</td>
</tr>
<tr>
<td></td>
<td>core_id</td>
</tr>
<tr>
<td></td>
<td>sizeof</td>
</tr>
<tr>
<td></td>
<td>task_id</td>
</tr>
<tr>
<td></td>
<td>to_physical</td>
</tr>
<tr>
<td></td>
<td>vmorg</td>
</tr>
</tbody>
</table>
5.1.1  .

Represents a location counter. The LCF considers every occurrence of location counter as a linker-defined symbol.

5.1.2  align

Aligns the location counter to the value that the align-value parameter specifies.

.align(align-value)

Parameter

align-value

Alignment value (without quotes).

5.1.3  endof

Returns the virtual address where the specified output section ends.

deof("section-name")

Parameter

"section-name"

The name of the output section (with quotes).

5.1.4  originof

Returns the original virtual address of the specified output section.

.originof("section-name")

Parameter

"section-name"
5.1.5  **core_id**

Returns the current core ID.

```c
core_id()
```

5.1.6  **sizeof**

Returns the size of the specified output section.

```c
sizeof("section-name")
```

**Parameter**

"section-name"

The name of the output section (with quotes).

5.1.7  **task_id**

Returns the current task ID.

```c
task_id()
```

5.1.8  **to_physical**

Returns the physical mapping to the specified virtual address.

```c
to_physical(virtual-address)
```

**Parameter**

`virtual-address`

The virtual address (without quotes).
5.1.9  vmorg

Returns the virtual address of the specified virtual memory space.

vmorg("vm-name")

Parameter

"vm-name"

The virtual memory space (with quotes).

5.2  Context-independent intrinsic functions

A context-independent intrinsic function does not have a specific scope. For example, a context-independent intrinsic function can be used anywhere in the LCF.

Table 5-2 lists the context-independent intrinsic functions that the StarCore linker supports.

<table>
<thead>
<tr>
<th>Scope</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entire LCF</td>
<td>num_task</td>
</tr>
<tr>
<td></td>
<td>physical_address</td>
</tr>
<tr>
<td></td>
<td>num_core</td>
</tr>
<tr>
<td></td>
<td>defined</td>
</tr>
<tr>
<td></td>
<td>test_arch</td>
</tr>
</tbody>
</table>

5.2.1  num_task

Returns the total number of tasks.

num_task()
5.2.2 physical_address

Returns the physical address of the specified symbol.

    physical_address("symbol-name")

**Parameter**

"symbol-name"

The symbol name (with quotes).

5.2.3 num_core

Returns the total number of cores.

    num_core()

5.2.4 defined

Instructs the linker to use definition of the symbol from the input file if the same symbol is defined in the LCF as well.

    defined("symbol-name")

**Parameter**

"symbol-name"

The symbol name (with quotes).

5.2.5 test_arch

Returns 1 if the "arch-name" is the target architecture; returns 0 otherwise.

    test_arch("arch-name")

**Parameter**

"arch-name"
Context-independent intrinsic functions

The architecture name (with quotes).
Chapter 6
LCF Expression Operators

This chapter describes the operators that you use to build expressions in the LCF.

In this appendix:

- LCF expression operators

6.1 LCF expression operators

The StarCore linker supports standard C language arithmetic, unary, binary, relational, bitwise, and logical operators.

Table 6-1 lists all the supported operators in the descending order of priority.

NOTE
All supported operators are left-associative.

Table 6-1. LCF expression operators (listed in order of the highest to the lowest priority)

<table>
<thead>
<tr>
<th>Type</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unary</td>
<td>()</td>
</tr>
<tr>
<td></td>
<td>-(negation) -(bit negation) ! (logical negation)</td>
</tr>
<tr>
<td>Binary</td>
<td>* / %</td>
</tr>
<tr>
<td></td>
<td>+ -</td>
</tr>
<tr>
<td></td>
<td>&gt;&gt; &lt;&lt;</td>
</tr>
<tr>
<td></td>
<td>&amp;</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>== != &gt;= &gt; &gt;= &lt; &lt;=</td>
</tr>
<tr>
<td></td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Assignment</td>
<td>= += -= *= /=</td>
</tr>
<tr>
<td></td>
<td>?:</td>
</tr>
</tbody>
</table>
The following assignment operators can only be used with the location counter:

- `+=`
- `-=`
- `*=`
- `/=`
Chapter 7
LCF Preprocessing

This chapter describes the preprocessing directives available with the StarCore linker.

In this appendix:

• LCF preprocessing

7.1 LCF preprocessing

This chapter lists the LCF preprocessing directives.

The linker supports the following preprocessing directives:

• Comments
• The include directive
• The define directive
• Conditional directives

7.1.1 Comments

Comments in the LCF can be:

• single-line, indicated by the // characters

  The LCF parser ignores all tokens/statements following the // characters, until the newline character is encountered

• multi-line, indicated by the /*...*/ characters

  The LCF parser ignores all tokens/statements between the /* and the */ characters.
NOTE
The multi-line comments cannot span across multiple files, and must start and end in the same file.

7.1.2 The include directive

Use the include directive to split the LCF in multiple files. The include directive lets you manage a large LCF by splitting it into manageable small files.

Include the multiple files into one file by using this syntax:

```
#include "file_name"
```

NOTE
The linker does not support recursive inclusion.

NOTE
The included files can be nested, but not more than ten levels.

7.1.3 The define directive

Use the define directive to declare preprocessing identifiers. The preprocessing identifiers are used with conditional directives, #if and #ifdef.

Declare a preprocessing identifier by using this syntax, where idtf indicates name of the preprocessing identifier:

```
#define idtf [value]
```

The value parameter is optional. However, when specified, the value parameter can only be an alphanumeric string or an integer.

7.1.4 Conditional directives

The conditional directives instruct the LCF parser to parse the LCF based on the conditions you specify.

The listing below shows an example of how you use the conditional directives.
Listing: Using Conditional Directives

#if expression
//if the expression evaluates to true, the LCF parser parses this code
#endif

#ifdef macro
//if the macro is defined using the #define directive, the LCF parser parses this code
#endif

#ifndef macro
//if the macro is not defined using the #define directive, the LCF parser parses this code
#endif

#else
//if the expression evaluates to false, the LCF parses this code
#endif
Chapter 8
Linker Predefinitions

The SC3000 linker supports a number of predefined symbols and predefined physical memory regions.

These predefined symbols and memory regions enhance the linker usability.

In this appendix:

- Predefined Symbols for MMU Descriptors
- Predefined Physical Memory Regions

8.1 Predefined Symbols for MMU Descriptors

The linker supports predefined symbols that represent the attributes of the MMU descriptors.

The predefined symbols for MMU descriptors are divided in two categories:

- Predefined symbols for MMU program descriptors
- Predefined symbols for MMU data descriptors

These categories can further be divided into two sub-categories:

- Predefined symbols for cache policies (cacheable/non-cacheable, write through, write back, etc), and burst size
- Predefined symbols for write access (Read/Write permission for user or super user)

The listing below shows predefined symbols for MMU program descriptors.

**Listing: Predefined Symbols for MMU Program Descriptors**

```c
//used to set M_PSDAx "Program Segment Descriptor Registers A (M_PSDAx)"
MMU_PROG_PREFETCH_MISS = 0x00000080 ; // PFP[8,7]
```
Predefined Symbols for MMU Descriptors

MMU_PROG_PREFETCH_ANY = 0x00000100 ; // PFP[8,7]
MMU_PROG_DEF_CACHEABLE = 0x00000020 ; // IC[5]
MMU_PROG_DEF_XPERM = 0x00000004 ; // PAPS[2]

//used to set M_PSDCx "Program Segment Descriptor Registers C (M_PSDCx)"
MMU_PROG_COHERENCY_MODE = 0x00010000 ; // PCM[16]

The listing below shows predefined symbols for MMU data descriptors.

Listing: Predefined Symbols for MMU Data Descriptors

// Data descriptors
//used to set M_DSDAx "Data Segment Descriptor Registers A (M_DSDAx)"
MMU_DATA_PREFETCH_MISS = 0x00000080 ; // PFP[8,7]
MMU_DATA_PREFETCH_ANY = 0x00000100 ; // PFP[8,7]
MMU_DATA_WRITE_THROUGH = 0x00000040 ; // DWP[6 =7]
MMU_DATA_CACHEABLE = 0x00000020 ; // SSVDM[5]
MMU_DATA_DEF_RPERM = 0x00000004 ; // DAPS[2 =1]
MMU_DATA_DEF_WPERM = 0x00000002 ; // DAPS[2 =1]

//used to set M_DSDCx "Data Segment Descriptor Registers C (M_DSDCx)"
MMU_DATA_DEF_BANK0_ACCESS = 0x00800000 ; // B0[31]
MMU_DATA_DEF_STACK = 0x00080000 ; // SD[19]
MMU_DATA_DEF_GUARDED = 0x00040000 ; // DG[18]
MMU_DATA_PERIPHERAL_SPACE = 0x00020000 ; // DP[17]
MMU_DATA_COHERENCY_MODE = 0x00010000 ; // PCM[16]

You can use the predefined symbols listed in Listing: Predefined Symbols for MMU Program Descriptors and Listing: Predefined Symbols for MMU Data Descriptors to create a set of attributes to set the attribute field in address translation entries. The listing below shows an example.

Listing: Creating and Using Set of Attributes

SYSTEM_DATA_MMU_DEF_REGA = MMU_DATA_CACHEABLE|
                            MMU_DATA_PREFETCH_ANY|
                            MMU_DATA_DEF_WPERM|
                            MMU_DATA_DEF_RPERM;

SYSTEM_DATA_MMU_DEF_C  = MMU_DATA_COHERENCY_MODE;

address_translation (*) {
    data_boot_c (SYSTEM_DATA_MMU_DEF_REGA, SYSTEM_DATA_MMU_DEF_REGC): DDR, org = _PRIVATE_DATA_BOOT_start;
}
Since the predefined symbols represent bit values, you can overwrite the symbols by redefining them in the LCF.

**NOTE**
Overwriting the predefined symbols is not recommended.

When you overwrite a predefined symbol, the linker generates a warning message:

```
Redefinition of linker predefined symbol 'S' found. User's definition will be used.
```

### 8.2 Predefined Physical Memory Regions

The linker supports the M3 and DDR predefined physical memory regions.

The listing below shows an example.

**Listing: Predefined Symbols for Physical Memory Regions**

```c
physical_memory shared {*}{
    M3: org = _M3_start, len = _M3_size;
    DDR: org = _DDR_start, len = _DDR_size;
}
}
```

As the listing above shows, the linker also supports the symbols for memory size and length. The listing below shows another example.

**Listing: Predefined Symbols for Memory Size and Length**

```c
_M3_start = 0x30000000;
_M3_size = (_M3_Setting == 0x0f) ? 0x80000 :
    (_M3_Setting == 0xff) ? 0x100000 :
    0x0;// M3 size
_M3_end = _M3_start + _M3_size - 1;
-DDR_start = 0x40000000;
-DDR_size = 0x40000000; // DDR size (1024M)
-DDR_end = _DDR_start + _DDR_size -1;
_M3_Setting= 0x0;
_M3_size = (_M3_Setting == 0x0f) ? 0x80000 :
    (_M3_Setting == 0xff) ? 0x100000 :
    0x0;// M3 size
```

By default, the \_M3\_Setting symbol configures the M3 as L3 cache.

The listing below shows an example of how the predefined physical memory regions are used in an address translation construct.

**Listing: Using Predefined Physical Memory Region**

```c
address_translation (*){
    m3_private_data_c_wb (SYSTEM_DATA_MMU_DEF_REGA, SYSTEM_DATA_MMU_DEF_REGC): M3, org = \_M3\_PRIVATE_start;

    ddr_private_data_c_wb (SYSTEM_DATA_MMU_DEF_REGA, SYSTEM_DATA_MMU_DEF_REGC): DDR, org = \_DDR\_PRIVATE_start;
}
```

You can disable the predefined symbols by using the \-disable-emit-machine-model-lcf\ command-line option.
Chapter 9
Command-Line Options

This chapter describes the command-line options.

Table 9-1 lists the command-line options that sc3000-ld linker supports.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>disable-allow-multiple-definition</td>
<td>Disables the use of multiple definitions for a symbol. By default, the sc3000-ld linker allows multiple definitions. To enable again if disabled, use -allow-multiple-definition.</td>
</tr>
<tr>
<td>-o2-place</td>
<td>Optimizes the intra-section space in case you place each variable in its own input section using the -Xllt --one_symb_per_sect option</td>
</tr>
<tr>
<td>-bsstable-file &lt;output_file.txt&gt;</td>
<td>Lets you skip emitting SREC records for the .bss type sections, which are not placed at the end of the segment. When a .bss type section is not placed at the end of the segment, the section is converted to a data type section during the linking process. Therefore, no address and size information for such sections exist in the __bss_table. When you use the -bsstable-file &lt;output_file.txt&gt; command to be able to skip emitting SREC records for such sections, the linker generates a new output file that contains the required information. The new output file is a text file that contains a table with the two columns: • physical_address; represented by a 32-bit hexadecimal unsigned integer • size; in bytes; represented by a 32-bit hexadecimal unsigned integer</td>
</tr>
<tr>
<td>-ignore-machine-model-specification</td>
<td>Ignores machine model specifications, and makes it mandatory for you to specify all information in the LCF using:</td>
</tr>
</tbody>
</table>

Table continues on the next page...
### Table 9-1. Command-line options (continued)

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• physical memory definitions, including symbols that define the start, the end, and the sizes of the physical memories&lt;br&gt;• symbols that define the MMU attributes</td>
</tr>
<tr>
<td></td>
<td>By default, this option is disabled because the default behavior of the linker is to use internal machine model information to validate physical memory definitions. You can use this option to link applications for new architectures that are not supported by StarCore linker. However, the unsupported architectures must have following characteristics similar to the original architecture:&lt;br&gt;• visibility of physical memory space&lt;br&gt;• usage of MMU, include maximum number of MMU descriptors&lt;br&gt;• supported segment models&lt;br&gt;• maximum number of cores in case of multi-core architecture</td>
</tr>
<tr>
<td></td>
<td>Note that the only accepted difference between unsupported and supported architectures is the size of the physical memory space, including its start and end addresses.</td>
</tr>
<tr>
<td>-L</td>
<td>Enables the user to provide linker search paths. These paths are used for searching the libraries provided with <code>-l</code> command line option and the LCF include files.</td>
</tr>
<tr>
<td>-warnings-as-errors</td>
<td>Forces the linker to treat all warnings as errors.</td>
</tr>
</tbody>
</table>
Chapter 10
Sections in LCF

This chapter describes the sections in LCF and lists the sections generated by the CodeWarrior linker and compiler.

A section is a relocatable block of code or data that is encapsulated by the SECTION and ENDSEC assembler directives and has an associated section name and type. Although you can create any name for a section, some section names are reserved by the debugger and the SmartDSP Operating System. The application must not use these reserved names (see the assembler user's guide and the corresponding SmartDSP OS documentation).

In addition, the assembler recognizes conventional ELF sections such as .text, .data, .rodata, and .BSS.

A section can be shared by multiple tasks. For more information, refer to the topic Understanding linker terminology.

The following rules apply when you access the symbols defined in a shared or private section:

- symbol defined in a private section can be accessed from:
  - other private sections of the same task
  - a shared section, only if the accessed symbol is defined at the same virtual address in all the tasks. In this case, the descriptors of all the cores must have the same starting virtual address.

- symbol defined in a shared section S can be accessed from:
  - any private section of the task sharing list of section S
  - any shared section, whose task sharing list is included in the task sharing list of section S

There are two types of sections:

- Core specific section:
The section is prefixed by the name of core of group of cores and is visible only in the cores part of the group (For example, c0`.private_data, c0`.text, c0`.private_text sections are visible only for the "c0" core; c0`c1`.data is visible only for cores "c0" and "c1").

- Non-core specific section

The section that is not prefixed by the name of the core (For example, .data, .data_private, .text, .private_text sections) is visible for all cores. These sections can be placed in the private or shared space.

The following table lists the sections generated by the CodeWarrior linker and compiler.

### Table 10-1. Sections in LCF

<table>
<thead>
<tr>
<th>Section name</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>.att_mmu</td>
<td>Data section that is used in startup file/ runtime library and system operation to set the MMU registers. File att_mmu.h defines the data structures and variables needed for the .att_mmu section. Placed in a private data descriptor.</td>
</tr>
<tr>
<td>.bss</td>
<td>Un-initialized data section that is placed in a private data descriptor</td>
</tr>
<tr>
<td>.bsstab</td>
<td>Read-only data section that is used in the startup file to fill the .bss sections with zeros. File bsstab.h defines the data structures and variables needed for the .bsstab section. Placed in a private data descriptor.</td>
</tr>
<tr>
<td>&lt;space&gt; _bss_sections_table</td>
<td>Read-only data section that is used in the startup file to fill the .bss sections with zeros. Defined only for architectures with MMU support. They are referred from the .bsstab sections. &lt;space&gt; is composed out of the cores names that share the space (e.g. for a section shared between c0 and c1: .c0c1_bss_sections_table) Must be placed in a descriptor shared between the cores that define &lt;space&gt; (e.g section .c0c1_bss_sections_table must be placed in shared section between cores c0 and c1).</td>
</tr>
<tr>
<td>.data</td>
<td>Data section that is placed in a private data descriptor.</td>
</tr>
<tr>
<td>.default</td>
<td>Program section that is created by assembler for code that is not put between &quot;section &lt;name&gt;&quot; and &quot;endsec&quot; directives. In Single Instruction Multi Data (SIMD) application model, needs to be placed in a shared program descriptor. In Multi Instruction Multi Data (MIMD) application model, needs to be placed in a private program descriptor.</td>
</tr>
<tr>
<td>.exception</td>
<td>Read-data section that is used in the startup file/runtime library to catch the C++ Exception (Exception table). Placed in a private data descriptor.</td>
</tr>
<tr>
<td>.exception_index</td>
<td>Read-only data section that is used to initialize the global variable ROM to RAM (-mrom option from scc). File init_table.h defines the data structures and variables needed for the .init_table section. Placed in a private data descriptor.</td>
</tr>
</tbody>
</table>

*Table continues on the next page...*
<table>
<thead>
<tr>
<th>Section name</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>.intvec</td>
<td>Program section that is used to define the interrupt vector code. Recommended to be placed in a shared program descriptor, if placed in a private program descriptor, need to set the VBA register again, as the support from runtime library assumes that the virtual and physical address for VBA share the same value.</td>
</tr>
<tr>
<td>os_ *</td>
<td>These sections are the system operation sections. These sections can be for code, data, read-only data or bss.</td>
</tr>
<tr>
<td>reserved_crt_tls</td>
<td>Data section that is used in reentrant runtime library. The context local data variable is defined in this section. Placed in a private data descriptor.</td>
</tr>
<tr>
<td>reserved_crt_mutex</td>
<td>Data section that is used in reentrant runtime library. The MUTEX variables are defined in this section. These variables are used by the critical region. This section needs to be mentioned in a non-cacheable descriptor from MMU among all cores. Placed in a shared data descriptor.</td>
</tr>
<tr>
<td>.rom</td>
<td>Un-initialized data section that is placed in a private data descriptor.</td>
</tr>
<tr>
<td>.rom_init_tables</td>
<td>Read-only data section that is used to initialize the global variable from ROM to RAM. File init_table.h defines the data structures and variables needed for the .rom_init_tables section. Placed in a private data descriptor.</td>
</tr>
<tr>
<td>.staticinit</td>
<td>Read-only data section that is used in the startup file/runtime library to initialize the C++ static objects. Placed in a private data descriptor.</td>
</tr>
<tr>
<td>.text</td>
<td>Program section that is placed in a shared program descriptor</td>
</tr>
<tr>
<td>.zdata</td>
<td>Data section that is fitted in the first 64k of memory. Placed in a private data descriptor.</td>
</tr>
</tbody>
</table>
Index

A
About this Document 11
Accompanying Documentation 12
align 124

C
Changes Made to Support Flexible Startup Configuration 82
Command-Line Options 139
Comments 131
Concepts 69
Conditional Directives 132
Configuration Error Messages
Configuring the Physical Memory 77
Configuring the Virtual Memory 74
Constraints With Flexible Startup Configuration 17
Context-dependent Intrinsic Functions 123
Context-independent Intrinsic Functions 126
core_id 125
Creating an Output Section 75

D
defined 127
Defining Tasks 72
Defining Virtual Memory and Output Sections 73

E
EID_ARCH_INCOMPATIBLE_WITH_DIRECTIVE 87
EID_ARCH_NOT_SPECIFIED 117
EID_ASSERT_FAIL 93
EID_ATTMMU_SIZE_UNSPECIFIED 93
EID_BAD_CORENAME 113
EID_COMMENT 113
EID_DEFINE 113
EID_EMPTY_EXP 113
EID_EXPR_CANNOT_EVAL 118
EID_EXPR_SECRENAME_CANNOT_EVAL 88
EID_FAIL_LAYOUT 88
EID_FAIL_VIRTUAL_LAYOUT 88
EID_FORCE_VALUE_TO_1 89
EID_INCLUDE 114
EID_INCOMPLETE_EXP 114
EID_INCONSISTENT_SYMADDR 89
EID_LAYOUT_UNSPECIFIED 89
EID_LCF_INCOMPLETE 118
EID_LCF_INCORRECT 118
EID_LNK_SECTION_TYPE_UNKNOWN 90
EID_MATT_MAP11_ORG 94
EID_MATT_SPEC 94
EID_MATT_V1ToPn 94
EID_MATT_VP_UNMATCH 95
EID_MATT_WX 95, 115
EID_MATTTS_OVERNUMBER 95
EID_MEM_ADDR_SIZE_UNALIGN 95
EID_MEM_CANNOT_FINAL 90
EID_MEM_EMPTY 96
EID_MEM_INCONSISTENT 91
EID_MEM_MULTI_AT 91
EID_MEM_NOT_FULLY_SPEC_RESERVE 91
EID_MEM_OVERLAP 96
EID_MEM_PLACEInto_RESERVE 96
EID_MEM_REAL_OVERLAP 97
EID_MEM_REAL_OVERLAP_1 97
EID_MEM_SMALL_MB 92
EID_MEM_UNDEF 92
EID_MEM_UNDEF_ADDR 92
EID_MEM_VIR_NO_PHY 93
EID_MISSING_PAREN_EXP 114
EID_MORE_AUTO_LAYOUT 90
EID_MULTI_ATTMMU_SIZE 97
EID_NUM_CORES_GT_ARCH 118
EID_NUM_CORES_LT_ONE 119
EID_NUM_CORES_NAN 119
EID_OLD_LCF_FORMAT 115
EID_PHY_CANNOT_LAYOUT 97
EID_PHY_MEM_ADDR_SIZE 98
EID_PHY_MEM_INVALID_RESERVE_PM 98
EID_PHY_MEM_MULTI 98
EID_PHY_MEM_OVERLAP 99
EID_PHY_MEM_OVERLAPPED 99
EID_PHY_MEM_PRIVATE 99
EID_PHY_MEM_RESERVE_OVERLAP_RESERVE 100
EID_PHY_MEM_UNDEF 100
EID_PHY_MEM_UNDEF_ADDR 100
EID_PHY_NO_RESULT 101
EID_PHY_PROBLEM_OVERSIZE 102
EID_PHY_SIZE_OVERFLOW 102
EID_PL_AFTER_CYCLE 102
EID_PL_MMATT 103
EID_PL_MULTI_MAPPING 103
EID_PL_PORG 103
EID_PREPROCESS 115
EID_PROGBIT_AFTER_NOBITS 104
EID_REDEF_LCF_SYM 119
EID_REDEF_MM_SYM 120
EID_REPEATED_SECTION_DIFF_OS 120
EID_REPEATED_SECTIONSAME_OS 120
EID_SCL_DIRECTIVE 104
EID_SEC_BAD_ATTR 105
EID_SEC_MEM_ATTR 105
EID_SEC_MEM_SIZE 105
EID_SEC_MULTI_DEF 106
EID_SEC_NO_MEM 106
EID_SEC_NOT_PLACED 107
EID_SEC_OSEC_ATTR 107
EID_SEC_PC_BACK 107
EID_SEC_SIZE_OVERFLOW 108
EID_SEC_UNDEF_MEM 108
EID_SEC_UNMATCH_ATTR 108
EID_SMALL_ATTMMU_SIZE 109
EID_SOME_CORES WITHOUT_TASKS 109
EID_START_ADDR_EXPR 110
EID_START_ADDR_MULTI 110
EID_TASK_OVERFLOW 110
EID_TASK_REDEF_VM 111, 121
EID_UNDEFINED 111
EID_TASKS_NOT_SPECIFIED 121
EID_UNDEF_OPER_IN_EXP 116
EID_UNEXPECTED_TOKEN 116
EID_UNDEFINED_INTRINSIC 116
EID_UNDEFINED_PERM_FLAG 117
EID.UNRESOLVE_REF 111
EID_UNSUPPORTED_ARCH 121
EID_UNSUPPORTED_ATTR 117
EID_WRONG_AT_ORG 112
EID_WRONG_VM_ORG 112
endof 124
Example for Multi-Core Architectures 24

H
How to Build Expressions in the LCF 43
How to check local symbols addresses 65
How to Create a Linker Command File (LCF) 14
How to Create Virtual Memory for Private Sections 35
How to Define and use a Custom set of Tasks 18
How to Define Physical Memory Address Space of Target Architecture 22
How to Define Physical Memory Layout for a Multi-core Application 26
How to Define Private Data Sections for Multiple Cores 41
How to Define Stack and Heap Memory Area in LCF 23
How to Define the Shared Memory 33
How to Define Unlikely Block of Code as Private Block of Code in a Multi-core Application 58
How to Define Virtual Memory for Read-Write-Execute (RWX) Access 38
How to Handle C++ Templates in Multi-core Applications 64
How to Limit Code and Data Visibility at Core Level 54
How to Make Code or Data Sections Visible to a Subset of Cores 62
How to Make LCF Compatible for Flexible Startup 16
How to Map Virtual Memory Areas to Physical Memory Address Space 45
How to Modify the LCF When Each Core Runs Different Code 29
How to Place a Symbol in Another Section in LCF 62
How to Reserve Physical Memory Area 40
How to Run Multiple Tasks on the Same Core 59
How to Setup Cache 21
How to Setup Virtual Trace Buffer (VTB) Using LCF 20
How to Share Code and Data Partially Among Different Cores 48
How to Specify the Content of Virtual Memory Areas 47
How to Troubleshoot Linker Error Messages 43

I
Introduction 11

L
LCF Expression Functions 123
LCF Expression Operators 129
LCF Preprocessing 131
Linker Configuration Concepts 69
Linker Configuration Tasks 13
Linker Error Messages 87
Linker Predefinitions 135
Linking Self-contained Libraries 79

N
num_core 127
num_task 126

O
originof 124

P
Parser Error Messages
physical_address 127
Predefined Physical Memory Regions 137

Predefined Symbols for MMU Descriptors 135

S
Scenario 1: True Private Code Model 58
Scenario 2: Code Partially Shared Among Different Cores 58
Sections in LCF 141
Setup Error Messages 117
sizeof 125
Specifying Address Translation Construct 78
Specifying Global Directives 71
Specifying Integers 71
Specifying Symbol Names 71
Specifying Target Architecture 72

T
task_id 125
Tasks 13
test_arch 127
The define Directive 132
The include Directive 132
to_physical 125

U
Understanding Cache Optimization in Linker 79
Understanding Flexible Segment Programming Model 84
Understanding Flexible Startup Configuration 81
Understanding L1 Defense 84
Understanding LCF Syntax 70
Understanding Linker Terminology 69
Understanding Startup Environment 83
Using Naming Conventions 70

V
vmorg 126