# Contents

1 Overview .................................................. 1
   1.1 Purpose ............................................. 1
   1.2 System Overview .................................... 1
   1.3 Document Overview ................................. 1
      1.3.1 General Modeling Framework .................. 2
      1.3.2 General Utilities .............................. 2
      1.3.3 Object Management (proxies) ................ 2
      1.3.4 Data Distribution Management (DDM) ........ 2
      1.3.5 External Modules .............................. 2
      1.3.6 Conventions .................................. 2
   1.4 Referenced Documents ............................... 2

2 General Modeling Framework ............................ 5
   2.1 RB\_SpDefineClass.H: RB\_DEFINE\_CLASS .......... 6
      2.1.1 Description ................................... 6
      2.1.2 Examples ..................................... 7
      2.1.3 Bugs ........................................ 8
      2.1.4 Used in ..................................... 8
      2.1.5 Notes ....................................... 8
   2.2 RB\_SpFrameworkFuncs.H: Standalone Rollbackable Functions ... 11
      2.2.1 Description ................................... 11
      2.2.2 Examples ..................................... 12
      2.2.3 See also .................................... 12
   2.3 SpAlt: Generic Rollback Mechanism ................. 13
      2.3.1 Description ................................... 13
      2.3.2 Examples ..................................... 13
      2.3.3 See also .................................... 16
   2.4 SpAltMgr: Alterable Item Manager .................. 17
      2.4.1 Description ................................... 17
      2.4.2 Examples ..................................... 17
      2.4.3 See also .................................... 17
      2.4.4 Notes ....................................... 17
   2.5 SpCancelHandle and RB\_SpCancelHandle: Event Cancel Handle 18
      2.5.1 Description ................................... 18
      2.5.2 Examples ..................................... 18
      2.5.3 See also .................................... 19
      2.5.4 Notes ....................................... 19
   2.6 SpDefineEvent.H: Point-To-Point Event Macros .... 20
      2.6.1 Description ................................... 20
      2.6.2 See also .................................... 24
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.7</td>
<td>SpDefineHandler.H: Handler Event Macros</td>
<td>25</td>
</tr>
<tr>
<td>2.7.1</td>
<td>Description</td>
<td>25</td>
</tr>
<tr>
<td>2.7.2</td>
<td>Examples</td>
<td>28</td>
</tr>
<tr>
<td>2.7.3</td>
<td>See also</td>
<td>30</td>
</tr>
<tr>
<td>2.8</td>
<td>SpDefineSimObj.H: Defining Simulation Object Macros</td>
<td>31</td>
</tr>
<tr>
<td>2.8.1</td>
<td>Description</td>
<td>31</td>
</tr>
<tr>
<td>2.8.2</td>
<td>Examples</td>
<td>32</td>
</tr>
<tr>
<td>2.8.3</td>
<td>See also</td>
<td>32</td>
</tr>
<tr>
<td>2.8.4</td>
<td>Notes</td>
<td>32</td>
</tr>
<tr>
<td>2.9</td>
<td>SpEvent: Generic Simulation Event Base Class</td>
<td>34</td>
</tr>
<tr>
<td>2.9.1</td>
<td>Description</td>
<td>34</td>
</tr>
<tr>
<td>2.9.2</td>
<td>See also</td>
<td>35</td>
</tr>
<tr>
<td>2.10</td>
<td>SPEEDES Global Functions</td>
<td>36</td>
</tr>
<tr>
<td>2.10.1</td>
<td>Description</td>
<td>36</td>
</tr>
<tr>
<td>2.10.2</td>
<td>See also</td>
<td>38</td>
</tr>
<tr>
<td>2.11</td>
<td>Simulation Initialization Macros</td>
<td>39</td>
</tr>
<tr>
<td>2.11.1</td>
<td>Description</td>
<td>39</td>
</tr>
<tr>
<td>2.11.2</td>
<td>See also</td>
<td>39</td>
</tr>
<tr>
<td>2.12.1</td>
<td>Description</td>
<td>40</td>
</tr>
<tr>
<td>2.12.2</td>
<td>See also</td>
<td>45</td>
</tr>
<tr>
<td>2.13.1</td>
<td>Description</td>
<td>46</td>
</tr>
<tr>
<td>2.13.2</td>
<td>Examples</td>
<td>51</td>
</tr>
<tr>
<td>2.13.3</td>
<td>See also</td>
<td>51</td>
</tr>
<tr>
<td>2.14</td>
<td>Global Functions for Scheduling Events</td>
<td>52</td>
</tr>
<tr>
<td>2.14.1</td>
<td>Description</td>
<td>52</td>
</tr>
<tr>
<td>2.14.2</td>
<td>See also</td>
<td>53</td>
</tr>
<tr>
<td>2.15</td>
<td>SpSimObj: Generic Simulation Object Base Class</td>
<td>54</td>
</tr>
<tr>
<td>2.15.1</td>
<td>Description</td>
<td>54</td>
</tr>
<tr>
<td>2.15.2</td>
<td>See also</td>
<td>57</td>
</tr>
<tr>
<td>2.16</td>
<td>SpSimObjMgr: Generic Simulation Object Manager</td>
<td>58</td>
</tr>
<tr>
<td>2.16.1</td>
<td>Description</td>
<td>58</td>
</tr>
<tr>
<td>2.16.2</td>
<td>Bugs</td>
<td>60</td>
</tr>
<tr>
<td>2.16.3</td>
<td>See also</td>
<td>60</td>
</tr>
<tr>
<td>2.16.4</td>
<td>Notes</td>
<td>60</td>
</tr>
<tr>
<td>2.17</td>
<td>SpSimTime: Simulation Time Abstraction</td>
<td>61</td>
</tr>
<tr>
<td>2.17.1</td>
<td>Description</td>
<td>61</td>
</tr>
<tr>
<td>2.17.2</td>
<td>Examples</td>
<td>63</td>
</tr>
<tr>
<td>2.17.3</td>
<td>See also</td>
<td>64</td>
</tr>
<tr>
<td>2.17.4</td>
<td>Notes</td>
<td>64</td>
</tr>
<tr>
<td>3</td>
<td>General Utilities</td>
<td>65</td>
</tr>
<tr>
<td>3.1</td>
<td>DEFINE_MEMPOOL: Macros for Memory Pool Management</td>
<td>66</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Description</td>
<td>66</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Examples</td>
<td>66</td>
</tr>
<tr>
<td>3.1.3</td>
<td>Bugs</td>
<td>67</td>
</tr>
<tr>
<td>3.1.4</td>
<td>Used in</td>
<td>67</td>
</tr>
<tr>
<td>3.2</td>
<td>C_ITEM: Base Class for Items Within a C_QUEUE (obsolete)</td>
<td>68</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>3.17</td>
<td>SpDataParser: Full Featured Data Parser (obsolete)</td>
<td>94</td>
</tr>
<tr>
<td>3.17.1</td>
<td>Description</td>
<td>94</td>
</tr>
<tr>
<td>3.17.2</td>
<td>Used in</td>
<td>97</td>
</tr>
<tr>
<td>3.17.3</td>
<td>Notes</td>
<td>97</td>
</tr>
<tr>
<td>3.18</td>
<td>SpDynPtrArray and RB_SpDynPtrArray: Dynamic Array of Pointers</td>
<td>98</td>
</tr>
<tr>
<td>3.18.1</td>
<td>Description</td>
<td>98</td>
</tr>
<tr>
<td>3.18.2</td>
<td>See also</td>
<td>98</td>
</tr>
<tr>
<td>3.19</td>
<td>C_FREE_LIST: Class for Maintaining Multiple Free Lists (obsolete)</td>
<td>99</td>
</tr>
<tr>
<td>3.19.1</td>
<td>Description</td>
<td>99</td>
</tr>
<tr>
<td>3.19.2</td>
<td>Examples</td>
<td>101</td>
</tr>
<tr>
<td>3.19.3</td>
<td>See also</td>
<td>101</td>
</tr>
<tr>
<td>3.20</td>
<td>SpHashTree and RB_SpHashTree: Hash Table of Trees</td>
<td>102</td>
</tr>
<tr>
<td>3.20.1</td>
<td>Description</td>
<td>102</td>
</tr>
<tr>
<td>3.20.2</td>
<td>Bugs</td>
<td>105</td>
</tr>
<tr>
<td>3.20.3</td>
<td>See also</td>
<td>105</td>
</tr>
<tr>
<td>3.21</td>
<td>SpHostUser: Internal Host Router Interface</td>
<td>106</td>
</tr>
<tr>
<td>3.21.1</td>
<td>Description</td>
<td>106</td>
</tr>
<tr>
<td>3.21.2</td>
<td>See also</td>
<td>108</td>
</tr>
<tr>
<td>3.22</td>
<td>SpList and RB_SpList: List Container Classes</td>
<td>109</td>
</tr>
<tr>
<td>3.22.1</td>
<td>Description</td>
<td>109</td>
</tr>
<tr>
<td>3.22.2</td>
<td>See also</td>
<td>110</td>
</tr>
<tr>
<td>3.23</td>
<td>Independent Iterators</td>
<td>111</td>
</tr>
<tr>
<td>3.23.1</td>
<td>Description</td>
<td>111</td>
</tr>
<tr>
<td>3.23.2</td>
<td>Examples</td>
<td>112</td>
</tr>
<tr>
<td>3.23.3</td>
<td>See also</td>
<td>112</td>
</tr>
<tr>
<td>3.23.4</td>
<td>Notes</td>
<td>112</td>
</tr>
<tr>
<td>3.24</td>
<td>SpMsg: Event Message Base Class</td>
<td>113</td>
</tr>
<tr>
<td>3.24.1</td>
<td>Description</td>
<td>113</td>
</tr>
<tr>
<td>3.24.2</td>
<td>Examples</td>
<td>114</td>
</tr>
<tr>
<td>3.25</td>
<td>SpNameValueList: List of Associated Names and Double Values</td>
<td>115</td>
</tr>
<tr>
<td>3.25.1</td>
<td>Description</td>
<td>115</td>
</tr>
<tr>
<td>3.25.2</td>
<td>Examples</td>
<td>116</td>
</tr>
<tr>
<td>3.25.3</td>
<td>Used in</td>
<td>116</td>
</tr>
<tr>
<td>3.26</td>
<td>SpObjHandle: Simulation Object Handle</td>
<td>117</td>
</tr>
<tr>
<td>3.26.1</td>
<td>Description</td>
<td>117</td>
</tr>
<tr>
<td>3.26.2</td>
<td>Examples</td>
<td>117</td>
</tr>
<tr>
<td>3.26.3</td>
<td>See also</td>
<td>117</td>
</tr>
<tr>
<td>3.27</td>
<td>SpParmSet and SpParmSetElement: Parameter Set for Interactions</td>
<td>118</td>
</tr>
<tr>
<td>3.27.1</td>
<td>Description</td>
<td>118</td>
</tr>
<tr>
<td>3.27.2</td>
<td>Examples</td>
<td>121</td>
</tr>
<tr>
<td>3.27.3</td>
<td>See also</td>
<td>121</td>
</tr>
<tr>
<td>3.27.4</td>
<td>Used in</td>
<td>121</td>
</tr>
<tr>
<td>3.27.5</td>
<td>Notes</td>
<td>121</td>
</tr>
<tr>
<td>3.28</td>
<td>SpParser: SPEEDES Parser</td>
<td>122</td>
</tr>
<tr>
<td>3.28.1</td>
<td>Description</td>
<td>122</td>
</tr>
<tr>
<td>3.28.2</td>
<td>See also</td>
<td>123</td>
</tr>
<tr>
<td>3.29</td>
<td>SpPriorityTree and RB_SpPriorityTree: Binary Tree Based Priority Queue</td>
<td>124</td>
</tr>
<tr>
<td>3.29.1</td>
<td>Description</td>
<td>124</td>
</tr>
<tr>
<td>3.30</td>
<td>SpRandom and RB_SpRandom: Random Number Generators</td>
<td>126</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>3.30.1 Description</td>
<td>126</td>
<td></td>
</tr>
<tr>
<td>3.30.2 Bugs</td>
<td>127</td>
<td></td>
</tr>
<tr>
<td>3.30.3 Notes</td>
<td>127</td>
<td></td>
</tr>
<tr>
<td>3.31 SpSet: Streamable Set of Sets And Elements</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>3.31.1 Description</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>3.31.2 See also</td>
<td>132</td>
<td></td>
</tr>
<tr>
<td>3.32 SpStopWatch: Stopwatch Timer Class</td>
<td>133</td>
<td></td>
</tr>
<tr>
<td>3.32.1 Description</td>
<td>133</td>
<td></td>
</tr>
<tr>
<td>3.32.2 Notes</td>
<td>134</td>
<td></td>
</tr>
<tr>
<td>3.33 SpString80: Fixed-Length String Class</td>
<td>135</td>
<td></td>
</tr>
<tr>
<td>3.33.1 Description</td>
<td>135</td>
<td></td>
</tr>
<tr>
<td>3.33.2 See also</td>
<td>135</td>
<td></td>
</tr>
<tr>
<td>3.34 SpTag: Trace File Debug Class</td>
<td>136</td>
<td></td>
</tr>
<tr>
<td>3.34.1 Description</td>
<td>136</td>
<td></td>
</tr>
<tr>
<td>3.34.2 Examples</td>
<td>136</td>
<td></td>
</tr>
<tr>
<td>3.34.3 Bugs</td>
<td>137</td>
<td></td>
</tr>
<tr>
<td>3.35 C_SQ_ITEM: Base Class for Items in RB_hash and RB_queue (obsolete)</td>
<td>138</td>
<td></td>
</tr>
<tr>
<td>3.35.1 Description</td>
<td>138</td>
<td></td>
</tr>
<tr>
<td>3.35.2 Examples</td>
<td>138</td>
<td></td>
</tr>
<tr>
<td>3.35.3 See also</td>
<td>139</td>
<td></td>
</tr>
<tr>
<td>4 Object Management (proxies)</td>
<td>141</td>
<td></td>
</tr>
<tr>
<td>4.1 F_SpProxyItem: Rollbackable Reference to an Object Proxy</td>
<td>142</td>
<td></td>
</tr>
<tr>
<td>4.1.1 Description</td>
<td>142</td>
<td></td>
</tr>
<tr>
<td>4.1.2 See also</td>
<td>142</td>
<td></td>
</tr>
<tr>
<td>4.2 BASE_ATTRIBUTE: Object Proxy Attribute Base Class</td>
<td>143</td>
<td></td>
</tr>
<tr>
<td>4.2.1 Description</td>
<td>143</td>
<td></td>
</tr>
<tr>
<td>4.2.2 Bugs</td>
<td>143</td>
<td></td>
</tr>
<tr>
<td>4.2.3 See also</td>
<td>144</td>
<td></td>
</tr>
<tr>
<td>4.3 BASE_DYNAMIC_ATTRIBUTE: Base Class for Dynamic Attributes</td>
<td>145</td>
<td></td>
</tr>
<tr>
<td>4.3.1 Description</td>
<td>145</td>
<td></td>
</tr>
<tr>
<td>4.3.2 See also</td>
<td>146</td>
<td></td>
</tr>
<tr>
<td>4.4 BINARY_BUFFER_ATTRIBUTE: Binary Buffer Attribute</td>
<td>147</td>
<td></td>
</tr>
<tr>
<td>4.4.1 Description</td>
<td>147</td>
<td></td>
</tr>
<tr>
<td>4.4.2 Examples</td>
<td>147</td>
<td></td>
</tr>
<tr>
<td>4.4.3 Bugs</td>
<td>148</td>
<td></td>
</tr>
<tr>
<td>4.4.4 See also</td>
<td>148</td>
<td></td>
</tr>
<tr>
<td>4.5 CIRCULAR_ORBIT: Circular Orbit Dynamic Position</td>
<td>149</td>
<td></td>
</tr>
<tr>
<td>4.5.1 See also</td>
<td>149</td>
<td></td>
</tr>
<tr>
<td>4.6 CONSTANT_MOTION: Constant Position Motion Dynamic Position</td>
<td>150</td>
<td></td>
</tr>
<tr>
<td>4.6.1 Description</td>
<td>150</td>
<td></td>
</tr>
<tr>
<td>4.6.2 See also</td>
<td>150</td>
<td></td>
</tr>
<tr>
<td>4.7 DOUBLE_ATTRIBUTE: Double Attribute</td>
<td>151</td>
<td></td>
</tr>
<tr>
<td>4.7.1 Description</td>
<td>151</td>
<td></td>
</tr>
<tr>
<td>4.7.2 See also</td>
<td>151</td>
<td></td>
</tr>
<tr>
<td>4.8 DYNAMIC_COMPLEX_EXPONENTIAL: Complex Exponential Dynamic Double</td>
<td>152</td>
<td></td>
</tr>
<tr>
<td>4.8.1 Description</td>
<td>152</td>
<td></td>
</tr>
<tr>
<td>4.8.2 See also</td>
<td>153</td>
<td></td>
</tr>
<tr>
<td>4.9 DYNAMIC_DOUBLE_ATTRIBUTE: Dynamic Double Attribute</td>
<td>154</td>
<td></td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>4.9.1</td>
<td>Description</td>
<td>154</td>
</tr>
<tr>
<td>4.9.2</td>
<td>Examples</td>
<td>154</td>
</tr>
<tr>
<td>4.9.3</td>
<td>Bugs</td>
<td>155</td>
</tr>
<tr>
<td>4.9.4</td>
<td>See also</td>
<td>155</td>
</tr>
<tr>
<td>4.9.5</td>
<td>Notes</td>
<td>156</td>
</tr>
<tr>
<td>4.10</td>
<td>DYNAMIC_DOUBLE_CONSTANT: Constant Dynamic Double</td>
<td>157</td>
</tr>
<tr>
<td>4.10.1</td>
<td>Description</td>
<td>157</td>
</tr>
<tr>
<td>4.10.2</td>
<td>See also</td>
<td>157</td>
</tr>
<tr>
<td>4.11</td>
<td>DYNAMIC_EXPONENTIAL: Exponential Equation Dynamic Double</td>
<td>158</td>
</tr>
<tr>
<td>4.11.1</td>
<td>Description</td>
<td>158</td>
</tr>
<tr>
<td>4.11.2</td>
<td>See also</td>
<td>158</td>
</tr>
<tr>
<td>4.12</td>
<td>DYNAMIC_EXTRAPOLATE: Extrapolation Dynamic Double</td>
<td>159</td>
</tr>
<tr>
<td>4.12.1</td>
<td>Description</td>
<td>159</td>
</tr>
<tr>
<td>4.12.2</td>
<td>See also</td>
<td>159</td>
</tr>
<tr>
<td>4.13</td>
<td>DYNAMIC_INT_ATTRIBUTE: Dynamic Integer Attribute</td>
<td>160</td>
</tr>
<tr>
<td>4.13.1</td>
<td>Description</td>
<td>160</td>
</tr>
<tr>
<td>4.13.2</td>
<td>Bugs</td>
<td>160</td>
</tr>
<tr>
<td>4.13.3</td>
<td>See also</td>
<td>160</td>
</tr>
<tr>
<td>4.13.4</td>
<td>Notes</td>
<td>160</td>
</tr>
<tr>
<td>4.14</td>
<td>DYNAMIC_INT_CONSTANT: Constant Dynamic Integer</td>
<td>161</td>
</tr>
<tr>
<td>4.14.1</td>
<td>Description</td>
<td>161</td>
</tr>
<tr>
<td>4.14.2</td>
<td>See also</td>
<td>161</td>
</tr>
<tr>
<td>4.15</td>
<td>DYNAMIC_LOGICAL_ATTRIBUTE: Dynamic Logical Attribute</td>
<td>162</td>
</tr>
<tr>
<td>4.15.1</td>
<td>Description</td>
<td>162</td>
</tr>
<tr>
<td>4.15.2</td>
<td>Bugs</td>
<td>162</td>
</tr>
<tr>
<td>4.15.3</td>
<td>See also</td>
<td>162</td>
</tr>
<tr>
<td>4.16</td>
<td>DYNAMIC_LOGICAL_CONSTANT: Constant Dynamic Logical</td>
<td>163</td>
</tr>
<tr>
<td>4.16.1</td>
<td>Description</td>
<td>163</td>
</tr>
<tr>
<td>4.16.2</td>
<td>See also</td>
<td>163</td>
</tr>
<tr>
<td>4.17</td>
<td>DYNAMIC_POLY_N: Polynomial Equation Dynamic Double</td>
<td>164</td>
</tr>
<tr>
<td>4.17.1</td>
<td>Description</td>
<td>164</td>
</tr>
<tr>
<td>4.17.2</td>
<td>See also</td>
<td>165</td>
</tr>
<tr>
<td>4.18</td>
<td>DYNAMIC_POSITION_ATTRIBUTE: Dynamic Position Attribute</td>
<td>166</td>
</tr>
<tr>
<td>4.18.1</td>
<td>Description</td>
<td>166</td>
</tr>
<tr>
<td>4.18.2</td>
<td>Examples</td>
<td>166</td>
</tr>
<tr>
<td>4.18.3</td>
<td>Bugs</td>
<td>168</td>
</tr>
<tr>
<td>4.18.4</td>
<td>See also</td>
<td>168</td>
</tr>
<tr>
<td>4.18.5</td>
<td>Notes</td>
<td>168</td>
</tr>
<tr>
<td>4.19</td>
<td>DYNAMIC_SPLINE_3: Cubic Spline Equation Dynamic Double</td>
<td>169</td>
</tr>
<tr>
<td>4.19.1</td>
<td>Description</td>
<td>169</td>
</tr>
<tr>
<td>4.19.2</td>
<td>See also</td>
<td>169</td>
</tr>
<tr>
<td>4.20</td>
<td>DYNAMIC_SPLINE_6: Quintic (fifth order) Spline Equation Dynamic Double</td>
<td>170</td>
</tr>
<tr>
<td>4.20.1</td>
<td>Description</td>
<td>170</td>
</tr>
<tr>
<td>4.20.2</td>
<td>See also</td>
<td>170</td>
</tr>
<tr>
<td>4.21</td>
<td>Elliptical: Elliptical Orbits Dynamic Position</td>
<td>171</td>
</tr>
<tr>
<td>4.21.1</td>
<td>Description</td>
<td>171</td>
</tr>
<tr>
<td>4.21.2</td>
<td>See also</td>
<td>172</td>
</tr>
<tr>
<td>4.22</td>
<td>EXTRAPOLATE_MOTION: Dead Reckoning Dynamic Position</td>
<td>173</td>
</tr>
<tr>
<td>4.22.1</td>
<td>Description</td>
<td>173</td>
</tr>
</tbody>
</table>
4.22.2 See also ................................................................. 173
4.23 SpFreeObjProxy: A Class for Managing Freelist of Proxies ................................. 174
  4.23.1 Description .......................................................... 174
  4.23.2 Examples ........................................................... 174
  4.23.3 See also .............................................................. 176
4.24 GREAT_CIRCLE: Great Circle Motion Dynamic Position ........................................ 177
  4.24.1 Description .......................................................... 177
  4.24.2 See also .............................................................. 178
4.25 INT_ATTRIBUTE: Integer Attribute ................................................................. 179
  4.25.1 Description .......................................................... 179
  4.25.2 Bugs ................................................................. 180
  4.25.3 See also .............................................................. 180
4.26 LIST_ATTRIBUTE: List Attributes ........................................................................ 181
  4.26.1 Description .......................................................... 181
  4.26.2 See also .............................................................. 181
4.27 LOGICAL_ATTRIBUTE: Logical Attribute .......................................................... 182
  4.27.1 Description .......................................................... 182
  4.27.2 See also .............................................................. 182
4.28 LOITER_MOTION: Circle about a Fixed Ground Point Dynamic Position ................ 183
  4.28.1 Description .......................................................... 183
  4.28.2 See also .............................................................. 184
4.29 OBJECT_ATTRIBUTE: Object Attributes ................................................................ 185
  4.29.1 Description .......................................................... 185
  4.29.2 Examples ........................................................... 186
  4.29.3 See also .............................................................. 187
4.30 POLY_N_MOTION: Least Squares Fit Polynomial Dynamic Position ......................... 188
  4.30.1 Description .......................................................... 188
  4.30.2 Bugs ................................................................. 189
  4.30.3 See also .............................................................. 189
4.31 POSITION_ATTRIBUTE: Position Attributes ......................................................... 190
  4.31.1 Description .......................................................... 190
  4.31.2 See also .............................................................. 190
4.32 RHUMB_LINE: Rhumb Line Motion Dynamic Position .............................................. 191
  4.32.1 Description .......................................................... 191
  4.32.2 See also .............................................................. 192
  4.32.3 Notes ................................................................. 192
4.33 S_SpHLA: Generic Self-Distributing Simulation Object ........................................... 193
  4.33.1 Description .......................................................... 193
  4.33.2 Examples ........................................................... 195
  4.33.3 See also .............................................................. 196
4.34 SpFreeDynAttributes: Dynamic Attribute Types Free List ...................................... 197
  4.34.1 Description .......................................................... 197
  4.34.2 See also .............................................................. 198
4.35 SpObjProxy: Base Class for Obtaining Proxy Values .............................................. 199
  4.35.1 Description .......................................................... 199
  4.35.2 Examples ........................................................... 204
  4.35.3 See also .............................................................. 206
4.36 SPLINE3_MOTION: Dynamic Position Fitting Position and Velocity ...................... 207
  4.36.1 Description .......................................................... 207
Chapter 1

Overview

1.1 Purpose

This document specifies the user interfaces for all of the classes in Synchronous Parallel Environment for Emulation and Discrete-Event Simulation (SPEEDES) which are needed or could be used to build a Parallel Discrete-Event Simulation (PDES). The Application Program Interface (API) for all relevant methods are specified and, in some cases, short examples are given. If additional information is required on usage, the SPEEDES User’s Guide should be consulted.

1.2 System Overview

The SPEEDES framework is a state-of-the-art object-oriented PDES Framework. This framework allows users to build both small and large optimistic time managed simulations. SPEEDES based simulations running on high performance computing platforms are able to incorporate far higher fidelity models and still achieve required run time performance. For real-time applications such as wargame and training simulations, the SPEEDES framework allows the use of high fidelity models while minimizing simulation lag behind wall clock time. Using SPEEDES, simulation performance is increased by processing events simultaneously on multiple processors. Further performance can be gained by utilizing optimistic processing. With SPEEDES’ optimistic processing algorithms, users can balance at risk forward processing of events that may have to be undone and reprocessed, with risk free forward processing of events whose output messages can be held until needed by other events. The current SPEEDES distribution incorporates a simulation engine that controls internal event processing, communications, and features to support a range of users.

1.3 Document Overview

Many documented classes contain additional public methods which are not documented within this API Reference Manual. Use of these methods is unsupported and these methods may be removed, changed, or modified in future releases. Users should avoid relying on undocumented and non-supported features of SPEEDES.

This document is organized into several major topics. Within a topic, a section is either a class or a header file. Header files are used whenever the information within the file does not fit neatly into a single
class. This includes groups of functions, global constants, or macros to be expanded. Sometimes, two
classes are presented in the same header file when it serves the purpose of simplifying the presentation
or when the classes have nearly identical APIs.

1.3.1 General Modeling Framework

Chapter 2 addresses standard modeling framework items. These include basic simulation objects,
events, and rollback infrastructure.

1.3.2 General Utilities

Chapter 3 examines utilities that are needed for general simulation support. These include rollbackable
classes, container classes, parsers and other support utilities.

1.3.3 Object Management (proxies)

Chapter 4 addresses the object proxy framework of SPEEDES. It examines proxies, their attributes, the
SSpHLA class, and a number of other classes needed to implement object proxies.

1.3.4 Data Distribution Management (DDM)

Chapter 5 looks at Data Distribution Management (DDM) in depth. There are several classes needed to
implement DDM which are presented here along with some examples of their use.

1.3.5 External Modules

Chapter 6 presents the external interface to SPEEDES. This includes the object proxy state manager,
events on that class, and other useful external interfaces.

1.3.6 Conventions

This document will frequently use the convention “<T>” to represent a given class name that should
be replaced in the text that follows. For example, The document may say: “RB(Create(T) where <T>
is any class” which means that any class name can be used in the place of ‘T’.

1.4 Referenced Documents

The following documents, although not necessarily referenced herein, guided preparation of this docu-
ment and its contents. Unless otherwise specified, the current revision shall apply.

Government
None.

Prime Contractor
1.4 Referenced Documents

<table>
<thead>
<tr>
<th>Reference</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>12146H601S</td>
<td>METRON SOW for Wargame 2000</td>
</tr>
</tbody>
</table>

**Contractor**
- S024 SPEEDES User’s Guide

**Other**
- 14882 ANSI standard for C++

Subcontractor Data Requirement List (SDRL) and other documentation can be found at:

Metron Incorporated  
514 Via De La Valle, Suite 306  
Solana Beach, CA 92075-2715
Chapter 2

General Modeling Framework
2.1 RB_SpDefineClass.H: RB_DEFINE_CLASS

2.1.1 Description

RB_SpDefineClass.H contains the macro, RB_DEFINE_CLASS, which generates functions for rollbackable memory management. For any given class, \(<T>\), RB_DEFINE_CLASS\((T)\) generates the following functions:

- **void RB_DELETE_ARRAY_T(T* t)**
  Rollbackably deletes an array of objects of type, \(T\). \(T\) may be char, int, double, etc., or any user-defined class passed to macro, RB_DEFINE_CLASS. For example, during the processing of an event, the code:

  ```cpp
  foo* cancel_item = RB_NEW_ARRAY_foo(3); // Described later
  ...
  RB_DELETE_ARRAY_foo(cancel_item);
  ```

  tells SPEEDES that cancel_item should be deleted with brackets, but SPEEDES will not delete it immediately. Rather, SPEEDES will wait until the event is committed to actually do the delete. This way, if the event gets rolled back, then SPEEDES will decide not to delete the memory. That code is similar to “delete [] cancel_item;”, except that, if the event is rolled back, the memory will not be deleted.

- **void RB_DELETE_T(T* t)**
  Rollbackably deletes an object of type \(T\). \(T\) may be char, int, double, etc., or any user-defined class passed to macro RB_DEFINE_CLASS. For example, during the processing of an event, the code

  ```cpp
  foo* cancel_item = RB_NEW_foo(); // Described later
  ...
  RB_DELETE_foo(cancel_item);
  ```

  tells SPEEDES that cancel_item should be deleted, but SPEEDES will not delete it immediately. Rather, SPEEDES will wait until the event is committed to actually do the delete. This way if the event gets rolled back, then SPEEDES will decide not to delete the memory. This code is similar to “delete cancel_item;”, except that, if the event is rolled back, the memory will not be deleted. Do not pass an array to this function (example of arrays are something returned by RB_NEW_ARRAY_T or new T[]).

- **T* RB_NEW_ARRAY_T(size_t n)**
  Rollbackably creates a new array of \(n\) objects of type, \(T\). \(T\) may be char, int, double, etc., or any user-defined class passed to macro, RB_DEFINE_CLASS. For example, the line “foo* link1 = RB_NEW_ARRAY_foo(3);” assigns to link1 an array of three foo objects during the processing of an event. It is similar to “foo* link1 = new foo[3];”, except that, if the event is rolled back, the memory will be deleted automatically by SPEEDES.

- **T* RB_NEW_T()**
  Rollbackably creates a new object of type, \(T\). \(T\) may be char, int, double, etc., or any user-defined class passed to macro, RB_DEFINE_CLASS. For example, during the processing of an event, the line “foo* cancel_item = RB_NEW_foo;” assigns to cancel_item a new object of type, foo. It is similar to “foo* cancel_item = new foo;”, except that, if the event is rolled back, the memory will be deleted automatically by SPEEDES.
2.1 RB_SpDefineClass.H: RB_DEFINE_CLASS

- class RB_PTR_T
  Defines a class that behaves like a rollbackable version of a “T*”. The API for this class is:

  - RB_PTR_T()
    Default constructor which has the pointer in an uninitialized state.
  - RB_PTR_T(T* t)
    Constructor of rollbackable pointer with an initial value of t.
  - operator T*()
    Conversion operator to allow direct casting to a T*.
  - operator =(T* t)
    Assignment operator from a T*.
  - operator ->()
    Arrow operator which allows for calling methods on the T*, which is represented by this class.

2.1.2 Examples

The following simulation object and simulation object event demonstrate many of the functions provided by RB_DEFINE_CLASS.

```cpp
#include "SpSimObj.H"
#include "SpDefineEvent.H"
#include "RB_SpDefineClass.H"
#include "SpGlobalFunctions.H"
#include "RB_ostream.H"

class OwnerInfo {
    public:
        char ownerName[80];
        char registrationId[40];
        char insurer[120];
        double salePrice;
    };

RB_DEFINE_CLASS(char);
RB_DEFINE_CLASS(OwnerInfo);

class S_Boat : public SpSimObj {
    public:
        void NewOwner(OwnerInfo newOwner) {
            RB_DELETE_ARRAY_char(currentName);
            currentName = RB_NEW_ARRAY_char(strlen(SpGetMsgData ()) + 1);
            strcpy(currentName, SpGetMsgData());
            RB_DELETE_OwnerInfo(ShipOwnerInfo);
            ShipOwnerInfo = RB_NEW_OwnerInfo();
            *ShipOwnerInfo = newOwner;
            RB_cout << "New ship owner is " << ShipOwnerInfo->ownerName << endl;
        }
    private:
        RB_PTR_char currentName;
        RB_PTR_OwnerInfo ShipOwnerInfo;
```
2.1.3 Bugs

Some compilers complain about calling RB_DEFINE_CLASS on primitive types, such as ints, doubles, and floats. Another macro, RB_DEFINE_CLASS_PRIMITIVE, can be used in these cases. This macro takes one argument just like RB_DEFINE_CLASS, provides a RB_PTR but does not provide the arrow operator (the “->” operator).

2.1.4 Used in


2.1.5 Notes

This macro does not make a class rollbackable. It only generates the rollbackable new and delete functions. Also, similar to C++ allocation, a mismatch between the array and non-array versions of new and delete may result in unexpected behavior.

Declaring RB_DEFINE_CLASS(foo) for a single class, foo, twice in any header or implementation file will result in a compile time error due to multiple-defined macro generated functions.

RB_DEFINE_CLASS(foo) expands into a great deal of code. The expansion (reformatted) of this macro is:

```c
static inline int RB_GET_foo_ID() {
    int opt = SpCurrentAltMgr->GetOptimistic();
    SpCurrentAltMgr->UnsetOptimistic();
    int rc = PO_GET_foo_ID();
    if (opt) {SpCurrentAltMgr->SetOptimistic();}
    return rc;
}

static inline int RB_GET_PO_PTR_foo_ID() {
    int opt = SpCurrentAltMgr->GetOptimistic();
    SpCurrentAltMgr->UnsetOptimistic();
    int rc = PO_GET_PO_PTR_foo_ID();
    if (opt) {SpCurrentAltMgr->SetOptimistic();}
    return rc;
}

inline void delete_single_foo(void* object) {
    delete((foo*)object);
}

inline void delete_single_foo_PTR(void* object) {
    delete((foo**)object);
}
```
2.1 RB_SpDefineClass.H: RB_DEFINE_CLASS

static inline foo* RB_NEW_foo(foo* object) {
    ErrorCatch();
    RB_NEW_HELPER(object, delete_single_foo, RB_GET_foo_ID(),
                   sizeof(foo));
    if (PersistenceEnabled) {
        PO_ADD_foo(object);
    }
    return object;
}

inline foo** RB_NEW_foo_PTR() {
    foo** object = new foo*;
    RB_NEW_HELPER(object, delete_single_foo_PTR, RB_GET_foo_ID(),
                   sizeof(foo));
    return object;
}

static inline void RB_DELETE_foo(foo* object) {
    RB_DELETE_HELPER(object, delete_single_foo, RB_GET_foo_ID(),
                      sizeof(foo));
}

inline void RB_DELETE_foo_PTR(foo** object) {
    RB_DELETE_HELPER(object, delete_single_foo_PTR);
}

inline foo* RB_NEW_foo() {
    if (PersistenceEnabled) {
        DefaultDataBase->IncrementAddLevelCount();
    }
    foo* object = new foo;
    RB_NEW_HELPER(object, delete_single_foo, RB_GET_foo_ID(),
                   sizeof(foo));
    if (PersistenceEnabled) {
        PO_ADD_foo(object);
        DefaultDataBase->DecrementAddLevelCount();
    }
    return object;
}

inline void delete_foo_array(void* array) {
    delete [] ((foo*)array);
}

inline void delete_foo_array_PTR(void* array) {
    delete [] ((foo**)array);
}

static inline foo* RB_NEW_ARRAY_foo(size_t num_foo) {
    if (PersistenceEnabled) {
        DefaultDataBase->IncrementAddLevelCount();
    }
    foo* object = new foo [num_foo];
    if (object == NULL) {
        cerr << "new operator returned NULL during call to "
        return object;
    }
}

ErrorCatch();
RB_NEW_HELPER(object, delete_single_foo, RB_GET_foo_ID(),
               sizeof(foo));
if (PersistenceEnabled) {
    PO_ADD_foo(object);
}
return object;

<< "RB_NEW_ARRAY_" << "foo" << endl;
abort();
}
RB_NEW_HELPER(object, delete_foo_array, RB_GET_foo_ID(),
    sizeof(foo) * num_foo);
if (PersistenceEnabled) {
    PO_ADD_foo(object, num_foo);
    DefaultDataBase->DecrementAddLevelCount();
}
return object;
}
inline foo** RB_NEW_ARRAY_foo_PTR(size_t num_foo) {
    foo** object = new foo * [num_foo];
    if (object == NULL) {
        cerr << "new operator returned NULL during call to "
            << "RB_NEW_ARRAY_" << "foo" << "_PTR" << endl;
        abort();
    }
    RB_NEW_HELPER(object, delete_foo_array_PTR);
    return object;
}
static inline void RB_DELETE_ARRAY_foo(foo* object) {
    RB_DELETE_HELPER(object, delete_foo_array, RB_GET_foo_ID());
}
inline void RB_DELETE_ARRAY_foo_PTR(foo** object) {
    RB_DELETE_HELPER(object, delete_foo_array_PTR);
}

class RB_PTR_foo {
public:
    RB_PTR_foo() {};
    RB_PTR_foo(foo* t) {Value = t;}
    operator foo*() (return (foo *) GetValue());
    foo* operator =(foo* t) {Value = t; return t;}
    foo* operator ->() {return (foo*) GetValue();}
private:
    void* GetValue() {return (void*) Value;}
    RB_voidPtr Value;
};
static void RB_REGISTER_CLASS_foo() {
    RB_GET_foo_ID();
    RB_GET_PO_PTR_foo_ID();
2.2 RB_SpFrameworkFuncs.H: Standalone Rollbackable Functions

2.2.1 Description

RB_SpFrameworkFuncs.H provides a set of standalone rollbackable functions that supplement the complement of rollbackable classes.

- **void RB_FREE_DELETE(C_FREE_LIST* freelist, C_ITEM* object)**
  Returns an item object to the free list, freelist, in a rollbackable fashion. This method ensures that the item is not returned to the free list until this event cannot be rolled back. Returning items to a free list that have not been obtained from a free list will result in undefined behavior.

- **C_ITEM* RB_FREE_NEW(C_FREE_LIST* freelist, int t)**
  Allocates an item of type, t, from the free list, freelist, and returns it in a rollbackable fashion. This will ensure that the memory will not be leaked and the free list will not grow abnormally large.

- **void* RB_memcpy(void* src, const void* dest, size_t size)**
  Rollbackably copies the size bytes from src to dest. Not having enough memory at dest for the copy will result in undefined behavior.

- **void RB_SaveState(void* buff, size_t size)**
  Saves the state of size bytes at buff. This can be used to rollbackably save the state of any class. One can save the state using this method and then overwrite the values during the same event. Upon a rollback, the object will be returned to its original state.

- **char* RB_strdup(const char* src)**
  Rollbackably returns a copy of the input string contained in src. Note: this function uses RB_NEW_ARRAY_char and not malloc.

- **void RBSendData(SpSimTime time, int externalId, char* data, int bytes, int globalId, SpHostUser* hostUser)**
  Rollbackably sends data to an external process through the host router. time is the current time; externalId is the identifier (id) of the object receiving the data; data is the data being sent; bytes is the size of the data; globalId is the global id of the object sending the data, and hostUser is the hostUser interface being used. Currently, only one host user interface is supported in SPEEDES.

- **void RBSendSubscribedData(SpSimTime time, char* name, char* data, int bytes, int globalId, SpHostUser* hostUser)**
  Rollbackably sends subscribed data to an external process through the host router. time is the current time; name is the name of the subscription; externalId is the id of the object receiving the data; data is the data being sent; bytes is the size of the data; globalId is the global id of the object
sending the data; and hostUser is the hostUser interface being used. Currently, only one host user interface is supported in SPEEDES.

### 2.2.2 Examples

RB_FREE_NEW and RB_FREE_DELETE are generally only used in the object proxy code for allocating or deallocating dynamic items.

```c
#include "S_SpHLA.H"
#include "RB_SpFrameworkFuncs.H"
#include "RB_SpDefineEvent.H"
#include "SpExportAttribute.H"
#include "SpGlobalFunctions.H"
#include "SpFreeDynAttributes.H"
extern SpFreeDynAttributes *FreeDynamicAttributes;

class S_Ship : public S_SpHLA {
public:
  S_Ship() : S_SpHLA("Ship") {}  
  //...
  void ChangeMotion() {
    double X[3];
    double EndTime = Position->GetEndTime();
    Position(EndTime, EARTH, X);  // Sets value of X

    CONSTANT_MOTION* ConstantMotion = (CONSTANT_MOTION *)
        RB_FREE_NEW(FreeDynamicAttributes, CONSTANT_MOTION_ID);
    ConstantMotion->SetEARTH();
    ConstantMotion->SetStartTime(EndTime);
    ConstantMotion->SetEndTime(EndTime + 1.0);
    ConstantMotion->SetPosition(X[0], X[1], X[2]);
    *Position += ConstantMotion;
  }
private:
  POSITION_ATTRIBUTE Position;
};
```

### 2.2.3 See also

2.3 SpAlt: Generic Rollback Mechanism

2.3.1 Description

The abstract base class SpAlt simplifies the process of making simulation state rollbackable. It is for advanced SPEEDES users. Generally, any new class created by users will automatically be rollbackable if it is made up of types (e.g. RB\_int, RB\_SpBinaryTree, etc.). However, users could have a class that represents a simulation state and is not covered by the rollbackable types provided, such as an opaque class provided by a third party, or a class that performs an operation that cannot be easily or efficiently canceled when a rollback occurs (like sending a Transmission Control Protocol/Internet Protocol (TCP/IP) message). In these cases, a user-created class that inherits from SpAlt may be needed.

An instance of a class that inherits from SpAlt is generally known as an “alt item”. Each alt item corresponds to a specific state change made in the context of an event being processed optimistically. If an event rolls back, its alt items know how to reverse the state changes that it represents.

- **int CheckUseOfMemoryRange(void* baseAddress, int numBytes) = 0**
  This pure virtual method is used to confirm that this alt item does not have a reference to any memory that falls in the range between baseAddress and (baseAddress + numBytes). This method should return 1 if there is such an overlap and 0 otherwise.

- **virtual void Cleanup(void) = 0**
  A pure virtual method called when the event that created the alt item is committed, meaning that further rollbacks or rollforwards are not possible and that whatever state the alt item is in is the correct one. At this point, any extra simulation state data that is being held onto in case of a rollback should be disposed of in the appropriate manner (e.g. delete for dynamic memory that is no longer needed).

- **char* GetType(void) = 0**
  A pure virtual method which returns a string that identifies the class that inherits from SpAlt. Typically, the string is simply the name of the class (e.g. “SpAltIntValue”).

- **int MemoryRegionsOverlap(void* baseAddress1, int numBytes1, void* baseAddress2, int numBytes2)**
  A protected method which can be used to help answer a call to the CheckUseOfMemoryRange method. This method returns 1 if any of the memory range [baseAddress1, baseAddress1 + numBytes1] overlaps with the range [baseAddress2, baseAddress2 + numBytes2] and returns 0 otherwise. See the Examples section below.

- **virtual void Rollback(void) = 0**
  A pure virtual method called when a simulation state controlled by the alt item needs to be reversed. This could happen if the event that created the alt item is rolled back, or if the event is rolled forward (i.e. a previous rollback is canceled). Thus, the method needs to be general enough to cover both cases.

2.3.2 Examples

Here is an example of a simple SpAlt class, designed to rollback the state of an integer. This is merely meant to illustrate how a SpAlt class can be created and used; the RB\_int class already provides easy-
to-use rollbackable integer functionality which would be represented by class RB_integer shown below. Also notice that the new alterable class (i.e. SpAltIntValue in this case) must supply a method which actually records the changing of the data. In this example, this method is called Alter.

```cpp
#include "SpAlt.H"

class SpAltIntValue : public SpAlt {

public:

  void Alter(int* ptrToInt) {
    ValuePtr = ptrToInt;
    Value   = *ValuePtr;
  }

  virtual void Rollback() {
    int temp = Value;
    Value = *ValuePtr;
    *ValuePtr = temp;
  }

  virtual void Cleanup() { ; } // No memory to be cleaned up

  virtual char* GetType() { return "SpAltIntValue"; }

  virtual int CheckUseOfMemoryRange(void* baseAddr, int size) {
    return MemoryRegionsOverlap(baseAddr, size,
                                ValuePtr, sizeof(int));
  }

private:

  int* ValuePtr;

  int Value;
};

// The following statement needs to be in the new class’ header file. // It will create a method to allocate an instance of the new class // from a free list. ALT_CHUNK_SIZE is the number of items created // in the free list at a single time and is a predefined value. One // can choose their own size here. See below.
DEFINE_MEMPOOL(SpAltIntValue, ALT_CHUNK_SIZE)
```

A “.C” file must also be created that has just the two lines:

```cpp
#include "SpAltIntValue.H"
PLUGIN_MEMPOOL(SpAltIntValue);
```

which actually creates the implementation of the free list for the “alt items”. Here is how this particular SpAlt class would be used.

```cpp
#include "SpAltMgr.H"
extern SpAltMgr* SpCurrentAltMgr;

void RB_integer::AssignInt(int newValue) {
  SpAltIntValue* altItem;

  if (SpCurrentAltMgr->GetOptimistic()) {
```
2.3 SpAlt: Generic Rollback Mechanism

```
altItem = ALLOCATE_SpAltIntValue(); // Created by DEFINE_MEMPOOL
altItem->Alter(&StateInteger); // StateInteger is part of
     // SomeClass
SpCurrentAltMgr->Insert(altItem);
}
StateInteger = newValue;
}
```

Thus, the SpAltIntValue instance holds onto the address of StateInteger and its value previous to the
assignment. If the assignment is rolled back, SpAltIntValue::Rollback is called, and StateInteger’s new
value is stored, while the previous value is restored. If the assignment is then rolled forward, SpAlt-
IntValue::Rollback is called again and the values are swapped once more. When the assignment is
committed, SpAltIntValue::Cleanup is called. But no clean up is necessary in this case, as no memory
was allocated. Note that SpCurrentAltMgr is a global pointer, much like SpCurrentSimObj or SpCur-
rentEvent.

SpAlt classes can also be used to implement actions that depend on simulation state but can not be
efficiently rolled back themselves. For example, sending a TCP/IP message that contains the value of a
RB_int. Instead of sending the message and then possibly sending cancel messages with rollbacks and
resending it with rollforwards, an alt item can be used.

```
#include "SpAlt.H"

class SpAltSendMsg : public SpAlt {
    public:
    void Alter(int valueToBeSent) {
        Value = valueToBeSent;
        RollbackFlag = 0;
    }

    virtual void Rollback() {
        RollbackFlag = ! RollbackFlag; // 0 to 1 or vice versa
    }

    virtual void Cleanup() {
        if (RollbackFlag == 0) {
            // If the event has not been rolled back, send the message here
        }
    }
    virtual char* GetType() { return "SpAltSendMsg"; }
    virtual int CheckUseOfMemoryRange(void* baseAddr, int size) {
        return 0; // No simulation state is being referenced here
    }

    private:
    int RollbackFlag;
    int Value;
};

DEFINE_MEMPOOL(SpAltSendMsg, ALT_CHUNK_SIZE)
```

Here is how this particular SpAlt class would be used:

```
#include "SpAltMgr.H"
extern SpAltMgr* SpCurrentAltMgr;
```
void SomeClass::SendMessage(int sendValue) {
    SpAltSendMsg* altItem;

    if (SpCurrentAltMgr->GetOptimistic()) {
        altItem = ALLOCATE_SpAltSendMsg();
        altItem->Alter(sendValue);
        SpCurrentAltMgr->Insert(altItem);
    } else {
        // Send the message here
    }
}

2.3.3 See also

SpAltMgr 2.4.
2.4 SpAltMgr: Alterable Item Manager

2.4.1 Description

The SpAltMgr class contains and manages instances of classes that inherit from SpAlt, otherwise known as “alt items”. Alt items correspond to state changes made in the context of an event being processed optimistically. Every SpEvent has its own SpAltMgr that holds the alt items generated when the event is processed. If an event is rolled back, its SpAltMgr goes through its internal queue of alt items and rolls back all state changes.

- **static int GetCheckMemoryRanges()**
  Returns the value of the CheckMemoryRanges flag, which determines whether or not the SpAlt::CheckUseOfMemoryRange method is called on alt items. Checking the memory ranges on the alt items can catch certain rollback related memory errors that are otherwise very hard to find and correct, but it does entail a minor decrease in performance. The flag is set by default.

- **int GetNumAltItems()**
  Returns the number of alt items (instances of classes that inherit from SpAlt) contained in the SpAltMgr.

- **static int GetOptimistic()**
  Returns the value of the optimistic processing flag. If the flag is set, the current event is processed optimistically and may be rolled back multiple times and reprocessed before being committed. If the flag is not set, any state changes made in the context of the current event will not be rolled back.

- **void Insert(SpAlt* altItem)**
  Adds the given alt item to the top of the SpAltMgr’s internal queue.

- **static void SetCheckMemoryRanges()**
  Sets the CheckMemoryRanges flag across all SpAltMgrs.

- **static void UnsetCheckMemoryRanges()**
  Unsets the CheckMemoryRanges flag across all SpAltMgrs, which will prevent memory checking operations on all alt items, as explained above, until the flag is set.

2.4.2 Examples

See the examples given in Section 2.3.

2.4.3 See also

SpAlt 2.3.

2.4.4 Notes

There is a global pointer, SpCurrentAltMgr, that points to the current event’s SpAltMgr.
2.5 SpCancelHandle and RB_SpCancelHandle: Event Cancel Handle

2.5.1 Description

When scheduling an event, a cancel handle can be stored so that the event can be canceled after being scheduled but before the event has been processed. The RB_SpCancelHandle and SpCancelHandle have identical APIs but the RB_CancelHandle has the advantage of being rollbackable.

- **int get_bytes()**
  Returns the size of the message for the event that was scheduled.

- **int get_etype()**
  Returns the event type that was scheduled and can be canceled.

- **SpEvent* get_event()**
  Returns the event that was scheduled. This only returns valid information when the event was scheduled on an object on the same node.

- **int get_node()**
  Returns the node of the object on which the event was scheduled.

- **int get_oid()**
  Returns the local id of the object on which the event was scheduled.

- **int get_obtype()**
  Returns the object manager type of the object on which the event was scheduled.

- **int get_rbid()**
  Returns the rollback id.

Canceling an event occurs through the scheduling of an event on that object that actually removes the event from the event queue. Attempting to cancel an event that is in the past will result in undefined behavior, possibly leading to the abnormal termination of the program.

2.5.2 Examples

Here is a quick example of storing a cancel handle in order to cancel a previously scheduled event:

```cpp
// S_Ship.H
#include "SpSimObj.H"
#include "RB_double.H"
#include "RB_int.H"
#include "RB_SpCancelHandle.H"
#include "RB_SpDefineEvent.H"
#include "RB_SpDefineSimObj.H"

class S_Ship : public SpSimObj {
public:
    S_Ship();
    void Init();
    void Dock();
};
```
2.5 SpCancelHandle and RB_SpCancelHandle: Event Cancel Handle

```cpp
void Scuttle();
private:
    RB_double CancelEventTime;
    RB_SpCancelHandle CancelHandle;
    RB_int Scuttled;
};
```

```cpp
DEFINE_SIMOBJ(S_Ship, 2, SCATTER);
DEFINE_SIMOBJ_EVENT_0_ARG(ShipDock, S_Ship, Dock);
DEFINE_SIMOBJ_EVENT_0_ARG(ShipScuttle, S_Ship, Scuttle);
```

```cpp
// S_Ship.C
#include "S_Ship.H"

S_Ship::S_Ship() {
    CancelEventTime = 1000; // Time at which event is scheduled
    CancelHandle = SCHEDULE_ShipDock(1000, SpGetObjHandle());
    Scuttled = 0;
}

S_Ship::Dock() {
    CancelEventTime = -1; // No longer can cancel this event and
    // we never dock at negative time.
}

S_Ship::Scuttle() {
    // We never scuttle a docked ship, only one that is at sea.
    // Of course, only cancel an event that is in the future.
    if (CancelEventTime > SpGetTime()) {
        SCHEDULE_CANCEL_EVENT(CancelHandle);
        // Set the time so that we do not cancel the event twice.
        CancelEventTime = -1;
        Scuttled = 1;
    }
}
```

2.5.3 See also


2.5.4 Notes

Because two internal fields will be adjusted in order to ensure that there is a unique order to the events in the simulation, the exact time the event is scheduled may be slightly different from the time originally requested. This should be taken into consideration when attempting to cancel an event. Only cancel events that are strictly in the future when comparing the floating point times between the event that was scheduled and the current simulation time.
2.6 SpDefineEvent.H: Point-To-Point Event Macros

2.6.1 Description

SpDefineEvent.H includes macros for defining event interfaces and classes. All of these macros are generally best placed in header files. Macros with arguments that are class names (e.g. simObjClass in the examples below) will cause compile errors if the class(es) in question is(are) not known by the compiler at that point. These macros only define the SPEEDES event classes, they still need to be plugged in before the simulation is started.

- **DEFINE_ASK_EVENT**

  - DEFINE_ASK_EVENT_INTERFACE_N_ARG(evName, arg1type, arg2type, ..., argNtype)
  - DEFINE_ASK_EVENT(evName, simObjClass, method, numArgs)
  - DEFINE_ASK_EVENT_N_ARG(evName, simObjClass, method, arg1type, arg2type, ..., argNtype)

Asks are a part of the process model and allow for the invocation of two way method calls between objects. These macros and their use are described in depth in the SPEEDES User’s Guide. The maximum number of arguments to an ask event is 8.

- **DEFINE_AUTONOMOUS_EVENT(evName, eventClass, method, numArgs)**

  An autonomous event stands alone from the simulation object that it acts upon, in contrast to the simulation object event detailed above. When the autonomous event is processed, it will invoke the specified method on the specified event class, which must inherit from SpEvent. The “numArgs” input to this macro is the number of arguments to the specified method. This macro will generate a descendant of the specified SPEEDES event class and a group of static inline functions, all of which are used internally. Usage of this macro depends on there being a DEFINE_EVENT_INTERFACE_N_ARG with the same event name and number of arguments somewhere else in the code. The signature of the method supplied to the DEFINE_AUTONOMOUS_EVENT macro must match the number and the types of arguments that were supplied to the DEFINE_EVENT_INTERFACE_N_ARG macro.

- **DEFINE_AUTONOMOUS_EVENT_N_ARG**

  - DEFINE_AUTONOMOUS_EVENT_0_ARG(evName, evtClass, method)
2.6 SpDefineEvent.H: Point-To-Point Event Macros

- **DEFINE_AUTONOMOUS_EVENT_1_ARG**(eventName, evtClass, method, arg1type)

- **DEFINE_AUTONOMOUS_EVENT_2_ARG**(eventName, evtClass, method, arg1type, arg2type)

- 

- **DEFINE_AUTONOMOUS_EVENT_8_ARG**(eventName, evtClass, method, arg1type, ..., arg8type)

This macro is a convenient combination of **DEFINE_EVENTINTERFACE_N_ARG** and **DEFINE_AUTONOMOUS_EVENT**. See **DEFINE_EVENTINTERFACE_N_ARG** and **DEFINE_AUTONOMOUS_EVENT** for examples of generated code.

- **DEFINE_EVENTINTERFACE_N_ARG**

- **DEFINE_EVENTINTERFACE_0_ARG**(eventName)

- **DEFINE_EVENTINTERFACE_1_ARG**(eventName, arg1type)

- **DEFINE_EVENTINTERFACE_2_ARG**(eventName, arg1type, arg2type)

- 

- **DEFINE_EVENTINTERFACE_8_ARG**(eventName, arg1type, ..., arg8type)

These macros will generate a schedule function and an event type id class specifically tailored for the supplied event name and argument list. A SPEEDES message class is also generated for internal usage. The interface can have up to eight argument types. These macros supply the means to schedule an event but have nothing to do with what happens when the event is processed. See the details on the **DEFINE_SIMOBJ_EVENT** and the **DEFINE_AUTONOMOUS_EVENT** macros below.

Useful code generated by these macros looks as follows:

```c
//DEFINE_EVENTINTERFACE_2_ARG(eventExample, NET_INT, MyClass)
SpCancelHandle SCHEDULE_eventExample(const SpSimTime& eventTime,  
const SpObjHandle& targetSimObj,  
NET_INT arg1,  
MyClass& arg2,  
const char* data = NULL,  
int dataBytes = 0);
```
class eventExample_EVENT_ID_TYPE {
    public:
        eventExample_EVENT_ID_TYPE();
        operator int();
    
};
static eventExample_EVENT_ID_TYPE eventExample_EVENT_ID;

Notice that the 2_ARG macro was used, and the “arg1” and “arg2” arguments to the schedule function. This macro creates a static global variable (e.g. eventExample_EVENT_ID in the above example) which can be used to access the integer event type id for this event.

All instances of DEFINE_EVENT_INTERFACE_N_ARG, DEFINE_LOCAL_EVENT_N_ARG, or DEFINE_SIMOBJ_EVENT_N_ARG generate a function for scheduling that particular event. This function always has the format:

```
SpCancelHandle SCHEDULE_eventExample(const SpSimTime& eventTime,
    /*obj*/ targetObj,
    arg1Type& arg1,
    arg2Type& arg2,
    ...,
    argNType& argN,
    const char* data = NULL,
    int dataBytes = 0);
```

where obj is an object handle in the case of defining an interface or a simulation object event. In the case of a local event, it is a reference class of the type upon which the event was defined.

This function copies out all data passed to it. The values passed in arg1, through argN are copied out using the copy constructor for their types and dataBytes of memory passed in the optional argument data are copied out as well. This means all of the arguments and the data can be freed (if dynamic) after a call to this method. When the event is executed, local copies of the arguments are passed to the event method. The memory passed in data is available through a call to SpGetMsgData() with the size of this data returned through a call to SpGetMsgDataBytes() (these are defined in SpGlobalFunctions.H in Section 2.10)

- **DEFINE_LOCAL_EVENT_N_ARG**

  - **DEFINE_LOCAL_EVENT_0_ARG**(evName, destClass, classMethod)

  - **DEFINE_LOCAL_EVENT_1_ARG**(evName, destClass, classMethod, arg1type)

  - **DEFINE_LOCAL_EVENT_2_ARG**(evName, destClass, classMethod, arg1type, arg2type)

  - ...
2.6 SpDefineEvent.H: Point-To-Point Event Macros

- `DEFINE_LOCAL_EVENT_8_ARG(eventName, destClass, classMethod, arg1type, ..., arg8type)

Normally, when events are scheduled, the target simulation object for the event is specified by its object handle (i.e. the combination of its node id, object type id, and local id). With local events, the target is not a SimObj. Instead, the target is an instance of any class that the scheduler has a reference to. When the event is processed, the specified method is invoked on the target object with the proper number and type of arguments. The method invoked when the local event is processed can have up to eight arguments. The useful code generated by these macros resembles the following:

```c
DEFINE_LOCAL_EVENT_2_ARG(localExample, localClass, localMethod, NET_INT, SpSimObj)
SpCancelHandle SCHEDULE_localExample(const SpSimTime& eventTime, const localClass& tObject, NET_INT arg1, SpSimObj arg2, const char* data = NULL, int dataBytes = 0);
```

```c
class localEventExample_EVENT_ID_TYPE {
    public:
        localEventExample EVENT_ID_TYPE ();
        operator int();
    }
static localEventExample_EVENT_ID_TYPE localEventExample_EVENT_ID;
```

Notice that the `2_ARG` macro was used and the “arg1” and “arg2” arguments were generated in the `SCHEDULE` function. This macro creates a static global variable (e.g. `localEventExample_EVENT_ID` in the above example) which can be used to access the integer event type id for this event.

- `DEFINE_SIMOBJ_EVENT(eventName, simObjClass, method, numArgs)`
A simulation object event is one that, when processed, will directly invoke a specific method (method) on the instance of the SpSimObj class that the event was scheduled for. The “numArgs” input to this macro is the number of arguments to the specified method. This macro will generate a SPEEDES event class and a group of static inline functions, all of which are used internally. Usage of this macro depends on there being a `DEFINE_EVENT_INTERFACE_N_ARG` with the same event name and number of arguments somewhere else in the code. The signature of the method supplied to the `DEFINE_SIMOBJ_EVENT` macro must match the number and the types of arguments that were supplied to the `DEFINE_EVENT_INTERFACE_N_ARG` macro.

- `DEFINE_SIMOBJ_EVENT_N_ARG`
– DEFINE_SIMOBJ_EVENT_0_ARG(evName, simObjClass, method)

– DEFINE_SIMOBJ_EVENT_1_ARG(evName, simObjClass, method, arg1type)

– DEFINE_SIMOBJ_EVENT_2_ARG(evName,
   simObjClass,
   method,
   arg1type,
   arg2type)

– ...

– DEFINE_SIMOBJ_EVENT_8_ARG(evName,
   simObjClass,
   method,
   arg1type,
   ..., arg8type)

This macro is a convenient combination of DEFINE_EVENT_INTERFACE_N_ARG and DEFINE_SIMOBJ_EVENT. See the macros DEFINE_EVENT_INTERFACE_N_ARG and DEFINE_SIMOBJ_EVENT above for examples of generated code.

2.6.2 See also

2.7 SpDefineHandler.H: Handler Event Macros

2.7.1 Description

SpDefineHandler.H includes macros for defining event handler interfaces and classes. Normal SPEEDES events are scheduled for a specific simulation object with the target’s SpObjHandle, but event handler events are somewhat different. There are two types of handler events: undirected and directed. Undirected handler events are scheduled with an optional string (called a trigger string) instead of a SpObjHandle, and are received by any simulation object or component that has registered an event handler for the same trigger string (via the SubscribeHandler method). Directed handler events are scheduled with an optional trigger string as well as a SpObjHandle and are received by the specified simulation object and any of its components that have registered an event handler for the same trigger string (via the AddHandler method).

The handlerName argument to the macros detailed below is not at all related to the trigger string used to schedule and subscribe to handler events. Rather, it is used to generate class names specific to the event handler being defined.

All of the macros here are generally best placed in header files. Macros with arguments that are class names (e.g. simObjClass used in the examples below) will cause compile errors if the class(es) in question is(are) not known by the compiler at that point.

- **DEFINE_HANDLER(handlerName, className, method)**
  Generates an event handler class and an id class. The id class is used to associate the handler class with a trigger string when subscribing to handler events (via AddHandler or SubscribeHandler). An instance of className must be provided when subscribing. When the subscribing simulation object receives a handler event with the proper trigger string and processes it, the generated handler class is used internally to invoke className::method on the instance of className supplied during subscription. The signature of className::method should be: void method(void).
  Handler events are scheduled with the SCHEDULE_HANDLER function. Any information that the sender wishes to send to the receiver(s) must be put into the event’s variable length data and extracted manually by the receiver(s) (see Section 2.14).
  The macro DEFINE_HANDLER expands as shown:
  
  ```
  DEFINE_HANDLER(handlerExample, containedClass, processMethod)
  ```
  ```
  class handlerExample_HDR_ID : public SpHandlerId {
    public:
      handlerExample_HDR_ID(containedClass* instancePtr);
  };
  ```

  See the Examples section below for more information on subscribing, scheduling, and processing handler events.

- **DEFINE_INTERACTION_HANDLER(handlerName, className, methodName)**
  Interactions are handler events where the scheduler passes information to the receiver(s) through a SpParmSet (one of the SPEEDES container classes), instead of the variable length data. This macro is identical to the DEFINE_HANDLER macro above, except that the signature of className::method should be: void method(SpParmSet&).
  Interactions are scheduled with the SCHEDULE_INTERACTION function (see Section 2.14).
  The macro DEFINE_INTERACTION_HANDLER expands as shown:
DEFINE_INTERACTION_HANDLER(interactionExample, containedClass, processMethod)

class interactionExample_HDR_ID : public SpHandlerId {
  public:
    interactionExample_HDR_ID(containedClass* instancePtr);
};

See the Examples section below for more information on subscribing, scheduling, and processing handler events.

- **DEFINE_SIMOBJ_HANDLER(handlerName, simObjClass, method)**
  This macro is identical to the DEFINE_HANDLER macro above, except that the supplied class-Name is assumed to be a descendant of SpSimObj. More specifically, simObjClass should be the class name of the subscribing simulation object. Because of that assumption, no instances are provided when the simulation object subscribes.
  The macro DEFINE_SIMOBJ_HANDLER expands as shown:

  DEFINE_SIMOBJ_HANDLER(handlerExample, S_ThisSimObjClass, processMethod)

  class handlerExample_HDR_ID : public SpHandlerId {
    public:
      handlerExample_HDR_ID();
  };

  See the Examples section below for more information on subscribing, scheduling, and processing handler events.

- **DEFINE_SIMOBJ_INTERACTION_HANDLER(handlerName, simObjClassName, methodName)**
  This macro is identical to the DEFINE_INTERACTION_HANDLER macro above, except that the supplied className is assumed to be a descendant of SpSimObj. More specifically, simObjClass should be the class name of the subscribing SimObj. Because of that assumption, no instances are provided when the simulation object subscribes.
  The macro DEFINE_SIMOBJ_INTERACTION_HANDLER expands as shown:

  DEFINE_SIMOBJ_INTERACTION_HANDLER(interactionExample, S_ThisSimObjClass, processMethod)

  class interactionExample_HDR_ID : public SpHandlerId {
    public:
      interactionExample_HDR_ID();
  };

  See the Examples section below for more information on subscribing, scheduling, and processing handler events.

- **DEFINE_HANDLER_INTERFACE N_ARG**
2.7 SpDefineHandler.H: Handler Event Macros

- DEFINE_HANDLER_INTERFACE_0_ARG(interfaceName)

- DEFINE_HANDLER_INTERFACE_1_ARG(interfaceName, arg1type)

- DEFINE_HANDLER_INTERFACE_2_ARG(interfaceName, arg1type, arg2type)

- ... 

- DEFINE_HANDLER_INTERFACE_8_ARG(interfaceName, arg1type, ..., arg8type)

These macros will generate a schedule function for the specified interface name and argument list. The interface can have up to eight argument types. Once the schedule function has been defined, handlers for that particular interface can be defined with the DEFINE_INTERFACE_HANDLER and DEFINE_SIMOBJ_INTERFACE_HANDLER macros, detailed above.

The macro DEFINE_HANDLER_INTERFACE_2_ARG expands, as shown, using types NET_INT and SpSimObj for this example:

```plaintext
DEFINE_HANDLER_INTERFACE_2_ARG(twoArgInterface,
    NET_INT, SpSimObj)
```

```plaintext
SpCancelHandle SCHEDULE_HANDLER_twoArgInterface(
    const SpSimTime& eventTime,
    const SpObjHandle& targetSimObj,
    NET_INT arg1,
    SpSimObj arg2,
    char* triggerString,
    const char* data = NULL,
    int dataBytes = 0);
```

```plaintext
SpCancelHandle SCHEDULE_HANDLER_twoArgInterface(
    const SpSimTime& eventTime,
    NET_INT arg1,
    SpSimObj arg2,
    char* triggerString,
    const char* data = NULL,
    int dataBytes = 0);
```

The first scheduled function is for directed interface handler events, and the second function is for undirected interface handler events. See the Examples section below for more information on subscribing, scheduling, and processing handler events.

- DEFINE_INTERFACE_HANDLER(handlerName, className, method, interfaceName, numArgs)

Interface handler events are similar to regular handler events, except that the scheduler passes information to the receiver(s) through a number of type-checked arguments, in addition to the variable length data. Unlike the other handler macros above, this macro does not stand alone. The interfaceName and numArgs arguments to this macro must up with a previous declaration of
the DEFINE_HANDLER_INTERFACE_N_ARG macro, detailed below. The signature of class-
Name::method should be: void method (<arguments>), where the number of arguments and their
types are determined by the matching DEFINE_HANDLER_INTERFACE_N_ARG macro.
The macro DEFINE_INTERFACE_HANDLER expands as shown:

```
DEFINE_INTERFACE_HANDLER(interfaceExample,
    containedClass,
    processMethod,
    twoArgInterface,
    2)
```

class interfaceExample_HDR_ID : public SpHandlerId {
    public:
        interfaceExample_HDR_ID(containedClass* instancePtr);
};

See the Examples section below for more information on subscribing, scheduling, and processing handler events.

- **DEFINE_SIMOBJ_INTERFACE_HANDLER**

```
DEFINE_SIMOBJ_INTERFACE_HANDLER(interfaceExample,
    S_ThisSimObjClass,
    processMethod,
    twoArgInterface,
    2)
```

class interfaceExample_HDR_ID : public SpHandlerId {
    public:
        interfaceExample_HDR_ID();
};

See the Examples section below for more information on subscribing, scheduling, and processing handler events.

### 2.7.2 Examples

Here is an example of defining various types of handlers:

```c++
#include "SpDefineHandler.H"
#include "SpDefineSimObj.H"
#include "SpComponent.H"
#include "SpSimObj.H"
```
class ExampleClass {
    public:
        HandleEvent();
};

class C_ExampleComponent : public SpComponent {
    public:
        HandleEvent();
        HandleTwoArgEvent(NET_INT arg1, SpSimTime arg2);
};

class S_ExampleSimObj : public SpSimObj {
    public:
        S_ExampleSimObj() {};
        void Init();
        void HandleEvent();
        void HandleInteraction(SpParmSet& params);
        void RegisterHandlers();
        ExampleClass NonSimObjHandler;
        C_ExampleComponent ComponentHandler;
};

DEFINE_HANDLER(HandlerForExampleClass,
    ExampleClass, HandleEvent)
DEFINE_HANDLER(HandlerForExampleComponent,
    C_ExampleComponent, HandleEvent)
DEFINE_SIMOBJ_HANDLER(SimObjHandler,
    S_ExampleSimObj, HandleEvent)
DEFINE_SIMOBJ_INTERACTION_HANDLER(SimObjInteractionHandler,
    S_ExampleSimObj, HandleInteraction)
DEFINE_HANDLER_INTERFACE_2_ARG(TwoArgInterface,
    NET_INT, SpSimTime)
DEFINE_INTERFACE_HANDLER(TwoArgComponentHandler,
    C_ExampleComponent, HandleTwoArgEvent,
    TwoArgInterface, 2)

Once the event handlers have been defined, a simulation object or a component can declare an interest in handler events of the right type with an optional trigger string. That interest is associated with one of the defined handlers. The simulation object or component must also decide if it wants only directed handler events or undirected ones. Declaring an interest in undirected handler events will cause the handler to not be invoked for directed handler events, and vice versa.

Here is an example of a simulation object registering the handlers defined above to be sensitive to certain trigger strings. Event handlers can be associated with multiple trigger strings.

void S_ExampleSimObj::RegisterHandlers() {
    // AddHandler is for directed handler events
    AddHandler(HandlerForExampleClass_HDR_ID(NonSimObjHandler),
        "Non-SimObj Handler Event");
    ComponentHandler.AddHandler(
        HandlerForExampleComponent_HDR_ID(ComponentHandler),
        "Component Handler Event");
// SubscribeHandler is for undirected handler events
SubscribeHandler(SimObjHandler_HDR_ID(),
    "SimObj Handler Event 1A");
SubscribeHandler(SimObjHandler_HDR_ID(),
    "SimObj Handler Event 2B");
AddHandler(SimObjInteractionHandler_HDR_ID(),
    "SimObj Interaction Event");
ComponentHandler.SubscribeHandler(
    TwoArgComponentHandler_HDR_ID(ComponentHandler),
    "Component TwoArg Event");
}

Here is an example of a simulation object sending out handler events for all of the defined types:

#include "SpSchedule.H"
#include "S_ExampleSimObj.H" // For the TwoArgInterface

void S_SomeOtherSimObj::ScheduleHandlerEvents(
    SpSimTime eTime, SpObjHandle target,
    const char* data, int dataBytes) {
    SCHEDULE_HANDLER(eTime, "Non-SimObj Handler Event",
        data, dataBytes);
    SCHEDULE_HANDLER(eTime, "Component Handler Event",
        data, dataBytes);
    SCHEDULE_HANDLER(eTime, target,
        "SimObj Handler Event 2B",
        data, dataBytes);
    SpParmSet dataSet;
    dataSet[0] = "Hello World!";
    SCHEDULE_INTERACTION(eTime, "SimObj Interaction Event",
        dataSet);
    NET_INT arg1 = 23;
    SpSimTime arg2 = 8.74;
    SCHEDULE_HANDLER_TwoArgInterface(eTime, target,
        arg1, arg2, "Component TwoArg Event",
        data, dataBytes);
}

2.7.3 See also

2.8 SpDefineSimObj.H: Defining Simulation Object Macros

2.8.1 Description

SpDefineSimObj.H includes macros for generating the necessary code to plug simulation objects and simulation object managers into a SPEEDES simulation. All of the macros here are generally best placed in header files. Macros with arguments that are class names (e.g. simObjClass) will cause compile errors if the class(es) in question are not known by the compiler at that point.

- **DEFINE_MANUAL_SIMOBJ_MGR_ID(simObjMgrClassName)**
  Obsolete version of DEFINE_SIMOBJ_MGR_ID kept for backward compatibility purposes. The useful code generated by this macro will look like the following:

  ```
  DEFINE_MANUAL_SIMOBJ_MGR_ID(O_ExampleSimObj)
  ```

  ```
  class O_ExampleSimObj_MGR_ID_TYPE {
  public:
    O_ExampleSimObj_MGR_ID_TYPE();
    operator int();
  }
  static O_ExampleSimObj_MGR_ID_TYPE O_ExampleSimObj_MGR_ID;
  ```

- **DEFINE_MANUAL_SIMOBJ_MGR(simObjMgrClassName)**
  Obsolete parallel to DEFINE_SIMOBJ_MGR kept for backward compatibility purposes when the user wrote their own SimObjMgr. This macro is meant to be used with the PLUG_IN_SIMOBJ-MGR (see Section 2.11) macro in order to fully introduce the new object class into the SPEEDES simulation. The DEFINE_MANUAL_SIMOBJ_MGR_ID macro is invoked as a part of this one.

- **DEFINE_SIMOBJ(simObjClassName, numSimObjs, decompAlg)**
  Takes a user-written simulation object class and creates the code needed to plug it into the SPEEDES simulation. The numSimObjs argument can either be a hard-coded number or any function that takes no arguments and returns an integer. The decompAlg argument can be “SCATTER”, “BLOCK”, ”FILE_SCATTER”, or “FILE_BLOCK”, depending on how the user wants these objects dealt to the simulation nodes.

  Users do not need to create their own simulation object manager for the SpSimObj class when using this macro. The manager is generated automatically. The useful code generated by this macro will look like the following:

  ```
  DEFINE_SIMOBJ(S_ExampleSimObj, 5, SCATTER)
  ```

  ```
  class S_ExampleSimObj_MGR : public O_SpDecomp_SCATTER {
  public:
    S_ExampleSimObj_MGR();
    virtual SpSimObj* CreateSimObj();
    virtual void DestroySimObj(SpSimObj* simObj);
    virtual char* GetSimObjMgrName();
  }
  ```

  The DEFINE_SIMOBJ_MGR_ID (see above) and RB_DEFINE_CLASS (see Section 2.1) macros are also invoked from this macro.

- **DEFINE_SIMOBJ_MGR_ID(typeName)**
  Declares a simulation object manager type. It creates an integer id named after the simulation
object manager for identifying the simulation object manager id part of SpObjHandles. All types of simulation objects must have a simulation object manager declared for them. The typeName argument can be anything, but typically it is the same as the name of the simulation objects the manager controls, minus the “S_” prefix.

The useful code generated by this macro will look like the following:

```cpp
DEFINE_SIMOBJ_MGR_ID(ExampleSimObj)

class ExampleSimObj_MGR_ID_TYPE {
    public:
    ExampleSimObj_MGR_ID_TYPE();
    operator int();
};
static ExampleSimObj_MGR_ID_TYPE ExampleSimObj_MGR_ID;
```

These macros must be used with the PLUG_IN_SIMOBJ (see Section 2.11) macro in order to fully introduce the new object class into the SPEEDES simulation. See the Examples section below for more information on creating SpSimObjs.

### 2.8.2 Examples

Here is an example of defining a simulation object:

```cpp
#include "SpMainPlugIn.H"
#include "SpDefineSimObj.H"

class S_ExampleSimObj : public SpSimObj {
    public:
    S_ExampleSimObj() {}
    void Init();
};

DEFINE_SIMOBJ(S_ExampleSimObj, 7, BLOCK)

int main(int argc, char** argv) {
    PLUG_IN_SIMOBJ(S_ExampleSimObj);
    ExecuteSpeedes(argc, argv);
    return 0;
}
```

### 2.8.3 See also

SpSimObjMgr 2.16, SpSimObj 2.15, SpObjHandle 3.26, and SpMainPlugIn.H 2.11.

### 2.8.4 Notes

`DEFINE_SIMOBJ(S_MySimObj, 17, SCATTER)` expands into the code:

```cpp
PO_DEFINE_CLASS(S_MySimObj_MGR) // See PO.H for this macro
class S_MySimObj_MGR : public O_SpDecomp_SCATTER {
```
public:
    S_MySimObj_MGR () {
        CreateSimObjs( 17 );
    }
    virtual SpSimObj* CreateSimObj() {
        return PO_NEW_S_MySimObj ();
    }
    virtual void DestroySimObj(SpSimObj* simObj) {
        RB_DELETE_S_MySimObj((S_MySimObj*) simObj);
    }
    virtual char* GetSimObjMgrName() {
        return "S_MySimObj_MGR";
    }
};

static inline void* new_S_MySimObj_MGR () {
    return PO_NEW_S_MySimObj_MGR ();
}

class S_MySimObj_MGR_ID_TYPE {
public:
    S_MySimObj_MGR_ID_TYPE() : val(-1) {}
    operator int() {
        if (val == -1)
            val = SpGetSimObjMgrId("S_MySimObj_MGR");
        return val;
    }
private:
    int val;
};

static S_MySimObj_MGR_ID_TYPE S_MySimObj_MGR_ID;
2.9 SpEvent: Generic Simulation Event Base Class

2.9.1 Description

This is the fundamental active building block in SPEEDES applications that operates on simulation objects. It is also the way SPEEDES passes simulated time. SpEvents are discrete events that are designed to act on a single class (i.e. type) of simulation object (i.e. a child of SpSimObj). Events are generally also responsible for scheduling other events.

- Virtual methods. These methods may be overridden by users creating their own events instead of using the DEFINE_EVENT macros (see Section 2.6). Their default behavior is to do nothing, and overriding them is optional.

  - virtual void init(SpMsg* msg)
    This method is called when the event is constructed. Any state changes done inside this method will not be rolled back.

  - virtual void process()
    This method is where the event does its computations and state changes, and should almost always be overridden by the user. Because of rollbacks, this method may be called more than once for the same event, so any state changes should either be done in a rollback-conscious manner or moved to the commit method.

  - virtual void commit()
    This method is a sort of addendum to the process method above, that is called after the event has been committed (i.e. after it has been passed by Global Virtual Time (GVT) and is, thus, incapable of rolling back). Anything that the user wishes to accomplish with the event that cannot be rolled back or should not be executed more than once (e.g. sending state data to a graphical display) can be put in here.

  - virtual void exchange()
    This method is called after the event is processed, before it is rolled back, and is generally not needed. But if rollbackable functions are not available for certain classes that contain state data, this method allows the user to write low-level code to exchange a value with a saved previous value.

  - virtual int lazy()
    This method is called before an event is to be reprocessed after a rollback, if lazy cancellation is enabled. The method should return 1 if this event does not need to be reprocessed (because processing the straggler event that triggered the rollback did not effect the outcome of this event), or 0 otherwise.

  - virtual void cleanup()
    This method is the complement of init. It is invoked when the event is destructed.

- int event_type(char* eventTypeName)
  Returns the event type id for the given event type name.

- void* get_manager(int objectType = -2)
  Returns a pointer to the SpSimObjMgr on this node for the given object type id.

- int GetEventId()
  Gets the event type id for this event
2.9 SpEvent: Generic Simulation Event Base Class

- **char* GetHandlerTrigger()**
  If this event is an event handler event, calling this method will return the string name of the handler.

- **char* GetMsgData()**
  Gets a pointer to the variable-length data for this event.

- **int GetMsgDataBytes()**
  Gets the number of bytes in the variable-length data for this event.

- **int GetMsgSize(int eventType)**
  Gets the number of bytes for the message header data for the supplied event type id.

- **int GetNodeId()**
  Gets the node (i.e. processor number on local machine) on which this event is executing.

- **SpObjHandle GetObjHandle()**
  Gets the object handle for this event’s SpSimObj. The object handle is the combination of the SpSimObj’s node id, object type id, and local id.

- **SpSimObj* GetSimObj()**
  Retrieves the pointer to the SpSimObj that this event was scheduled for.

- **int GetSimObjGlobalId()**
  Gets the global id for this event’s SpSimObj, which is unique throughout the simulation.

- **int GetSimObjLocalId()**
  Gets the local id for this event’s SpSimObj.

- **int GetSimObjMgrId()**
  Gets the object type id for this event’s SpSimObj.

- **SpMsg* GetMsg()**
  Gets a pointer the SpMsg header for this event.

- **SpSimTime& GetTime()**
  Gets the time that this event was scheduled for.

- **void memory_print()**
  Prints out event free list memory statistics.

- **int object_type(char* objecType_Name)**
  Returns the object type id for the given object type name.

### 2.9.2 See also

2.10 SPEEDES Global Functions

2.10.1 Description

SpGlobalFunctions.H includes a large number of global functions for accessing a great deal of information about the simulation, the current SimObj, and the current event being processed.

- **void SpAddHandler(const SpHandlerId& handlerId, char* trigger = NULL)**
  Adds the handler specified by handlerId with the trigger.

- **SpEvent* SpGetEvent()**
  Returns a pointer to the actual event being processed. This event is hidden from the user when the unified API is used for generating events.

- **int SpGetEventId()**
  Every event type has a unique id. This returns the id of the current event.

- **int SpGetExternalId()**
  External modules can schedule events on objects through the method, SpEmHostUser::SendTwo-WayCommand, inside the simulation and expect a response. This contains the id of the external module that is expecting the response.

- **char* SpGetHandlerTrigger()**
  Returns a pointer to the trigger that caused the current event handler to be executed. Do not delete the value returned by this function.

- **SpHostUser* SpGetHostUser()**
  Returns a pointer to the SpHostUser for communication with the SpEmHostUser. Do not delete the value returned by this function. SpHostUser provides the mechanism to communicate with external applications through SpEmHostUser.

- **SpMasterSimObjMgr* SpGetMasterSimObjMgr()**
  Returns a pointer to the master simulation object manager. This is the simulation object that manages all the simulation object managers. Do not delete the value returned by this function.

- **SpMsg* SpGetMsg()**
  Returns the actual SPEEDES message used for scheduling an event. This message is macro generated whenever the unified API is used for generating events.

- **char* SpGetMsgData()**
  Every SCHEDULE_* function has two additional optional arguments called char* data and int dataBytes. This function returns a pointer to this optional data. Do not delete the value returned by this function.

- **int SpGetMsgDataBytes()**
  Every SCHEDULE_* function has two additional optional arguments called char* data and int dataBytes. This function returns the number of bytes in the additional data (dataBytes).

- **int SpGetNodeId()**
  Returns the id of the node on which this event is being executed.

- **int SpGetNumNodes()**
  Returns the number of nodes in the simulation.
2.10 SPEEDES Global Functions

- SpObjHandle SpGetObjHandle()
  Returns the object handle for the current simulation object.

- SpObjHandle SpGetObjHandle(char* mgrName, char* objName)
  Returns the object handle for the object with name, objName, and manager of type, mgrName.

- SpObjHandle SpGetObjHandle(char* mgrName, int simObjKindId)
  Returns the object handle for the manager of type, mgrName, and kind id, simObjKindId.

- SpObjHandle SpGetObjHandle(int simObjKindId)
  Returns the object handle for the simulation object with the same type (same simulation object manager id) as the current object with the simObjKindId.

- SpObjHandle SpGetObjHandle(int mgrId, char* objName)
  Returns the object handle for the object with manager id, mgrId, and name, objName.

- SpObjHandle SpGetObjHandle(int mgrId, int simObjKindId)
  Returns the object handle for object with manager id, mgrId, and kind id, simObjKindId.

- RB_SpRandom* SpGetRandom()
  Every simulation object has a random number generator. This returns a pointer to the rollbackable random number generator for the current simulation object. Do not delete the value returned by this function.

- SpSimObj* SpGetSimObj()
  Returns the simulation object that is currently being acted upon by the current event.

- int SpGetSimObjGlobalId()
  Each simulation object is assigned a unique simulation-wide id, which is called global id. This function returns the value of the unique global id.

- int SpGetSimObjKindId()
  Each simulation object manager assigns a unique id to its objects across all the nodes. This function returns the value of the unique id.

- int SpGetSimObjLocalId()
  Each manager on each node manages a set of simulation object. Each object has an id unique to that manager on that node. This function returns the id of the object for which the current event or handler is being executed.

- SpSimObjMgr* SpGetSimObjMgr()
  Returns a pointer to the current SimObjMgr. Do not delete the return value of this function.

- SpSimObjMgr* SpGetSimObjMgr(char* mgrName)
  Returns a pointer to the manager with name, mgrName. Do not delete the return value of this function.

- SpSimObjMgr* SpGetSimObjMgr(int mgrId)
  Returns a pointer to the manager with id, mgrId. Do not delete the return value of this function.

- int SpGetSimObjMgrId()
  Each simulation object manager has a unique type or id and this id is used in the SpObjHandle for scheduling events. This function returns the id of the object manager for the current simulation object.
• **int SpGetSimObjMgrId(char* n)**
  Each object manager has a unique type or id and this id is used in the SpObjHandle for scheduling events. This function returns the id of the object manager whose name is n.

• **char* SpGetSimObjName()**
  Returns the simulation object name (the name is set by the method SetName(char*) on SpSimObj). Do not delete the value returned by this function.

• **SpSimTime& SpGetTime()**
  Returns the simulation time for the current event.

• **SpTag* SpGetTag()**
  Returns the tag for the current event.

• **void SpRemoveHandler(const SpHandlerId& handlerId, char* trig = NULL)**
  Removes (unsubscribes) the handler type specified by handlerId and trigger, trig, for the current simulation object and undirected handlers.

• **void SpSubscribeHandler(const SpHandlerId& handlerId, char* trig = NULL)**
  Subscribes the current simulation object with handler specified by handlerId to undirected handlers with trigger, trig.

### 2.10.2 See also

2.11 Simulation Initialization Macros

2.11.1 Description

SpMainPlugIn.H gives the users the functions and macros necessary to set up the simulation and start it running.

- **int ExecuteSpeedes(int argc = 0, char** ** argv = NULL)**
  Starts the simulation. Once this function is called, SPEEDES takes over until the simulation reaches the end. Simulation object and event types should be plugged into SPEEDES before this function is invoked. Users should also pass the parameters from int main(int, char** ) to this function.

- **PLUG_IN_EVENT(eventName)**
  Adds the specified event to the simulation. This macro should be used before the call to ExecuteSpeedes. The event type in question must be defined with one of the macros found in SpDefineEvent.H (see Section 2.6) elsewhere in the code in order for this macro to work.

- **PLUG_IN_SIMOBJ(simObjClassName)**
  Adds the specified SpSimObj class to the simulation. This macro should be called before the call to ExecuteSpeedes. The SpSimObj class in question must be defined with the DEFINE_SIMOBJ macro (see Section 2.8) elsewhere in the code in order for this macro to work.

- **PLUG_IN_SIMOBJ_MGR(simObjMgrClassName)**
  Adds the specified manual SpSimObjMgr class to the simulation. This macro should be called before the call to ExecuteSpeedes. The SpSimObjMgr class in question must be defined with the DEFINE_MANUAL_SIMOBJ_MGR macro (see Section 2.8) elsewhere in the code in order for this macro to work.

- **SET_CLUSTER(clusterName)**
  Sets the current cluster to clusterName. This remains in effect until further SET_CLUSTER calls are made.

- **SET_NUM_FREE(numFree)**
  Sets the default number of items created in the free list for events. This value remains in effect until another SET_NUM_FREE call is made.

2.11.2 See also


2.12.1 Description

The file, SpProc.H, provides all of the macros needed to support processes. Processes are point-to-point events that are re-entrant. Normally, events occur at an instant in simulated time. Conversely, processes allow the event to occur in phases separated by simulation time intervals. Macros described in this section support the two functions currently provided by the process model: waits and asks.

- Process model infrastructure: All process models must have the following macros defined in their process model event:

  - **P_BEGIN(numReentryLabels)**
    This macro marks the start of the process model user code. This macro is placed right after the last P_LV variable or the last local variable, whichever is last. The argument, numReentryLabels, corresponds to the total number of macros in the process that signify a potential re-entry point (i.e. waits and asks):
    ```
    void MyProcess(SpObjHandle oh) {
        P_VAR;
        P_LV(int, a);
        P_BEGIN(3);
        ASK_1_ARG(1, 100.0, oh, GetA, a, NULL, 0);
        RB_cout << "a = " << a << endl;
        WAIT(2, 5.0);
        RB_cout << "Waited 5 ticks" << endl;
        WAIT_FOR(3, sem, -1);
        RB_cout << "Semaphore set " << a << endl;
        P_END;
    }
    ```

  - **P_END**
    This macro is the final process model construct needed to transform a point-to-point event into a process model. The macro is placed at the end of the process algorithm, as shown below:
    ```
    void MyProcess(SpObjHandle oh) {
        P_VAR;
        P_LV(int, a);
        P_BEGIN(1);
        ASK_1_ARG(1, 100.0, oh, GetA, a, NULL, 0);
        RB_cout << "a = " << a << endl;
        P_END;
    }
    ```

  - **P_LV**
    This process model macro is used to declare process model state variables which are local to this process model event. Any variables which are declared using this macro will be rollbackable for this process model event. This macro is used only if the process model event needs a local state variable, hence an optional macro for the process model. If it is used, then the macro definitions must physically lie between macros P_VAR and P_BEGIN. The example below shows variable, “a”, being declared as a local state variable, since it is declared using the macro, P_LV. Variable “i” is declared normally, hence a local stack variable whose state is not retained on rollbacks.

```c
void MyProcess(SpObjHandle objHandle) {
    P_VAR;
P_LV(int, a);
    int i;
P_BEGIN(1);
    ASK_1_ARG(1, SpGetTime() + 2.0, objHandle, GetA, a, NULL, 0);
    for (i = 0; i < 5; i++) {
        SCHEDULE_Update(SpGetTime() + 5.0, SpObjHandle(0, 0, i), a);
    }
P_END;
}

- P_VAR
  This macro must be the first line in the process model event.

- ASK_*_ARG(reentryLabel, time, objHandle, eventName, args, data, bytes)
  This macro commands a process to
  1. send arguments to a specified non-local method,
  2. temporarily exit,
  3. invoke the specified method on a non-local simulation object with any of up to eight parameters in combination of input, output, and input-output parameters, and then,
  4. resume with its output and input-output arguments appropriately modified by the non-local method.

As such, it is the SPEEDES functional equivalent to a remote procedure call in UNIX. It is a remote call in the sense that a simulation object calls a two-way method on another simulation object that may be on a remote node. The method invoked must be registered as an ASK method (see Section 2.6). The “*” in ASK_*_ARG is substituted with an integer (0 through 8) representing the number of arguments in the ASK method.

The first argument, reentryLabel, is an integer index representing the numeric order in which the process model construct is being used (i.e. the first wait uses 1, the second wait uses 2, etc.).

The second argument, time, is the simulation time at which to schedule the method invocation on the remote simulation object.

The third and fourth arguments, objHandle and eventName, are the object handle (SpObjHandle) and event name of the ASK method.

The next zero to eight arguments, args, make up an ordered list of arguments (dereferenced, not by pointer) to pass as parameters to the ASK method. These will be type-checked by the compiler. Arguments that are represented as by-reference parameters in the ASK signature and that are modified by the ASK method will have those modifications copied to the corresponding ASK arguments when the process resumes. The ASK method may also be a process that passes simulation time. Thus, the duration of simulation time that elapses when calling ASK is the duration until the time argument plus, if the ASK method is a process, the time elapsed during the ASK method.

The final two arguments, data and bytes, are the user data buffer and its length in bytes to pass to the ASK method. These arguments are not modified by the ASK method. These arguments are not optional, so one must pass NULL and 0 for these last two arguments, even if they are not used.
void MyProcess(SpObjHandle oh) {
    P_VAR;
    P_LV(int, a);
    P_BEGIN(1);
    ASK_1_ARG(1, 100.0, oh, GetA, a, NULL, 0);
    RB_cout << "a = " << a << endl;
    P_END;
}

• RETURN
This macro is functionally equivalent to the keyword, return, in C++, but must be used in place of return in process methods. This allows a process method to properly clean up on its exit. Since process methods return void, this macro has no arguments and is unnecessary unless the process terminates before the end of the algorithm. An example is shown below:

void MyProcess(SpLogicalSem& sem) {
    P_VAR;
    P_LV(int, loop);
    P_BEGIN(1);
    timeOuts = 0;
    for(loop = 0; loop < 5; loop++) {
        timeOuts = 0;
        if (sem.IsSet()) {
            RB_cout << "Semaphore set; exiting" << endl;
            RETURN;
        }
    }
    RB_cout << "Timed out 5 times; giving up" << endl;
    P_END;
}

• WAIT(reentryLabel, duration)
This macro commands a process to exit and then resume at a later simulation time. The first argument, reentryLabel, is an integer index representing the numeric order in which the process model construct is being used (i.e. the first wait uses 1, the second wait uses 2, etc.). The second argument, duration, specifies the simulation time for which the process will wait before awaking to continue processing. An example is shown below:

void MyProcess(SpObjHandle oh) {
    P_VAR;
    P_LV(int, a);
    P_BEGIN(1);
    RB_cout << "Beginning wait" << endl;
    WAIT(1, 5.0);
    RB_cout << "Waited 5 ticks" << endl;
    P_END;
}

• WAIT_FOR(reentryLabel, semaphore, timeOut)
This macro commands a process to exit until the specified logical or counter semaphore is set or until a timeout condition occurs, whichever comes first. However, when waiting for a counter semaphore, users may prefer using the functionally equivalent WAIT_FOR_COUNTER macro for semantic purposes (see below).
The first argument, reentryLabel, is an integer index representing the numeric order in which the process model construct is being used (i.e. the first wait uses 1, the second wait uses 2, etc.).

The second argument, semaphore, is the logical semaphore (see Section 2.13) on which to wait. This semaphore should be a state variable on the local simulation object, which allows other events that ability to set the semaphore.

The third argument, timeOut, represents the timeout condition, which fits into one of three categories:

- If the third argument is negative, no timeout condition exists. In this case, the process will wait indefinitely until the semaphore is set. If the semaphore is already set, SPEEDES will pass through the command without exiting the process.
- If the third argument is zero, SPEEDES will pass through this command without exiting, regardless of whether the semaphore is set.
- If the third argument is positive, it represents a maximum duration of simulation time through which to wait for the semaphore. If the semaphore is already set, SPEEDES will pass through the command without exiting the process. If the semaphore is not set, the process will exit and then return either when the semaphore is set, or when the maximum time duration has elapsed, whichever comes first.

To determine why a process resumed, simply check whether the semaphore was set. If the semaphore was set, the process resumed because the semaphore was set before the maximum duration elapsed. If the semaphore was not set, the maximum duration elapsed before the semaphore was set.

The timeout condition is useful for creating interrupt-driven processes. In this paradigm, the timeout duration represents a normal wait period, during which other events can interrupt the process by means of setting a semaphore. An example is shown below:

```c
void MyProcess(SpLogicalSem& sem) {
    P_VAR;
    P_BEGIN(1);
    WAIT_FOR(1, sem, 3.0);
    if (sem.IsSet()) {
        RB_cout << "Process interrupted" << endl;
    } else {
        RB_cout << "Total wait time elapsed" << endl;
    }
    P_END;
}
```

- **WAIT_FOR_COUNTER**(reentryLabel, semaphore, timeOut)
  This macro is identical in function to the WAIT_FOR macro, except that the semaphore is a counter semaphore (see Section 2.13).

- **WAIT_FOR_RESOURCE**(reentryLabel, semaphore, amount, timeOut, successFlag)
  This macro is identical in function to the WAIT_FOR macro, except that the semaphore is a resource semaphore (see Section 2.13). Available resource semaphores are SpIntSem and SpDoubleSem. This macro commands a process to wait until a specified amount is obtained from a resource semaphore, or until a timeout condition occurs, whichever comes first.
The first argument, reentryLabel, is an integer index representing the numeric order in which the process model construct is being used (i.e. the first wait uses 1, the second wait uses 2, etc.).

The second argument, semaphore, is the resource semaphore on which to wait (by value, not by pointer). This semaphore should be a state variable on the local simulation object (as opposed to a process local variable) so that other events can add to and subtract from the semaphore.

The third argument, amount, is an integer or double representing the amount of resource requested from the semaphore.

The fourth argument, timeOut, represents the timeout condition, which fits into one of three categories:

- If the fourth argument is negative, no timeout condition exists. In this case, the process will wait indefinitely until requested amount is granted from the semaphore. If resources allow, the semaphore will grant the amount immediately, and SPEEDES will pass through the command without exiting the process.
- If the fourth argument is zero, SPEEDES will pass through this command without exiting the process, granting the amount of resource requested only if the semaphore has the amount immediately available.
- If the fourth argument is positive, it represents a maximum amount of simulation time to wait for the semaphore. If the semaphore currently has the requested amount, SPEEDES will pass through the command without exiting, granting the request. If the semaphore does not have the requested amount, the process will exit and then return either when the semaphore grants the amount requested once it gains enough resource to do so, or when the maximum time duration has elapsed, whichever comes first.

The fifth argument is an integer, which is set to 1 if the amount of requested resource was granted and 0 otherwise.

```java
void MyProcess(SpIntSem& sem) {
  P_VAR;
  P_LV(int success);
  P_BEGIN(1);
  WAIT_FOR_RESOURCE(1, sem, 4, 60.0, success);
  if (success) {
    RB_cout << "4 units granted from semaphore" << endl;
  }
  P_END;
}
```

- **WAIT UNTIL(reentryLabel, time)**
  This macro commands a process to exit and then resume at a later simulation time. The first argument, reentryLabel, is an integer index representing the numeric order in which the process model construct is being used (i.e. the first wait uses 1, the second wait uses 2, etc.). The second argument, time, specifies the simulation time at which to re-enter the process. If the value given for time is earlier than the current simulation time, SPEEDES passes through this command without exiting the process. An example is shown below:

```java
void MyProcess(SpObjHandle oh) {
  P_VAR;
  P_LV(SpSimTime, simTime);
  P_BEGIN(2);
```

```
ASK_1_ARG(1, 100.0, oh, GetA, simTime, NULL, 0);
WAIT_UNTIL(2, simTime);
RB_cout << "Waited until " << simTime.GetTime() << endl;
P_END;
```

### 2.12.2 See also

SpEvent 2.9, SpDefineEvent 2.9, and SpProcSem 2.13.

2.13.1 Description

SpProcSem.H provides all of the classes associated with process model semaphores. Semaphores are objects that allow events to either interrupt or regulate the timing of SPEEDES processes (i.e. reentrant events). There are several kinds of semaphores. In fact, although not discussed in this manual, the SPEEDES framework supports creating an endless variety of semaphores by inheriting from a common abstract base class.

All types of semaphores have one basic functionality in common: SPEEDES processes may suspend themselves in association with any kind of semaphore on their simulation object in order to wait to be released from that semaphore. What makes each kind of semaphore unique is how they determine when to release their waiting processes. A process waiting to be released from a semaphore can wait either indefinitely, up to a maximum simulation time duration, or not at all before resuming.

Currently, semaphores break into two basic categories: state semaphores (SpLogicalSem, SpCounterSem, SpInterruptSem, and SpRendezvousSem) and resource semaphores (SpIntSem and SpDoubleSem).

State semaphores have a boolean state: they are always either set or unset. While a state semaphore is unset, all waiting processes must continue to wait (unless their timeout elapses). Once a state semaphore is set, all waiting processes are released simultaneously. Fundamentally, state semaphores differ only in how other events set and unset them.

Resource semaphores have either an integral (SpIntSem) or floating point (SpDoubleSem) amount of resource associated with them. Events can add to or subtract from resource semaphores at any time. Processes wait for resource semaphores by requesting a certain amount from them, and then inserting themselves into a first-in-first-out queue in the semaphore. As resources accumulate in a resource semaphore (by means of events adding to them), processes waiting on that semaphore subtract their requested amounts as increasing resources become available. Processes are released on a first-come-first-serve basis. If a single event adds enough resources, several processes may be released simultaneously.

Semaphores play a central role in allowing process-based simulations to leverage the discrete-event technique for minimizing Central Processing Unit (CPU) processing. They achieve this by providing an interrupt-driven mechanism as a superior alternative to time-step driven processes (such as using a while loop with a wait statement inside of it). Rather than periodically polling to decide whether to take action (requiring unnecessary CPU processing), a well-written process waits for a semaphore to let another event wake it up when and only when action must be taken (minimizing CPU processing). For example:

```c
void MyProcess() {
  P_VAR;
  P_LV(int, var);
  P_BEGIN(1);
  while (SpGetTime() < 1000) {
    WAIT_FOR(1, aSemaphore, -1);
    TakeAction(var); // Action is taken due to being interrupted
    aSemaphore.Unset();
  }
  RB_cout << "Final value for var = " << var << endl;
  P_END;
}
```
In addition, semaphore timeouts provide a semantically simple and easily maintainable form of event cancellation. Using event cancellation directly is often tedious, since developers must keep track of cancel handles for each event that might be canceled, and then manually call SpCancelEvent(SpCancelHandle &) when needed. The process model relieves the developer of both of these burdens by means of the semaphore timeout mechanism. Thus, processes that use semaphore timeouts often provide a simpler design alternative to algorithms that would otherwise need to involve developer-written event cancellation code.

- **SpLogicalSem**
  This is the most basic state semaphore class, having a simple set or unset state. A logical semaphore functions like a boolean variable (more precisely, an integer whose value is either 0 or 1), which can be used in mathematical or logical equations. Once set, all processes that are waiting for them are instantly released. A logical semaphore on the left side of an assignment statement is set to 1 if the right-hand side is non-zero or to 0 if the right-hand side is 0.

  - **SpLogicalSem(int state = 0)**
    The constructor allows for an optional argument specifying the initial state of the semaphore (zero = unset, non-zero = set). Logical semaphores default to the unset state.

  - **int GetState()**
    Returns 1 if the semaphore is set; 0 if unset.

  - **int IsSet()**
    Returns 1 if the semaphore is set, 0 if unset (equivalent function to GetState).

  - **int Set()**
    This method sets the semaphore (i.e. to the logical value 1). It returns 1.

  - **int Toggle()**
    This method toggles the semaphore. That is, if the semaphore is set, this method unsets the semaphore (returning 0), while if the semaphore is unset, this method sets the semaphore (returning 1).

  - **int Unset()**
    This method unsets the semaphore (i.e. to the logical value 0). It returns 0.

  - **operator int()**
    Returns 1 is the semaphore is set, 0 if unset (equivalent function to GetState).

  - **int operator =(int rhs)**
    Sets the semaphore if rhs is non-zero, returning 1, or unsets the semaphore if rhs is 0, returning 0.

- **SpCounterSem**
  This is a state semaphore class that is nearly equivalent in functionality to a logical semaphore. The difference is that, rather than behaving like an integer that only has two possible values (0 and 1), it behaves like an integer that can have any value. It is considered unset when the value is zero, and set when the value is non-zero. Like a logical semaphore, it can be used in mathematical or logical equations and, once set, all processes that are waiting for that semaphore are instantly released.

  A common usage for a counter semaphore is as a counter for some quantity in the simulation, such that when the quantity is non-zero, processes waiting on it are to be released. An example is a counter semaphore representing the number of objects in view of a radar. Processes that handle the radar’s actions wait for this counter semaphore. This minimizes processing because
the radar’s processes are inactive when there are no objects in view without needing to poll to check if objects are in view. By waiting for the counter semaphore to be non-zero, the process automatically activates when one or more objects are in view.

- **SpCounterSem**(int state = 0)
  The constructor allows for an optional argument specifying the initial state of the counter. Otherwise, counter semaphores are initialized as 0 (unset).

- **int GetState()**
  Returns the current value of the counter.

- **int IsSet()**
  Returns the current value of the counter (equivalent function to GetState).

- **int Set(int state)**
  This method sets the semaphore to the value of state and returns that value.

- **int Toggle()**
  This method toggles the semaphore. That is, if the semaphore is set, this method unsets the semaphore to 0 (returning 0). While, if the semaphore is unset, this method sets the semaphore to 1 (returning 1).

- **int Unset()**
  This method unsets the semaphore (i.e. to the logical value 0). It returns 0.

- **operator int()**
  Returns the current value of the counter (equivalent function to GetState).

- **operator =**(int state)
  This method sets the semaphore to the value of state and returns that value (equivalent to Set(state)).

- **int operator ++()**
  Adds one to the counter, returning the newly incremented value for the counter (i.e. pre-increment).

- **int operator ++(int)**
  Adds one to the counter, returning the original value for the counter (i.e. post-increment).

- **int operator --()**
  Subtracts one from the counter, returning the newly decremented value for the counter. Exception: if the counter value is zero or negative, the counter value remains the same and the current value for the counter is returned (i.e. pre-increment).

- **int operator --(int)**
  Subtracts one from the counter, returning the original value for the counter (i.e. before it was decremented). Exception: if the counter value is zero or negative, the counter value remains the same (i.e. post-increment).

- **int operator +=**(int amount)
  Adds the value of amount to the counter and returns the new value for the counter.

- **int operator -=**(int amount)
  Subtracts the value of amount from the counter and returns the new value for the counter. Exception: if the counter value is zero or negative, the counter value remains the same and the current value for the counter is returned.

- **SpInterruptSem,**
- **SpRendezvousSem**
These two classes are used to support keyed interrupts for processes. That is, setting the semaphore is always associated with a typed key, representing the reason for the interrupt. Keys, represented as either integers or strings, are first registered. Then events may only set the semaphore according to registered keys. Keys may be registered and unregistered dynamically. The two possible types for keys (integers or strings) cannot be mixed: once a key is registered (whether it be an integer or a string), all subsequent keys must be the same type or a runtime error is returned. Once a semaphore determines its key type (by whatever key type is first registered), its key type is fixed for the remainder of the semaphore’s instantiation.

Interrupt semaphores differ from rendezvous semaphores in that interrupt semaphores require setting only one key to release waiting processes, while rendezvous semaphores require setting all registered keys prior to releasing waiting processes.

- **int GetInterrupt(int& num)**
  If the interrupt semaphore is currently set, this method sets num to the key with which the semaphore was set and returns 1. If the semaphore is currently unset, or if keys are typed as strings, this method returns 0 (error). This method is normally called by released processes to find out the reason they were released (i.e. interrupted). For the semaphore to be waited for again later, call Unset. Note this is for SpInterruptSem only.

- **int GetInterrupt(int*& str)**
  If the interrupt semaphore is currently set, this method sets str to the key with which the semaphore was set and returns 1. If the semaphore is currently unset, or if keys are typed as integers, this method returns 0 (error). This method is normally called by released processes to find out the reason they were released (i.e. interrupted). For the semaphore to be waited for again later, call Unset. Note, this is for SpInterruptSem only.

- **int GetNumRegistered()**
  Returns the number of keys currently registered.

- **Key_Type GetRegisterType()**
  Returns an enumerated type representing the type of keys registered: KEY_TYPE_NONE (if no keys have been registered yet), KEY_TYPE_INT (if integers keys are registered), or KEY_TYPE_STRING (if string keys are registered).

- **int IsRegistered(char* str)**
  Returns 1 if the string key str is currently registered; otherwise, returns 0.

- **int IsRegistered(int num)**
  Returns 1 if the integer key num is currently registered; otherwise, returns 0.

- **int IsSet(char* str)**
  Returns 1 if the str key is currently set; otherwise, returns 0 (SpRendezvousSem only).

- **int IsSet(int num)**
  Returns 1 if the num key is currently set; otherwise, returns 0 (SpRendezvousSem only).

- **int Register(char* str)**
  Registers str as a string key and returns 1. Exception: if integer keys have been registered, then this does not register str and, instead, returns 0 (error), since string and integer keys cannot be mixed.

- **int Register(int num)**
  Registers num as an integer key and returns 1. Exception: if string keys have been registered, then this does not register num and, instead, returns 0 (error), since integer and string keys cannot be mixed.
– int Set(char* str)
  If the string key str is currently registered, then this sets the semaphore according to the
  string key str and returns 1 (successful). If the semaphore uses integer keys, or if it is using
  string keys but str is not currently registered, then this does not set the semaphore and,
  instead, returns 0 (error).

– int Set(int num)
  If the integer key num is currently registered, then this sets the semaphore according to the
  integer key num and returns 1 (successful). If the semaphore uses string keys, or if it is using
  integer keys but num is not currently registered, then this does not set the semaphore and,
  instead, returns 0 (error).

– int Unset()
  Unsets all registered keys in the semaphore and returns 0 (meaning unset).

– int Unregister(char* str)
  Unregisters str as a string key and returns 1, unless it is not currently registered, in which
  case it returns 0.

– int Unregister(int num)
  Unregisters num as an integer key and returns 1, unless it is not currently registered, in which
  case it returns 0.

– int UnregisterAll()
  Unregisters all currently registered keys and returns 1. After registering according to a
  certain type (integer or string) and then unregistering, users must later register according to
  the original type.

• SpIntSem
  This class represents a resource semaphore that uses an integer quantity for its resource. See
  introduction to this Semaphore section for a description of resource semaphores.

  – SpIntSem(int amount = 0)
    Integer semaphores are constructed with an initial amount specified by amount. SpIntSem
    has a default value of 0.

  – int AddResource(int amount)
    Adds amount to the resource and returns amount.

  – int operator +=(int amount)
    Adds amount to the resource and returns amount.

  – int operator =(int amount)
    Changes the resource amount to amount and returns amount.

  – operator int ()
    Allows integer semaphores to behave as an integer with a value equal to the current amount
    of the resource.

• SpDoubleSem
  This class represents a resource semaphore that uses a floating point (double) quantity for its
  resource. See introduction to this Semaphore section for a description of resource semaphores.

  – SpDoubleSem(double amount = 0)
    Double semaphores may be constructed with an initial amount specified by amount. Sp-
    DoubleSem has a default value of 0.

- **double AddResource(double amount)**
  Adds amount to the resource and returns the new value for resource.

- **double operator +=(double amount)**
  Adds amount to the resource and returns the new value for resource.

- **double operator =(double amount)**
  Changes the resource amount to amount and returns amount.

- **operator double ()**
  Allows double semaphores to behave as a double with a value equal to the current amount of the resource.

2.13.2 Examples

The following is an example of a semaphore (MySem), a process that waits for the semaphore using the timeout mechanism (MyInterruptibleProcess), and an event method that sets or unsets the semaphore (MyInterruptEvent):

```cpp
#include "SpSimObj.H"
#include "SpProcSem.H"
#include "SpDefineSimObj.H"
#include "SpDefineEvent.H"

class MySemSimObj : public SpSimObj {
public:
  void MyInterruptEvent() { MySem.Set(); }  
  void MyInterruptibleProcess();
private:
  SpLogicalSem MySem;
};
DEFINE_SIMOBJ(MySemSimObj, 1, BLOCK);
DEFINE_SIMOBJ_EVENT_0_ARG(MyEvent, MySemSimObj, MyInterruptEvent);
DEFINE_SIMOBJ_EVENT_0_ARG(MyProcess, MySemSimObj, 
                           MyInterruptibleProcess);

void MySemSimObj::MyInterruptibleProcess() {
  P_VAR;
  P_BEGIN(1);
  WAIT_FOR(1, MySem, 3);
  if (MySem.IsSet()) {
    RB_cout << "Process interrupted" << endl;
  } else {
    RB_cout << "Wait time elapsed without interruption" << endl;
  }
  MySem.Unset();
  P_END;
}
```

2.13.3 See also

SpProc 2.12.
2.14 Global Functions for Scheduling Events

2.14.1 Description

SpSchedule.H includes many of the functions that are used to schedule events. All SPEEDES events need a simulation time and a target SpSimObj in order to be scheduled. Some specialized event types, like interactions and handler events, may require additional arguments when scheduled. Most events can also be given an optional variable-length data block when scheduled, which is defined by a character array (the data) and the size of the array (the number of bytes in the block). The purpose of the data block is to allow the scheduler to pass information to the event target as part of the event.

- **void SCHEDULE_CANCEL_EVENT(SpCancelHandle& cancelHandle)**
  Cancels the event that is associated with the supplied cancel handle.

- **SpCancelHandle SCHEDULE_EVENT(const SpSimTime& time,**
  const char* data = NULL,**
  int dataBytes = 0)**
  Reschedules the current event at the specified time, with a new variable-length data block. If the optional data arguments are not used, the variable-length data associated with the current event will not be reused when the event is rescheduled.

- **SpCancelHandle SCHEDULE_EVENT(const SpSimTime& time,**
  const SpMsg& headers,**
  const char* data = NULL,**
  int dataBytes = 0)**
  This function is the same as SCHEDULE_EVENT, above, except that the caller provides the SpMsg message (i.e. headers) instead of letting it be filled out by the function.

- **SpCancelHandle SCHEDULE_EVENT(const SpSimTime& time,**
  int eventType,**
  const SpObjHandle& targetSimObj,**
  const char* data = NULL,**
  int dataBytes = 0)**
  Schedules an event of the provided type, on the specified simulation object, at the desired time. An optional byte array may be appended to the event for the purpose of passing data to the target. This is a low-level function that will not be needed by most users, since the DEFINE_EVENT macros (see Section 2.6) generate their own schedule functions for specific event types.

- **SpCancelHandle SCHEDULE_EVENT(const SpSimTime& time,**
  int eventType,**
  const SpObjHandle& targetSimObj,**
  const SpMsg& headers,**
  const char* data = NULL,**
  int dataBytes = 0)**
  This function is the same as SCHEDULE_EVENT, immediately above, except that the caller provides the header information instead of letting it be filled out by the function. This is a low-level function that will not be needed by most users, since the DEFINE_EVENT macros (see Section 2.6) generate their own schedule functions for specific event types.
2.14 Global Functions for Scheduling Events

- **SpCancelHandle SCHEDULE_HANDLER(const SpSimTime& time, const SpObjHandle& targetSimObj, const char* triggerString = NULL, const char* data = NULL, int dataBytes = 0)**

  Schedules a directed handler event for the target simulation object, at the specified time, with the supplied trigger string. An optional byte array may also be supplied for the purpose of passing data to the target. See Section 2.7 for more information on event handlers and handler events.

- **SpCancelHandle SCHEDULE_HANDLER(const SpSimTime& time, const char* triggerString = NULL, const char* data = NULL, int dataBytes = 0)**

  This function is the same as SCHEDULE_HANDLER, above, except that this function will schedule an undirected handler event, which will be sent to all objects that have an interest in the supplied trigger string. See Section 2.7 for more information on event handlers and handler events.

- **SpCancelHandle SCHEDULE_INTERACTION(const SpSimTime& time, const SpObjHandle& targetSimObj, const char* triggerString, const SpParmSet& paramSet)**

  Schedules a directed interaction for the target simulation object, at the specified time, with the supplied trigger string. Instead of an optional data block, interactions use a mandatory SpParmSet to pass data to the target object. See Section 2.7 for more information on interactions.

- **SpCancelHandle SCHEDULE_INTERACTION(const SpSimTime& time, const char* triggerString, const SpParmSet& paramSet)**

  This function is the same as SCHEDULE_INTERACTION, above, except that this function will schedule an undirected interaction, which will be sent to all objects that have an interest in the supplied trigger string. See Section 2.7 for more information on interactions.

2.14.2 See also

2.15 SpSimObj: Generic Simulation Object Base Class

2.15.1 Description

Simulation objects are the fundamental modeling units in SPEEDES applications on which events (i.e. class SpEvent) act. Simulation objects must be designed independent of other simulation objects to permit each simulation object to temporarily exist at different logical (simulated) times as controlled by the time management algorithm. Simulation objects are permitted to schedule events for other simulation objects.

- Virtual methods that may be overridden by the implementor:

  - virtual void Init()
    Initializes this object. Automatic object managers execute this method during the object creation process.

  - virtual void NamedQuery(double timeTag, SpNameValueList& nameValueList, char* queryName)
    Queries this simulation object for a name-value list of information. Simulation objects can be queried during run-time for real-time updates of their state. The state information queried is defined by the simulation object at compile time by overriding this virtual function. This form of the query passes a query name to allow specific types of queries.

  - virtual void print()
    Prints out data regarding this simulation object.

  - virtual void Query(double timeTag, SpNameValueList& nameValueList)
    Queries this simulation object for a name-value list of information. Simulation objects can be queried during run-time for real-time updates of their state. The state information queried is defined by the simulation object at compile time by overriding this virtual function. Data is returned in the SpNameValueList (see Section 3.25) nameValueList.

  - virtual void SetName(char* n)
    Sets the name of this object to n.

  - virtual void Terminate(double time)
    Virtual method used for operations (such as printing statistics) associated with this simulation object to be performed when the simulation completes. time is the time at which the object was terminated (the end of the simulation).

- void AddComponent(SpComponent* component)
  Adds the component (see Section 3.15) to this simulation object. When a SpComponent is added to or removed from this simulation object using AddComponent or RemoveComponent, the event handlers associated with that SpComponent will automatically be added to or removed from this simulation object as well. This method can be used rollbackably during runtime.

- void AddHandler(const SpHandlerId& handlerId, char* trigger = NULL)
  Adds the handler described by handlerId to this simulation object. If trigger is not NULL, the handler will be called whenever a directed handler is scheduled on this simulation object with the given trigger.

- int CheckSubscription(char* name)
  Checks if a subscription type specified by the input argument name is in the external subscriptions list. Returns 1 if the subscription is found, 0 otherwise.
2.15 SpSimObj: Generic Simulation Object Base Class

- int CheckSubType(char* type, int externalId)
  Checks if an external module (specified by the externalId argument) has subscribed to a specified type (represented as the string type). Returns 1 if the subscription is found, 0 otherwise.

- int event_type(char* name)
  Gets the enumerated id for an event type, specified by the event type’s plugged in string name.

- void* get_manager(int objectType = -2)
  Gets the object manager for a given id.

- RB_queue* GetExtSubscriptions()
  Returns an RB_queue* (see Section 3.9) of external subscriptions. Use CheckSubType to identify if there is a specific subscription to this object.

- int GetGlobalId()
  Gets the unique global id for this simulation object. Each object in a simulation has a unique global id. Global ids are generated automatically but can be overridden by the user. The value given is, in general, not repeatable for different numbers of nodes.

- int GetName()
  Gets the name of this object (for referring to this Simulation object by name).

- int GetNodeId()
  Gets the node on which simulation object is instantiated.

- int GetNumNodes()
  Returns the total number of nodes in the entire simulation.

- int GetNumSimObjsInMgrInCluster()
  Gets the total number of objects of this type in this cluster.

- int GetNumSimObjMgrIds()
  Gets the total number of object manager types in the current simulation.

- RB_SpRandom* GetRandom()
  Returns a pointer to the RB_SpRandom (see Section 3.30) for this simulation object.

- int GetSeed()
  Gets the seed for the random number generator.

- int object_type(char* name)
  Retrieves a simulation object’s integer type based on its plugged in string name.

- int GetSimObjKindId()
  Returns the enumerated id of this object (i.e. the nth overall object of this type). When not using user-defined object managers, this value is automatically set for the user.

- int GetSimObjLocalId()
  Returns the SimObjId or local id of this object on the node.

- int GetSimObjMgrId()
  Gets the numeric id or type for this simulation object. All similar objects (e.g. “missiles” is one type, “cars” is another type, etc.) of the same type are assigned a unique type or id during SPEEDES initialization. This method returns that id.
- **void RemoveComponent(SpComponent* component)**
  Removes the component (see Section 3.15) from this simulation object. When a SpComponent is added to or removed from this simulation object using AddComponent or RemoveComponent, the event handlers associated with that SpComponent will automatically be added to or removed from this simulation object as well. This method can be used rollbackably during runtime.

- **void RemoveHandler(const SpHandlerId& handlerId, char* trigger = NULL)**
  Removes the handler specified by handlerId for the string, trigger.

- **void SubscribeHandler(const SpHandlerId& handlerId, char* trigger = NULL)**
  Associates handler, denoted by handlerId, with given string, trigger. This will invoke this handler whenever an undirected handler is scheduled with the specified trigger.

- **Methods for getting the object handle (see Section 3.26) of a specific object:**
  - ```SpObjHandle GetObjHandle()```  
    Returns the object handle for this object.
  - ```SpObjHandle GetObjHandle(char* mgrName, char* objName)```  
    Returns the object handle for the object with manager name, mgrName, and object name, objName.
  - ```SpObjHandle GetObjHandle(char* mgrName, int kindId)```  
    Returns the object handle for the object with manager name, mgrName, and enumerated id, kindId.
  - ```SpObjHandle GetObjHandle(int kindId)```  
    Returns the object handle for an object of the current object’s type with enumerated id, kindId.
  - ```SpObjHandle GetObjHandle(int mgrId, char* objName)```  
    Returns the object handle for the object of type, mgrId, with name, objName.
  - ```SpObjHandle GetObjHandle(int mgrId, int enumId)```  
    Returns the object handle for the object of type, mgrId, with enumerated id, enumId.

- **void SetNodeId(int node)**
  Sets the node that this simulation object has been instantiated on. Should not be called, except by the object manager constructing this object.

- **void SetNumNodes(int nNodes)**
  Sets the total number of nodes in the entire simulation. Should not be called, except by the object manager constructing this object.

- **void SetNumSimObjsInMgrInCluster(int numSimObjs)**
  Sets the total number of objects in this cluster of the simulation. That is, the sum of the number of each object of each object type on each node.

- **void SetNumSimObjMgrIds(int numMgrIds)**
  Sets the number of simulation object manager ids there are in the simulation. Should not be called, except by the object manager constructing this object.

- **void SetSimObjLocalId(int simObjId)**
  Sets the id of this simulation object. Each object manager has its own set of simulation objects, each with its own simObjId (local id). Thus, the simObjId only needs to be unique for a given object manager on a given node. Should not be called, except by the object manager constructing this object.
2.15 SpSimObj: Generic Simulation Object Base Class

- **void SetSimObjKindId()**
  Sets the enumerated id of this object (i.e. the nth overall object of this type). When not using user-defined object managers, this value is automatically set for the user.

- **void SetSimObjMgrId(mgrId)**
  Sets the enumerated type for this type of simulation object. A unique SimObjMgrId is required for each simulation object class. Each simulation object class is managed by one instantiation of SimObjMgr. Thus, the SimObjMgrId also uniquely defines the type of simulation object that each simulation object manager instantiation manages. Each node has a single instantiation of SpMasterSimObjMgr, which, in turn, manages a list of simulation object managers. SpMasterSimObjMgr refers to each simulation object manager by the SimObjMgrId it manages. Should not be called, except by the object manager constructing this object.

### 2.15.2 See also

SpEvent 2.9 and SpSimObjMgr 2.16
2.16 SpSimObjMgr: Generic Simulation Object Manager

2.16.1 Description

Simulation object managers create and distribute simulation objects of any given type on their nodes. Although the simulation object manager generally does not model any aspects of the simulation, it does inherit from the SpSimObj class and can receive and process events. Simulation developers can write their own manager that inherits from SpSimObjMgr, but in most cases, a simulation object manager can be auto-generated by macros, which alleviates the user from creating an object manager.

- Virtual methods that may be overridden by the implementor:
  - virtual SpSimObj* CreateSimObj()
    Creates a new simulation object of the type supervised by the simulation object manager. Unless this virtual method is overridden, it returns NULL.
  - virtual void define_object(SpSimObj* toBeDefined)
    Takes the supplied simulation object (which must have already been created) and stores a pointer to it in an internal array. A pointer to the object may be subsequently retrieved using the get_obj method.
  - virtual void DestroySimObj(SpSimObj* toBeDestroyed)
    Destroys a simulation object that was created via the CreateSimObj method. Unless this virtual method is overridden, it will do nothing.
  - virtual void get_nodid(int globalIdIn, int& nodeIdOut, int& localIdOut)
    Given the global id of a simulation object of the manager’s id, the object’s node id and local id are returned via the last two arguments.
  - virtual SpSimObj* get_obj(int localId)
    Returns the simulation object supervised by the manager that has the supplied localId. The object must have been passed into a call to the define_object method prior to the invocation of get_obj.
  - virtual void get_oid(int& globalIdOut, int nodeIdIn, int localIdIn)
    Given the node id and local id of a simulation object of the manager’s id, the object’s global id is returned via the first argument.
  - virtual void init_events()
    Does nothing by default. Developers writing their own managers can redefine and use this method to schedule initial events for the manager’s simulation objects. This method is called just before the beginning of the simulation proper.
  - virtual void reserve_n_objects(int numSimObjsOnNode)
    Allocates memory for an internal array of pointers to the simulation objects that the manager will be creating and supervising.

- void get_nodid(char* simObjNameIn, int& nodeIdOut, int& localIdOut)
  Given the name of a simulation object of the manager’s id, the object’s node id and local id are returned via the last two arguments. If the name is not found, nodeIdOut and localIdOut are both set to -1. The set_global_names method should be called prior to invoking this method.

- int GetNumSimObjLocalIds()
  Retrieves the number of simulation objects of the manager’s id that are on the manager’s node.
• **int GetNumSimObjSimObjKindIds()**
  Returns the number of simulation objects of the manager’s id across the entire simulation, as opposed to the number of simulation objects of the manager’s type on its node.

• **SpObjHandle GetObjHandle(const char* simObjName)**
  Returns the handle of a simulation object supervised by the manager that has the supplied name. If the manager does not know of any simulation object with the supplied name, the handle (-1, -1, -1) is returned instead.

• **void print_global_names()**
  The internal database created by set_global_names is printed to cout.

• **void set_global_names()**
  Creates an internal database of the names of all of the simulation objects of the manager’s id in the simulation. This allows for easy name lookups using the get_nodid(char*, int&, int&) method. The internal database is not updated if simulation objects are dynamically created or destroyed.

• **void SetNumSimObjLocalIds()**
  Sets the number of simulation objects of the manager’s id that are on the manager’s node, which it derives from the number of objects of the proper type across the simulation and the number of nodes. The NumSimObjSimObjKindIds variable must have been set prior to calling this method. Developers writing their own managers are responsible for calling this in the manager constructor.

• **void SetNumSimObjLocalIds(int numLocalIds)**
  Sets the number of simulation objects of the manager’s id that are on the manager’s node. Developers writing their own managers are responsible for calling this in the manager constructor.

• **void SetNumSimObjSimObjKindIds(int numKindIds)**
  Sets the number of simulation objects of the manager’s id across the simulation. Developers writing their own managers are responsible for calling this in the manager constructor when objects are created and destroyed.

• Protected Members and Methods
  – **int dealme(int totalNumSimObjs)**
    Given the total number of simulation objects of the manager’s id across the entire simulation, this method will return the number of objects that should be located on the manager’s node assuming an equal distribution. The purpose of this method is to determine how many simulation objects a manager should create.

  – **int dealme(int totalNumSimObjs, int& offset)**
    Given the total number of simulation objects of the manager’s id across the entire simulation, this method will return the number of objects that should be located on the manager’s node assuming an equal distribution. The purpose of this method is to determine how many simulation objects a manager should create. The offset argument is an out parameter whose value can be used to determine the global ids of the simulation objects supervised by the manager.

  – **int NumSimObjLocalIds**
    The number of simulation objects of the manager’s id that are on the manager’s node. Developers writing their own managers can modify this variable directly or use the SetNumSimObjLocalIds methods. This variable should be set in the manager constructor.
– **int NumSimObjSimObjKindIds**
  The number of simulation objects of the manager’s id across the entire simulation, as opposed to the number of simulation objects of the manager’s type on its node. Developers writing their own managers can modify this variable directly or use the SetNumSimObjSimObjKindIds method. This variable should be set in the manager constructor.

### 2.16.2 Bugs

The `set_global_names` method can cause unexpected behavior if multiple clusters are used within the same SPEEDES execution.

### 2.16.3 See also


### 2.16.4 Notes

For the purpose of scheduling events for a manager, its object handle is `(NodeId, SimObjMgrId, -1)`. Developer-written managers that inherit from SpSimObjMgr should set the variables NumSimObjLocalIds and NumSimObjSimObjKindIds, execute methods `reserve_n_objects` and `define_object_methods`, and increment NumSimObjsInMgrInCluster.
2.17 SpSimTime: Simulation Time Abstraction

2.17.1 Description

SpSimTime is composed of a double for representing the physical time and two integer priority fields, which are used for tie-breaking. This ensures repeatability when utilizing multiple nodes and rollbacks are involved. Since rollbacks are not repeatable, when events are rescheduled, there must be a way to order them the same way each time they are reprocessed.

Since physical times are often the same, priorities are needed to indicate the order in which events which have the same physical time are to be processed. The first priority field has precedence over the second priority field and lower values indicate higher priority. The combination of the physical time and the two priority fields is called logical time.

Repeatable, rollbackable simulations are assured only if events always schedule new events in future logical time. In other words, scheduling must either have physical time non-zero lookahead, or physical time zero lookahead with higher priority values (meaning later logical time) than the priorities of the current event.

- Methods for modifying and examining a SpSimTime:
  - SpSimTime(double t = 0.0, int p1 = 0, int p2 = 0, int counter = 0, int uniqueId = 0)
    Default constructor for SpSimTime, initializes all values to 0 currently.
  - SpSimTime(SpSimTime&)
    Copies the constructor.
  - SpSimTime& DecrementPriority1(int value = 1)
    Subtracts a specified amount from the first priority field.
  - SpSimTime& DecrementPriority2(int value = 1)
    Subtracts a specified amount from the second priority field.
  - void DisableAutoTieBreaking(void)
    Ensures that the tie-breaking fields will be set to exactly the same as the current event.
  - int GetPriority1() const
    Gets the first priority field.
  - int GetPriority2() const
    Gets the second priority field.
  - double GetTime()
    Gets the physical time.
  - SpSimTime& IncrementPriority1(int value = 1)
    Adds a specified amount to the first priority field.
  - SpSimTime& IncrementPriority2(int value = 1)
    Adds a specified amount to the second priority field.
  - void print (ostream& out = cout)
    Prints the simulation time to a stream.
  - void SetPriority(int p1)
    Sets the first priority field.
  - void SetPriority(int p1, int p2)
    Sets the first and second priority fields.
- **void SetPriority1(int p)**
  Sets the first priority field.

- **void SetPriority2(int p)**
  Sets the second priority field.

- **void setTime(double tt)**
  Sets the physical time.

- **operator double() const**
  When used as a primitive, SpSimTime automatically converts to a double representing its physical time.

- **SpSimTime& operator = (double tt)**
  When assigned to a double, SpSimTime assigns that double to its physical time without changing the original priority fields.

- **SpSimTime& operator += (double tt)**
  When add-assigned to a double, SpSimTime adds that double to its physical time without changing the original priority fields.

- **SpSimTime& operator -= (double tt)**
  When subtract-assigned to a double, SpSimTime subtracts that double from its physical time without changing the original priority fields.

- **SpSimTime& operator = (const SpSimTime& tt)**
  When assigned to another SpSimTime, this SpSimTime undergoes an element-by-element copy. This method sets the right-hand side priority fields to the same values contained in the SpSimTime on the left-hand side.

- **int operator < (const SpSimTime& tt) const**
  Performs a less-than logical time comparison between this SpSimTime and another SpSimTime. This is a logical time comparison, which means that the physical time is compared and, if they are equal, the first priority fields are compared and, if they are equal, the second priority fields are compared.

- **int operator > (const SpSimTime& tt) const**
  Performs a greater-than logical time comparison between this SpSimTime and another SpSimTime. This is a logical time comparison, which means that the physical time is compared and, if they are equal, the first priority fields are compared and, if they are equal, the second priority fields are compared.

- **int operator <= (const SpSimTime& tt) const**
  Performs a less-than-or-equal logical time comparison between this SpSimTime and another SpSimTime. This is a logical time comparison, which means that the physical time is compared and, if they are equal, the first priority fields are compared and, if they are equal, the second priority fields are compared.

- **int operator >= (const SpSimTime& tt) const**
  Performs a greater-than-or-equal logical time comparison between this SpSimTime and another SpSimTime. This is a logical time comparison, which means that the physical time is compared and, if they are equal, the first priority fields are compared and, if they are equal, the second priority fields are compared.

- **int operator == (const SpSimTime& tt) const**
  Performs an equality logical time comparison between this SpSimTime and another SpSimTime. This is a logical time comparison, which means that the physical time is compared and, if they are equal, the first priority fields are compared and, if they are equal, the second priority fields are compared.
2.17 SpSimTime: Simulation Time Abstraction

- int operator !(const SpSimTime& tt)
  Performs a not-equal logical time comparison between this SpSimTime and another SpSimTime. This is a logical time comparison, which means that the physical time is compared and, if they are equal, the first priority fields are compared and, if they are equal, the second priority fields are compared.

- int operator < (double tt) const
  Performs a less-than comparison between this SpSimTime and a double.

- int operator > (double tt) const
  Performs a greater-than comparison between this SpSimTime and a double.

- int operator <= (double tt) const
  Performs a less-than-or-equal comparison between this SpSimTime and a double.

- int operator >= (double tt) const
  Performs a greater-than-or-equal comparison between this SpSimTime and a double.

- int operator ==(double tt) const
  Performs an equality comparison between this SpSimTime and a double.

- int operator != (double tt) const
  Performs a not-equal comparison between this SpSimTime and a double.

- ostream& operator << (ostream& out, const SpSimTime& tt)
  Prints out the contents of a SpSimTime.

- Additional inline functions:

  - int operator < (double t, const SpSimTime& tt)
    Performs a less-than comparison between a double and a SpSimTime.

  - int operator > (double t, const SpSimTime& tt)
    Performs a greater-than comparison between a double and a SpSimTime.

  - int operator <= (double t, const SpSimTime& tt)
    Performs a less-than-or-equal comparison between a double and a SpSimTime.

  - int operator >= (double t, const SpSimTime& tt)
    Performs a greater-than-or-equal comparison between a double and a SpSimTime.

  - int operator ==(double t, const SpSimTime& tt)
    Performs an equality comparison between a double and a SpSimTime.

  - int operator != (double t, const SpSimTime& tt)
    Performs a not-equal comparison between a double and a SpSimTime.

2.17.2 Examples

When scheduling events, always ensure time is in the future. Also, when incrementing current time, use the += operator, rather than using the + operator. For example:

```c++
/*...*/
void myClass::myMethod() {
    SpSimTime currentTime = SpGetTime();

    // Schedule firstEvent for now.
    SCHEDULE_firstEvent(currentTime, SpGetObjHandle(0));
```
SpSimTime newTime = SpGetTime();

    // Schedule secondEvent 12 seconds in the future
    newTime += 12.0; // Preserves the priority fields
    SCHEDULE_secondEvent(newTime, SpGetObjHandle(2));

    // Ensure thirdEvent occurs after secondEvent
    newTime.IncrementPriority2();
    SCHEDULE_thirdEvent(newTime, SpGetObjHandle(3));
}

2.17.3 See also


2.17.4 Notes

Consider the following code fragment:

    SpSimTime y(1.0, 2, 3, 4, 5);
    SpSimTime z = y + 1.0;
    y += 1.0;

The value of z will end up as (2.0, 0, 0, 0, 0), while y will become (2.0, 2, 3, 4, 5). The reason for the
difference is that there is no + operator for SpSimTime. The assignment to z is performed by changing
y to a double, adding it to 1.0, converting back to a SpSimTime, and, then, assigning to z while the the
change to y occurs through the operator +=.
Chapter 3

General Utilities
3.1 DEFINE_MEMPOOL: Macros for Memory Pool Management

3.1.1 Description

In order to reduce memory fragmentation and thrashing from frequent allocation and deallocation of common classes, SPEEDES uses free lists and memory pools to reduce processing time. One place in particular that this is used is in the management of SpAlt (see Section 2.3) classes.

There are two main macros used for memory pools:

- **DEFINE_MEMPOOL(class, size)**
  Defines a memory pool for the given class with a default allocation number of size. That is, when a new item of type “class” is requested and no more remain in the memory pool, it will create size items and add them to the pool. class must inherit from the class SpMempool, which contains no useful user methods. This macro makes two useful functions:

  - **class* ALLOCATE_class()**
    Returns an item from the memory pool.

  - **void DEALLOCATE_class(class* item)**
    Returns “item” to the memory pool for later use.

- **PLUGIN_MEMPOOL(class)**
  Registers the memory pool with the SPEEDES system. This macro must appear within a compiled .C file in order to use the macros expanded from DEFINE_MEMPOOL.

3.1.2 Examples

An example of using this macro is given in the example of SpAlt (see Section 2.3). Here is another example, using this macro for its general free list capabilities through multiple inheritance:

```c
// Main.C
#include "SpMempool.H"

class MyClass {
    /*...*/
};

class MyClassExtended : public MyClass, public SpMempool{};

// Create 100 at a time.
DEFINE_MEMPOOL(MyClassExtended, 100);
PLUGIN_MEMPOOL(MyClassExtended);

static inline MyClass* GetMyClassFromMempool() {
    return ALLOCATE_MyClassExtended();
}

// Note: returning a class to the mempool that was not obtained
// from the mempool will likely corrupt memory
static inline void ReturnMyClassToMempool(MyClass* usedItem) {
    DEALLOCATE_MyClassExtended((MyClassExtended*)usedItem);
}
3.1 DEFINE_MEMPOOL: Macros for Memory Pool Management

```c
int main (int argc, char** argv) {
    /*...*/
}
```

### 3.1.3 Bugs

Memory pools never shrink in size. They also do not return their allocated items to the heap upon program exit.

### 3.1.4 Used in

SpAlt 2.3.
3.2 **C_ITEM: Base Class for Items Within a C_QUEUE (obsolete)**

3.2.1 **Description**

Classes must inherit from this class to be inserted into C_QUEUEs. This class does not have any rollback support and is simply a generic queue item that can be sorted according to simulated time tags by a C_QUEUE. These classes can also be used in a C_QUEUE used as a stack. That is, popping must always be from the top of the queue (since it is singly linked) but pushing can be to the top or the bottom of the queue. When consistently pushing to the top, the C_QUEUE acts as a stack. When consistently pushing to the bottom, the C_QUEUE acts as a queue. Sorting and inserting operations always put the low time value on top so that popping off the sorted queue or stack occurs in simulated time order.

This class is also used as a base type for many of the other structures within SPEEDES. For example, it is used in the class, C_HASH (a hash table of C_QUEUEs), and in the SPEEDES event queue.

- **virtual ~C_ITEM()**
  Virtual destructor.

- **virtual const char* GetName()**
  User-overridable method to get the name for a class which inherits from C_ITEM. Default prints an error message and returns NULL.

- **C_ITEM* get_link()**
  Gets the link for this C_ITEM (i.e. the link in the singly linked data structure C_QUEUE).

- **SpSimTime& GetQitemTimeTag()**
  Returns the simulation time tag for this C_ITEM.

- **void set_link(C_ITEM* link)**
  Sets the link for this C_ITEM (i.e. the link in the singly linked C_QUEUE data structure).

- **void SetQitemId(int i)**
  Sets the id of this C_ITEM (used in C_hash).

- **void SetQitemTimeTag(const SpSimTime& t)**
  Sets the simulation time tag for this C_ITEM.

3.2.2 **See also**

SpEvent 2.9, RB_hash 3.7, and RB_queue 3.9.
3.3 NET_INT and NET_DOUBLE: Network Independent Data Types

3.3.1 Description

NET_INT and NET_DOUBLE are two classes used in SPEEDES in order to support heterogeneous computing between Institute of Electrical and Electronics Engineers (IEEE) machines using the same word size but different endian natures. Conversions are automatically made to the correct endian nature without requiring any user intervention. When running on a homogeneous environment, these are typeeffed to int and double resulting in no performance penalty.

In order to compile SPEEDES to operate in a heterogeneous environment, compile using the command line “make HETEROGENEOUS=”, rebuilding all files that depend on the SPEEDES baseline.

Used in


Notes

SpNetMode.H is used to determine whether SPEEDES is compiled in a homogeneous or heterogeneous mode. Heterogeneous mode carries the penalty of using larger data structures along with more time wasted in converting data types. Choose an appropriate SpNetMode.H (both heterogeneous and homogeneous versions are provided) for your situation.
3.4 RB_double: Rollbackable Double

3.4.1 Description

RB_double is a rollbackable floating point class. Use this class to contain any floating point state information for simulation objects.

- **RB_double()**
  Default constructor initializes value to 0.0.

- **RB_double(double value)**
  Constructor from double.

- **double operator ++()**
  Prefix increment operator (adds 1.0).

- **double operator ++(int)**
  Postfix increment operator (adds 1.0).

- **double operator --()**
  Prefix decrement operator (subtracts 1.0).

- **double operator --(int)**
  Postfix increment operator (subtracts 1.0).

- **double operator =(const RB_double& value)**
  Assignment operator from RB_double.

- **double operator =(double value)**
  Assignment operator from double.

- **double operator +=(double delta)**
  Assignment addition operator.

- **double operator -==(double delta)**
  Assignment subtraction operator.

- **double operator *==(double factor)**
  Assignment multiplication operator.

- **double operator /==(double factor)**
  Assignment division operator.

3.4.2 See also

RB_int 3.6.
3.5 **RB\_exostream: Rollbackable External Ostream**

3.5.1 **Description**

RB\_exostream functions like other C++ ostremms, except that it is rollbackable and that the data streamed into it is sent outside of SPEEDES to any external modules that are listening. The SpSortedOutput command line utility is an example of an external module that can receive data sent by a RB\_exostream. An instance of the RB\_exostream class is created with a string which is the stream name. Data inserted into the exostream via its $<<$ operators will be converted to a string and sent through the HostRouter to any external module processes that were started with the same stream name. This is done in a rollbackable manner, so that nothing will happen if the event that sent data to the exostream is rolled back. The data sent to the external module processes is time-tagged, which allows the data to be displayed in time order. By having all nodes stream data to a single SpSortedOutput process, you can get a very good picture of the sequence of events across the entire simulation. Data inserted into an instance of RB\_exostream will not actually be sent out across the network until a flush, endl, or ends stream manipulator is inserted into to the exostream.

- **RB\_exostream(char* streamName)**
  Creates an instance of the rollbackable external ostream class. The name of this exostream is streamName, which is used by external modules.

- **RB\_exostream& operator $<<(char data)**
  Inserts char data into the stream.

- **RB\_exostream& operator $<<(char* data)**
  Inserts a string into the stream.

- **RB\_exostream& operator $<<(const char* data)**
  Inserts const char* data into the stream.

- **RB\_exostream& operator $<<(const SpSimTime data)**
  Inserts SpSimTime data into the stream.

- **RB\_exostream& operator $<<(double data)**
  Inserts double data into the stream.

- **RB\_exostream& operator $<<(float data)**
  Inserts float data into the stream.

- **RB\_exostream& operator $<<(int data)**
  Inserts int data into the stream.

- **RB\_exostream& operator $<<(long data)**
  Inserts long data into the stream.

- **RB\_exostream& operator $<<(short data)**
  Inserts short data into the stream.

- **RB\_exostream& operator $<<(const RB\_SpBool data)**
  Inserts RB\_SpBool data into the stream.

- **RB\_exostream& operator $<<(const SpBool data)**
  Inserts SpBool data into the stream.
- **RB\_exostream\& operator \(\llbracket\) (SpOmanip data)**
  Inserts manipulator into the stream data. SpOmanip is a typedef of ostream\& (* SpOmanip ) ( ostream\& ). Examples of stream manipulators are flush, endl, and ends.

- **RB\_exostream\& operator \(\llbracket\) (unsigned char data)**
  Inserts unsigned char data into the stream.

- **RB\_exostream\& operator \(\llbracket\) (unsigned char* data)**
  Inserts unsigned char* data into the stream.

- **RB\_exostream\& operator \(\llbracket\) (unsigned int data)**
  Inserts unsigned int data into the stream.

- **RB\_exostream\& operator \(\llbracket\) (unsigned long data)**
  Inserts unsigned long data into the stream.

- **RB\_exostream\& operator \(\llbracket\) (unsigned short data)**
  Inserts unsigned short data into the stream.

- **RB\_exostream\& operator \(\llbracket\) (void* data)**
  Inserts void* data into the stream.

### 3.5.2 Notes

As with all rollbackable types, RB\_exostreams should not be created on the stack.
3.6 RB_int: Rollbackable Integer

3.6.1 Description

This class mimics the int class in almost every way and should be used whenever int type state is necessary.

- **RB_int()**
  Default constructor initializes the value to 0.

- **RB_int(int value)**
  Constructor from an int.

- **operator int() const**
  Integer conversion operator.

- **int operator ++()**
  Postfix increment operator.

- **int operator ++(int)**
  Prefix increment operator.

- **int operator --()**
  Postfix decrement operator.

- **int operator --(int)**
  Prefix decrement operator.

- **int operator =(int value)**
  Assignment operator from int.

- **int operator ==(int i) const**
  Equality operator.

- **int operator !=(int i) const**
  Inequality operator.

- **int operator +=(int delta)**
  Assignment addition operator.

- **int operator -==(int delta)**
  Assignment subtraction operator.

- **int operator *==(int factor)**
  Assignment multiplication operator.

- **int operator /==(int factor)**
  Assignment division operator.

- **int operator %==(int value)**
  Assignment remainder operator.

- **int operator ^==(int value)**
  Assignment not operator.
• `int operator &=(int value)`
  Assignment and operator.

• `int operator |=(int value)`
  Assignment or operator.

• `int operator >>==(int value)`
  Assignment right shift operator.

• `int operator <<==(int value)`
  Assignment left shift operator.

3.6.2 See also

RB_double 3.4 and RB_SpBool 3.14.

3.6.3 Notes

The American National Standards Institute (ANSI) standard for C++ states that only integer types may be used to index an array. This definition does not allow types which can be promoted or demoted to integer types to be used. This includes the RB_int. Therefore, to use an RB_int to index an array, assign it to a normal int first and use that normal int to index the array.
3.7 **RB_hash: Rollbackable Hash of Lists (obsolete)**

### 3.7.1 Description

RB\_hash is a hash table of RB\_queues (see Section 3.9). It provides much faster searches compared to the RB\_queue at the expense of a larger memory usage. Similar to the RB\_queue, all the modifier methods are not rollbackable, but functions in RB\_SpContainerFuncs.H (see Section 3.10) can make rollbackable changes to the RB\_hash class.

- **RB\_hash(int n = 100)**
  Default constructor; creates an RB\_hash with the given number of RB\_queues.
- **void add(C\_SQ\_ITEM* item)**
  Pushes specified item to the bottom of the appropriate queue in the hash array. It is hashed and inserted based on C\_SQ\_ITEM::GetQitemId.
- **void ClearAndDestroy()**
  Clears this hash table and deletes all of the items.
- **void combine()**
  Concatenates each queue in this hash array into one queue, leaving the original hash table intact. The resulting single queue can then be retrieved with the method, get\_all.
- **C\_SQ\_ITEM* find(const char* name)**
  Finds the item from the appropriate queue in the hash array. Returns the item if found, NULL otherwise.
- **C\_SQ\_ITEM* find(int hid)**
  Finds the item from the appropriate queue in the hash array with the given hash id. Returns the item if found, NULL otherwise.
- **RB\_queue* get\_all()**
  Gets the combined queue generated by the last call to combine.
- **int get\_length()**
  Gets the number of C\_SQ\_ITEMs in this hash table.
- **int get\_size()**
  Gets the number of queues in this hash array.
- **int get\_hashid(const char* name)**
  Passes in the name that is used for hashing and returns the hash id that is used for the array index.
- **RB\_queue* GetQ(int hashid = 0)**
  Gets the RB\_queue for the given hashid.
- **void remove(C\_SQ\_ITEM* item)**
  Removes (but does not delete) the specified item from this hash table.
- **C\_SQ\_ITEM* remove(int id)**
  Removes (but does not delete) an item from this hash table specified bid. Returns the item if found, NULL otherwise.
- **void reset()**
  Removes (but does not delete) every item in the hash table.
3.7.2 See also

C\_SQ\_ITEM 3.35 and RB\_queue 3.9.
3.8 RB::ostream: Rollbackable Stream

3.8.1 Description

RB::ostream is a rollbackable ostream and can be used as you would a normal ostream when that ostream is part of the state of a SpSimObj or a non-SpSimObj object (i.e. when it needs to be rollbackable). RB::cout and RB::cerr are two standard streams that have been defined for use in simulation objects in the header file.

- **RB::ostream(char* filename)**
  Creates a rollbackable ofstream to the file, filename.

- **RB::ostream(ostream* ost)**
  Creates a rollbackable ostream from another ostream ost.

- **RB::ostream& operator <<(char data)**
  Writes the char data to the stream.

- **RB::ostream& operator <<(char* data)**
  Writes the char* data to the stream.

- **RB::ostream& operator <<(const char* data)**
  Writes the const char* data to the stream.

- **RB::ostream& operator <<(double data)**
  Writes the double data to the stream.

- **RB::ostream& operator <<(float data)**
  Writes the float data to the stream.

- **RB::ostream& operator <<(int data)**
  Writes the int data to the stream.

- **RB::ostream& operator <<(long data)**
  Writes the long data to the stream.

- **RB::ostream& operator <<(short data)**
  Writes the short data to the stream.

- **RB::ostream& operator <<(SpBool data)**
  Writes the SpBool to the stream.

- **RB::ostream& operator <<(SpOmanip data)**
  Writes the manipulator to the stream data. SpOmanip is a zero argument manipulator and is a typedef of ostream& (* SpOmanip ) ( ostream& ).

- **RB::ostream& operator <<(SpSimTime data)**
  Writes the SpSimTime to the stream.

- **RB::ostream& operator <<(unsigned char data)**
  Writes the unsigned char data to the stream.

- **RB::ostream& operator <<(unsigned char* data)**
  Writes the unsigned char* data to the stream.
- **RB\_ostream& operator \(<\>(\text{unsigned int data})\)**  
  Writes the unsigned int data to the stream.

- **RB\_ostream& operator \(<\>(\text{unsigned long data})\)**  
  Writes the unsigned long data to the stream.

- **RB\_ostream& operator \(<\>(\text{unsigned short data})\)**  
  Writes the unsigned short data to the stream.

- **RB\_ostream& operator \(<\>(\text{void* data})\)**  
  Writes the void* data to the stream.

### 3.8.2 Notes

Because the \(<\>\) operator does not actually write the data to the stream until the event being processed is committed, creating an RB\_ostream from a strstream will result in unexpected behavior. This is the desired behavior for a file or terminal stream, but may not be desired for a strstream.
3.9 RB_queue: Rollbackable Doubly Linked List (obsolete)

3.9.1 Description

This class is simply a rollbackable generic queue that sorts C_SQ_ITEMs according to simulated time tags. When consistently popping from the top of the queue, it can be used as a stack. Pushing can be to the top or the bottom of the queue. When consistently pushing to the top, the RB_queue acts as a stack. When consistently pushing to the bottom, the RB_queue acts as a queue. Pushing effects are vice-versa when consistently popping from the bottom (rather than the top) of the queue (for RB queues or C_DBL_QUEUEs only). Sorting and inserting operations always put the low time value on top so that consecutive pops off the top of the sorted queue or stack occur in simulated time order.

Unlike the newer container classes (RB_SpList, RB_SpBinaryTree, etc.), this class only accepts items that inherit from the base class C_SQ_ITEM.

The modification methods are not rollbackable, but rollbackable modification methods can be found in Section 3.10.

- **void ClearAndDestroy()**
  
  Clears this queue and deletes all of the items.

- **C_SQ_ITEM* find(int id)**
  
  Finds (without deleting) an item from the queue.

- **int get_length()**
  
  Gets the number of items in the queue.

- **C_SQ_ITEM* get_bot()**
  
  Gets a pointer to the item at the bottom of the queue without removing it from the queue.

- **C_SQ_ITEM* get_top()**
  
  Gets a pointer to the item at the top of the queue without removing it from the queue.

- **void join(RB_queue* q)**
  
  Concatenates a specified queue to this queue, but does not reset the specified queue.

- **C_SQ_ITEM* pop_bottom()**
  
  Pops a specified item off of (i.e. removes a specified item from) the bottom of the queue and returns it.

- **C_SQ_ITEM* pop_top()**
  
  Pops a specified item off of (i.e. removes a specified item from) the top of the queue and returns it.

- **void push_bot(C_SQ_ITEM* x_item)**
  
  Pushes a specified item onto (i.e. inserts a specified item at) the bottom of the queue.

- **void push_top(C_SQ_ITEM* x_item)**
  
  Pushes a specified item onto (i.e. inserts a specified item at) the top of the queue.

- **void remove(C_SQ_ITEM* x_item)**
  
  Removes (but does not delete) an item from the queue.

- **void reset()**
  
  Clears this queue without deleting the items.
3.9.2 See also

C\_SQ\_ITEM 3.35 and RB\_SpContainerFuncs.H 3.10.
3.10 RB_SpContainerFuncs.H: Rollbackable Functions for RB_hash and RB_queue (obsolete)

3.10.1 Description

Since the RB_hash (see Section 3.7) and RB_queue (see Section 3.9) modifier methods are not rollbackable, this header file provides methods for rollbackably modifying the state of either of these classes.

- **void RB_INSERT(RB_hash* hash, C_SQ_ITEM* item)**
  Inserts item into the hash. It is hashed on C_SQ_ITEM::GetQitemId.

- **C_SQ_ITEM* RB_POP_TOP(RB_queue* queue)**
  Removes the top item from the queue and returns it.

- **C_SQ_ITEM* RB_POP_BOT(RB_queue* queue)**
  Removes the bottom item from the queue and returns it.

- **void RB_PUSH_BOT(RB_queue* queue, C_SQ_ITEM* item)**
  Rollbackably pushes item onto the bottom of the queue.

- **void RB_PUSH_TOP(RB_queue* queue, C_SQ_ITEM* item)**
  Rollbackably pushes item onto the top of the queue.

- **C_SQ_ITEM* RB_REMOVE(RB_queue* queue, C_SQ_ITEM* item)**
  Locates and removes item within the queue. If item is not found, this function returns NULL.

- **C_SQ_ITEM* RB_REMOVE(RB_hash* hash, C_SQ_ITEM* item)**
  Removes item from the hash and returns it. If item is not found, this function returns NULL.

- **C_SQ_ITEM* RB_REMOVE(RB_hash* hash, int index)**
  Finds, removes, and returns the item with C_SQ_ITEM::GetQitemId equal to index. If not found, returns NULL.

- **C_SQ_ITEM* RB_REMOVE(RB_hash* hash, char* name)**
  Finds, removes, and returns the item with name equal to index. If not found, this function returns NULL.

- **C_SQ_ITEM* RB_REMOVE(RB_queue* queue, int index)**
  Finds the item within the queue with C_SQ_ITEM::GetQitemId equal to index and returns it if found. Otherwise, this function returns NULL.

3.10.2 See also

C_SQ_ITEM 3.35, RB_queue 3.9, and RB_hash 3.7.
3.11 RB_SpString: Rollbackable String

3.11.1 Description

This class is used like a normal character pointer is used on the right-hand side of equations when that character pointer is part of the state of a SpSimObj or non-SpSimObj object (i.e. when it needs to be rollbackable). In addition to regular use on the right-hand side of equations as a character pointer, this object can be set equal to another character pointer using the “=” operator, and the string that the other character pointer points to will be rollbackably copied to this character pointer. Memory management is handled automatically. The “=” operator is the only left-hand side operator supported.

- **RB_SpString()**
  Default constructor initializes to NULL string.

- **RB_SpString(const char* value)**
  Constructor from a char* string. Makes a copy of value and stores it inside the class.

- **` RB_SpString()`**
  Destructor frees any memory that was created in the class.

- **operator const char*() const**
  Returns a pointer to the string contained within. Changing values within this string is not rollbackable. Do not delete the string returned by this operator.

- **const char* operator =(const char* value)**
  Assignment operator that copies the string from value and stores it in the class in a rollbackable fashion. Returns const char* so that the assignment can be chained or used in another function or method.

3.11.2 Examples

Since the RB_SpString does its own memory management, assignments can be made to the RB_SpString from a variety of sources:

```c
#include "RB_SpString.H"
#include "SpSimObj.H"

class S_MySimObj : public SpSimObj {
public:
  void Method1() {
    SimobjType = SpGetMsgData(); // Copies out of current message data
  }
  void Method2(char* string = NULL) {
    SimobjType = string; // Copies out the value from string and
    // assignments from NULL are acceptable
  }
  void Method3() {
    char* localString = "Some String";
    /*...*/
    SimobjType = localString; // Copies out of local storage
  }
private:
```
3.11 RB_SpString: Rollbackable String

    RB_SpString SimobjType;
    }

3.11.3 Notes

The copy constructor is private and the string maintains its own storage for the char* data.
3.12 RB_voidPtr: Rollbackable Void Pointer

3.12.1 Description

Use this as a normal void or character pointer is used on the right-hand side of equations when that pointer is part of the state of a SpSimObj or non-SpSimObj object (i.e. when it needs to be rollbackable). In addition to regular use on the right-hand side of equations as a void or character pointer, this object can be rollbackably set equal to another void pointer using the “=” operator. The “=” operator is the only left-hand side of equations operator supported.

- **RB_voidPtr(void* value = NULL)**
  Default constructor initializes to NULL.

- **operator void*()**
  A void* conversion operator for use on the right-hand side of equations.

- **void* operator =(void* value)**
  Rollbackably assigns value to this class. No data is copied, just the pointer is stored rollbackably.

3.12.2 Examples

The RB_voidPtr saves changes to its value rollbackably. For example, suppose the pointer is pointing at A and then, after the event is processed, the pointer is pointing at B. If the event is rolled back, the pointer will be restored to the A value. However, a common mistake to to assume that RB_voidPtr makes the memory pointed to rollbackable. Consider the following example:

```c++
#include "RB_voidPtr"
#include "SpSimObj.H"
#include "RB_SpDefineClass.H"

RB_DEFINE_CLASS(int);

class S_MySimObj : public SpSimObj {
  public:
    void BadMethod(int numNewValues, int* newValues) {
      int* currValue = (int*)(void*)CargoAmounts;
      for (int i = 0; i < numNewValues; ++i) {
        // Memory pointed to is not rollbackable!!! This is bad!!!!
        currValue[i] = newValues[i];
      }
    }

    void GoodMethod(int numNewValues, int* newValues) {
      // First, delete the old values, create a new memory location
      // and then copy the values into the new memory location.
      RB_DELETE_ARRAY_int((int*) (void*)CargoAmounts);
      CargoAmounts = RB_NEW_ARRAY_int(numNewValues);
      int* currValue = (int*) (void*)CargoAmounts;
      for (int i = 0; i < numNewValues; ++i) {
        currValue[i] = newValues[i];
      }
    }
  private:
```
3.12 RB_voidPtr: Rollbackable Void Pointer

```
    RB_voidPtr CargoAmounts;
```

3.12.3 Used in

RB_SpDynPtrArray 3.18.

3.12.4 Notes

Some compilers do not allow the cast of a RB_voidPtr directly to a pointer of another type. To work around this issue, first cast to a (void*) and then cast to the other class.
3.13 SpBinaryTree and RB_SpBinaryTree: Tree Container Classes

3.13.1 Description

The RB_SpBinaryTree and the SpBinaryTree are two tree container classes which support a general SPEEDES container class API with additional tree specific calls. In general, a well-balanced tree will provide $O(N \log(N))$ inserts, removals, and searches and these trees support several modes which can affect this performance.

Rather than using templates, these trees contain void* values. This means that users are responsible for memory management of items inserted or removed from these trees.

- Insertion. Items of any type can be inserted into the tree. Simply pass the address and the key. Duplicates are allowed and are then found in the order in which they are inserted. That is, if one inserts two items with the same key, the first one inserted will be found first by Find or Remove, etc. The trees are sorted from least key to greatest using either the ordering of numbers for numeric keys or the ordering imposed by the library function, strcmp, for string keys.

Since there is not a standard ordering for the set of strings and doubles, it is not allowed to mix two types of inserts. The SpBinaryTree and RB_SpBinaryTree do not allow inserting using doubles and switching to using strings (or vice-versa) on a non-empty tree. If, however, a tree is filled using double keys and then emptied, that tree can then be used with string keys or vice-versa.

- void Insert(void* item, char* keyName)
  Inserts item using the key, keyName. keyName is then copied and stored internally with the item in the tree. Do not insert NULL.

- void Insert(void* item, double keyNum)
  Inserts item using the key, keyNum. Use of integer keys is handled by this call, as well, through C++’s type promotion from integer to double. Do not insert NULL.

- Removal. The following methods provide several options for removing items from a tree. One can remove a specific item by passing in a key. The item is returned upon removal. If the item is not found, NULL is returned.

- void* Remove(char* keyName)
  Removes and returns the first item inserted and still in the tree with the key, keyName.

- void* Remove(double keyNum)
  Removes and returns the first item inserted and still in the tree with the key, keyNum.

- void* RemoveFirstElement()
  Removes and returns the first item in the tree. If the tree is empty, returns NULL. If the key (or keyName) is desired, retrieve this value prior to removal by calling “GetFirstElement; GetCurrentKey<Name: >” first.

- Find an item in the tree with a given key. If multiple items are in the tree with the key being sought, then the item that was inserted first will be returned (subsequent items can be found by calling FindIterator and then GetNextElement on the returned iterator). If a requested element cannot be found, NULL is returned.

- void* Find(const char* keyName)
  Finds and returns an element with key, keyName.
3.13 SpBinaryTree and RB_SpBinaryTree: Tree Container Classes

- **void** Find(const double keyNum)
  Finds and returns an element with key, keyNum.

- **SpIterator** SpBinaryTree FindIterator(const char* keyName)
  Returns a SpIterator SpBinaryTree pointing to the key, keyName.

- **SpIterator** RB_SpBinaryTree FindIterator(const char* keyName)
  Returns a SpIterator RB_SpBinaryTree pointing to the key, keyName.

- **SpIterator** SpBinaryTree FindIterator(double keyNum)
  Returns a SpIterator SpBinaryTree pointing to the key, keyNum.

- **SpIterator** RB_SpBinaryTree FindIterator(double keyNum)
  Returns a SpIterator RB_SpBinaryTree pointing to the key, keyNum.

- **int** GetNumElements()
  Returns the number of elements in the tree.

- There are many methods for traversing a tree and examining all of its elements. The iterator methods are affected by manipulating the contents of a tree, so do not call GetNextElement or GetPreviousElement after changing the tree or before a call to GetFirstElement or GetLastElement. Because of this, the methods below are deprecated and SpIterator RB_SpBinaryTree and SpIterator SpBinaryTree are preferred. Section 3.23 discusses iterators in more detail.

  - **void** GetCurrentElement()
    Returns the current element of the tree.

  - **void** GetFirstElement()
    Returns the first element in the tree and sets the built-in iterator to point to it. If the tree is empty, returns NULL.

  - **void** GetLastElement()
    Returns the last element in the tree and sets the built-in iterator to point to it. If the tree is empty, returns NULL.

  - **void** GetNextElement()
    Advances to the next element in the tree and returns it. If the end of the tree has been reached, returns NULL.

  - **void** GetPreviousElement()
    Moves back to the previous element and returns it. If the start of the tree has been reached, returns NULL.

  - **void** operator ++()
    Advances to the next element in the tree and returns it. If the end of the tree has been reached, returns NULL. This operator is the same as GetNextElement.

  - **void** operator ++(int)
    Returns the current element and advances the built-in iterator to next element in the container.

  - **void** operator --()
    Moves back to the previous element and returns it. If the start of the tree has been reached, returns NULL. This operator is the same as GetPreviousElement.

  - **void** operator --(int)
    Returns the current element and moves the built-in iterator back to the previous element in the container.
– **double GetCurrentKey()**
  Returns the key of the current element for a tree using double keys.

– **char* GetCurrentKeyName()**
  Returns the key of the current element for a tree using string keys. Do not delete the string returned by this call.

• The **RB_SpBinaryTree** and the **SpBinaryTree** support several modes of operation relating to the way the tree is balanced.

  – **NormalTreeMode** does not attempt to balance the tree (default mode). If data is to be inserted or removed from the tree in a completely random order, this method will perform very well.

  – **BalancedTreeMode** does not keep the tree perfectly balanced but uses a heuristic to balance the tree whenever it becomes too unbalanced. This will help maintain a relatively well-balanced tree if the data is well ordered when it is inserted into the tree or if the data is always removed from one end of the tree.

  – **SplayTreeMode** implements a splay tree algorithm. Whenever an item is inserted into the tree, that item is rotated to the root of the tree. Similarly, when an item is removed, it is first rotated to the root of the tree and then removed. Finally, whenever an item is found in the tree through one of the Find calls, it is rotated to the root of the tree. This algorithm has a similar effect as does a “caching” algorithm. Whenever an item is accessed, the items close to it are moved into a position where they can more quickly be accessed as well. If the items in the tree are inserted, removed, or accessed in a completely random fashion, this mode will yield poor performance.

• The methods for changing and examining the balancing heuristic are:

  – **void SetBalancedTreeMode()**
    Sets the trees to balanced tree mode.

  – **void SetNormalTreeMode()**
    Sets the trees to normal tree mode.

  – **void SetSplayTreeMode()**
    Sets the trees to splay tree mode.

  – **int IsBalancedTreeMode()**
    Returns 1 if the trees are in balanced tree mode, 0 otherwise.

  – **int IsNormalTreeMode()**
    Returns 1 if the trees are in normal tree mode, 0 otherwise.

  – **int IsSplayTreeMode()**
    Returns 1 if the trees are in splay tree mode, 0 otherwise.

### 3.13.2 Bugs

Free lists are used to allocate the tree items and their keys. These free lists do not clean themselves up at exit, leading to what appears to be a memory leak.
3.13 SpBinaryTree and RB_SpBinaryTree: Tree Container Classes

3.13.3 See also


3.13.4 Used in

RB_SpHashTree 3.20, SpHashTree 3.20, SpPriorityTree 3.29, and RB_SpPriorityTree 3.29.
3.14 SpBool and RB_SpBool: Boolean Class

3.14.1 Description

Both the SpBool and RB_SpBool classes have the same interface, but their behaviors differ. The RB_SpBool is rollbackable while the SpBool is not. The following is a look at the SpBool interface. Note that the RB_SpBool is identical.

- enum SpBoolState { SpFALSE, SpTRUE }
  Possible state values.

- SpBool(int s)
  Initializes state to SpFALSE if (s == 0), SpTRUE otherwise.

- SpBool()
  Default constructor, state initialized to SpFALSE.

- SpBoolState IsTrue() const
  Returns SpTRUE if state is non-zero.

- SpBoolState IsFalse() const
  Returns SpFALSE if state is non-zero.

- SpBool operator =(SpBool)
  Assignment operator.

- SpBool operator == (SpBool)
  Equals operator.

- SpBool operator !=(SpBool)
  Not equals operator.

- SpBool operator !()
  Negation operator.

3.14.2 See also

RB_ostream 3.8 and RB_exostream 3.5.

3.14.3 Notes

Because some C++ compilers do not provide the ANSI standard bool class, this class is provided in order to provide similar functionality.
3.15 SpComponent: Generic Reusable Object for SpSimObjs

3.15.1 Description

Components in SPEEDES are subobjects that model states like a simulation object, but instead of standing on their own, they only function when “plugged in” to a SpSimObj. Because each component’s functionality is separate from the simulation objects that it plugs into, the SpComponent is suited for modeling systems that can be found on unrelated types of simulation objects. For example, a radar component could be useful for a wide variety of vehicles in a military simulation. A radar model could be designed as a component which can be plugged into an Airborne Warning and Control System (AWACS) airplane and a ground based radar installation. As requirements or models change, different radar components can be plugged in. Adding and removing components from simulation objects occurs in a rollbackable fashion and can happen at any time during the course of the simulation.

Components interact with the simulation through event handlers. A component can add and remove event handlers for directed and undirected handler events just like a simulation object. When the component is plugged into an object, its handlers are added to the objects. Similarly, when the component is removed, its handlers are removed as well.

- **void AddHandler(const SpHandlerId& handlerId, char* triggerString = NULL)**
  Adds the handler described by handlerId to this component. If the triggerString is not NULL, the handler will be called whenever a directed handler event is scheduled on this component’s simulation object with the given trigger string.

- **int PluggedIn()**
  Returns whether or not this component is currently plugged into a SpSimObj.

- **void RemoveHandler(const SpHandlerId& handlerId, char* triggerString = NULL)**
  Removes the event handler specified by handlerId for the given trigger string.

- **void SubscribeHandler(const SpHandlerId& handlerId, char* triggerString = NULL)**
  Adds the handler described by handlerId to this component. If the triggerString is not NULL, the handler will be called whenever an undirected handler event is scheduled with the given trigger string.

3.15.2 See also

SpSimObj 2.15 and SpDefineHandler.H 2.6.
3.16 SpConvert.H: Time and Space Conversion Functions

3.16.1 Description

- Vector triplet operations.
  - `void CrossProduct(double x1[3], double x2[3], double x3[3])`
    Computes the cross product of x1 and x2, returning it in x3.
  - `void Difference(double x1[3], double x2[3], double x3[3])`
    Computes x1 - x2 and returns the result in x3.
  - `double DotProduct(double x1[3], double x2[3])`
    Returns the dot product of the vectors, x1 and x2.
  - `double Magnitude(double x[3])`
    Returns the Euclidean magnitude (square root of sum of squares) of the vector, x.
  - `void Normalize(double x[3])`
    Divides each component of x by the Euclidean norm of x, giving the vector length, 1.
  - `void ScalarMultiply(double x[3], double scaler)`
    Multiplies x by scaler.
  - `void Sum(double x1[3], double x2[3], double x3[3])`
    Computes the sum of x1, x2 and return it in x2.
  - `void VectorCopy(double xout[3], double xin[3])`
    Copies the contents of xin to xout.

- Coordinate transformations.
  - `void EARTH_to_ECI(double t, double earth[3], double eci[3])`
    Computes and returns the Earth Centered Inertial (ECI) coordinate system values for the given earth coordinate system.
  - `void EARTH_to_ECR(double earth[3], double ecr[3])`
    Computes and returns the Earth Centered Rotating (ECR) coordinate system values for the passed in earth (Latitude, Longitude, Altitude) coordinate system.
  - `void ECR_to_EARTH(double ecr[3], double earth[3])`
    Computes and returns the earth coordinate system values for the given ECR coordinate system.
  - `void ECI_to_EARTH(double t, double eci[3], double earth[3])`
    Computes and returns the Earth coordinate system values for the given ECI position and time.
  - `void ECI_to_ECR(double t, double x[3])`
    Converts ECI to ECR for the given time and position. Conversion is done in place.
  - `void ECI_to_ECR(double t, double x[3], double v[3])`
    Converts ECI to ECR for the given time, position, and velocity. Conversion is done in place.
  - `void ECI_to_ECR(double t, double x[3], double v[3], double a[3])`
    Converts ECI to ECR for the given time, position, and velocity. Conversion is done in place.
  - `void ECR_to_ECI(double t, double x[3])`
    Converts ECR to ECI for the given time and position. Conversion is done in place.
3.16 SpConvert.H: Time and Space Conversion Functions

- **void ECR_to_ECI(double t, double x[3], double v[3])**
  Converts ECR to ECI for the given time, position, and velocity. Conversion is done in place.

- **void ECR_to_ECI(double t, double x[3], double v[3], double a[3])**
  Converts ECR to ECI for the given time, position, and acceleration. Conversion is done in place.

- **double get_gsam()**
  Returns the value for the additional angle through which ECI transformations are produced.

- **void set_gsam(double g)**
  All coordinate transformations to or from ECI coordinate systems require a time parameter, which is the number of seconds past 12:00 AM on January first of the current year. gsam is the additional angle through which the earth should be rotated in order to make the conversion. All these conversions assume a 365 day year (rather than 365.24... days), which may result in slightly incorrect calculations.

- **double Kilometers_to_Miles(double kilometers)**
  Returns the miles in the passed in kilometers.

- **double Kilometers_to_NauticalMiles(double kilometers)**
  Returns the nautical miles in the passed in kilometers.

- **double Miles_to_Kilometers(double miles)**
  Returns the kilometers in the passed in miles.

- **double NauticalMiles_to_Kilometers(double nauticalMiles)**
  Returns the kilometers in the passed in nautical miles.
3.17 SpDataParser: Full Featured Data Parser (obsolete)

3.17.1 Description

This SpDataParser is designed to parse files of the format shown by:

```c
// Begin example parameter file

Set_1 {
    int i_1 14 // The value of i_1 can be retrieved as an int
    reference Group_1 Set_2 // Creates a reference to later mentioned set
    reference Group_1 Set_3 // Creates a reference to later mentioned set
    Group_1 Set_4 {
        // Creates a group of sets referenced by
        // "Group_1"
        int i 40
    }
    Set_5 {
        // Embedded set as opposed to a referenced set.
        int i 50
    }
    int i_2 16 // The value of i_2 can be retrieved as an int
    enum e_1 // There is now the enumerated value "e_1" to
    // be used later
    string s_1 happy // Strings do not require quotation marks
    float f_1 1.4 // Floating point value
    logical l_1 T // Logical values are either 'T' or 'F'
}

Set_2 {
    // Non-embedded set referenced in Set_1
    int i 20
}

Set_3 {
    // Non-embedded set referenced in Set_1
    int i 30
}
```

Definitions.

- **Top-Level Set**: A named, braced collection of primitives and/or embedded sets that is not embedded in another set.
- **Embedded Set**: A named, braced collection of primitives and/or embedded sets that is embedded in another set.
- **Reference Set**: An alias of (i.e. a specifier to include) a top-level set embedded in either a top-level set or an embedded set.
- **Set Group**: A collection of embedded sets and/or reference sets nested one level into a top-level set or an embedded set unified by a common group name for the purpose of accessing each set in the group in turn via a loop control structure.
- **Group Name**: The string tag that unifies a set group.
- **Primitive**: A named value of one of the fundamental units of data extractable by the SpDataParser, consisting of one of the following types: integer, enumeration, logical, float, string, or define.
Keywords.

- “reference” specifies a set as a reference set, followed by a group name, followed by a top-level set name.
- “int” specifies an integer primitive, followed by a variable name, followed by a value.
- “enum” specifies an enumeration primitive, followed by a variable name, followed by a value.
- “logical” specifies a logical primitive, followed by a variable name, followed by a value.
- “float” specifies a floating point primitive, followed by a variable name, followed by a value.
- “string” specifies a string primitive, followed by a variable name, followed by a value.
- “define” specifies a define primitive, followed by a variable name, followed by a value.

Overview.

Methods are provided to move in and out of top-level sets (such as “Set_1” above), embedded sets (such as “Set_4” above), and reference sets (such as “Set_2” nested in “Set_1” above). The set to which the user has most recently moved is called the current set. Get* methods are provided to extract primitive data (such as ints, floats, etc.) from the current set.

Sets are reference sets when preceded by two words: the key word “reference” and a group name. Reference sets are nested sets that are actually an inline inclusion of a top-level set of the same set name. Sets may be grouped together for the user to loop through them by having a common group name (such as “Group_1” in the above example). Sets that are members of a group may be reference sets (such as “Set_2” nested in “Set_1” above) or embedded sets (such as “Set_4” above).

When looping through a group (call it Group_1) nested one level into a set (call it Set_1), the current set changes each iteration of the loop to the nested set in Group_1 currently arrived at by the loop. This set must be inline or a “reference” to a top-level set. At any point during set_a’s loop through Group_1, the user may MoveIn to the current set in Group_1, perform any Get*s, loop through groups, perform nested MoveIns and MoveOuts, and then, finally, MoveOut back to Set_1 and continue where Set_1 left off in its loop through Group_1.

Implementation.

The SpDataParser handles such nested navigation through a parameter file through internally keeping track of a stack of pointers to sets. It “pushes” a set pointer onto this stack when the user calls MoveIn or MoveToFirstSetInGroup, “pops” set pointer(s) off this stack when the user calls MoveOut or MoveOutCompletely, and both “pops” and “pushes” a set pointer when the user calls MoveToNextSetInGroup.

- **SpDataParser(const char* fileName)**
  The data parser must always be constructed with the file it is to parse. Thus, each SpDataParser is designed to parse one and only one file. To parse multiple files, simply create multiple SpDataParsers.

- **GetArraySize(char* name)**
  Gets the size of the array of a primitive, specified by its name, nested one level inside of the current set, without changing which set is the current set. If the specified primitive is not an array, this results in a terminal error.
• **GetDefine(char* name, int index = -1)**
  Gets the value of a define primitive, specified by its name, nested one level inside of the current set, without changing which set is the current set. If the specified primitive is not a define, this results in a terminal error.

• **GetEnum(char* name, int index = -1)**
  Gets the value of an enumeration primitive, specified by its name, nested one level inside of the current set, without changing which set is the current set. If the specified primitive is not an enumeration, this results in a terminal error.

• **GetFloat(char* name, int index = -1)**
  Gets the value of a float (floating point) primitive, specified by its name, nested one level inside of the current set, without changing which set is the current set. If the specified primitive is not a float, this results in a terminal error.

• **GetInt(char* name, int index = -1)**
  Gets the value of an integer primitive, specified by its name, nested one level inside of the current set, without changing which set is the current set. If the specified primitive is not an integer, this results in a terminal error.

• **GetLogical(char* name, int index = -1)**
  Gets the value of a logical primitive, specified by its name, nested one level inside of the current set, without changing which set is the current set. If the specified primitive is not a logical, this results in a terminal error.

• **GetNumSetsInGroup(char* groupName)**
  Gets the number of sets belonging to the specified set group nested one level in from the current set. This method does not change which set is the current set. This method is normally used just before looping through a set group so as to know how many sets to loop through.

• **GetString(char* name, int index = -1)**
  Gets the value of a string primitive, specified by its name, nested one level inside of the current set, without changing which set is the current set. If the specified primitive is not a string, this results in a terminal error.

• **MoveIn(char* setName)**
  Moves into a top-level set (if not currently in a set) or into a set nested in the current set. The set moved to is called the current set, in which subsequent data extraction is possible. MoveIn is used to move to a set (with or without a reference name or the keyword “reference”) that is nested in the current set (i.e. the set to which the user last moved). MoveIn must be called at least once (i.e. to move to a top-level set and then move into nested sets as desired) before any data extraction is possible. MoveIn, MoveOut, MoveOutCompletely, MoveToFirstSetInGroup, and MoveToNextSetInGroup are the five recommended methods to use for navigating through the nested data structures in a parameter file.

• **MoveOut(int numLevels = 1)**
  Moves out of a set to the set in which it is nested, optionally multiple times. Changes the current set to be the set in which the current set is nested. MoveIn, MoveOut, MoveOutCompletely, MoveToFirstSetInGroup, and MoveToNextSetInGroup are the five recommended methods to use for navigating through the nested data structures in a parameter file.

• **MoveOutCompletely()**
  Jumps out of the arbitrarily deeply nested current set, such that the next call to MoveIn moves
3.17 SpDataParser: Full Featured Data Parser (obsolete)

MoveOutCompletely is equivalent to calling MoveOut multiple times until it is no longer in any sets (i.e. resetting the SpDataParser to the initial condition where there is no current set). Thus, the user must call MoveIn with a top-level set as the argument just after calling MoveOutCompletely in order to do any subsequent data extraction. MoveIn, MoveOut, MoveOutCompletely, MoveToFirstSetInGroup, and MoveToNextSetInGroup are the five recommended methods to use for navigating through the nested data structures in a parameter file.

- **MoveToFirstSetInGroup(char* groupName)**
  Moves into the first set of the specified set group. The set to which the SpDataParser moves becomes the current set. If the first set in the specified set group is an inline set, the SpDataParser will move one nested level into that set, which becomes the current set. On the other hand, if the first set in the specified set group is a reference, the SpDataParser will actually move to the top-level set referenced and make that set the current set. MoveIn, MoveOut, MoveOutCompletely, MoveToFirstSetInGroup, and MoveToNextSetInGroup are the five recommended methods to use for navigating through the nested data structures in a parameter file.

- **MoveToNextSetInGroup()**
  Moves into the next set of the set group specified by the last call to MoveToFirstSetInGroup at this level in the nested structure. The set to which the SpDataParser moves becomes the current set. If the next set in the set group specified by the last call to MoveToFirstSetInGroup is an inline set, the SpDataParser will move out of the current set and then into that inline set, which becomes the current set. On the other hand, if the next set is a reference, the SpDataParser will move out of the current set and then actually move to the top-level set referenced and make that set the current set. MoveIn, MoveOut, MoveOutCompletely, MoveToFirstSetInGroup, and MoveToNextSetInGroup are the five recommended methods to use for navigating through the nested data structures in a parameter file.

3.17.2 Used in

SpSimObj 2.15.

3.17.3 Notes

Obsolete parser. See Section 3.28 for an improved parser and additional information.
3.18 SpDynPtrArray and RB_SpDynPtrArray: Dynamic Array of Pointers

3.18.1 Description

Several classes are provided by SPEEDES that provide varying tradeoffs in access, removal, or insertion speed. The SpDynPtrArray and its rollbackable version, the RB_SpDynPtrArray, provide constant ($O(1)$) lookup time.

- **SpDynPtrArray(int arraySize = 2)**
  Constructor that allows one to define the initial size of the dynamic pointer array if the default of 2 is not acceptable.

- **RB_SpDynPtrArray(int arraySize = 2)**
  Same as SpDynPtrArray(int arraySize = 2), above.

- **RB_voidPtr operator[](int index)**
  `void* operator[](int index)`
  Allows one to access a SpDynPtrArray as if it were a real array by using the [] operator. The $index^{th}$ element is returned which is a void* pointer. All the pointers are initialized to NULL and users can set or retrieve the $index^{th}$ element using the [] operator.

3.18.2 See also

RB_voidPtr 3.12.
3.19 C_FREE_LIST: Class for Maintaining Multiple Free Lists (obsolete)

3.19.1 Description

A free list contains instances of a class that are heavily used at run-time but with an unpredictable usage pattern (i.e., it is not known how many instances of the class will be needed or when they will be needed before runtime). Typically, instances of such classes are dynamically allocated and de-allocated as they are needed; however, dynamic memory allocation/deallocation can be very slow. Free lists keep a number of pre-allocated instances on hand to be used as needed and returned to the free list instead of deallocated. Using free lists can increase performance at runtime in exchange for potentially using more memory than is strictly necessary (as it is not likely that all instances on a free list will be in use at all times).

The C_FREE_LIST class maintains multiple free lists, each of a different class or type (which must inherit from C_ITEM and cannot have multiple inheritance). The set_types method determines how many different free lists the C_FREE_LIST will manage and should be invoked prior to any other method. The set_type methods can then be used to define the parameters of each free list in the C_FREE_LIST. Each class that has a free list also has a typeId (set by using the set_type method) that identifies that class in calls to other C_FREE_LIST methods. Instances of a free list class are accessed via the new_object method, and disposed of with the delete_object or delete_list methods.

If new_object is called for a type and that particular free list is empty (because all the instances of that type are being used), another group of instances of the proper class are allocated.

- **FREE_DEFINE_CLASS(nameForCtorFunc, className)**
  This macro, defined in the SpFreeList.H header file, declares a class that will be used in a free list. It is meant to be used with the set_type method, detailed below. The macro creates a global function with the name, nameForCtorFunc, that can be used as set_type’s arrayCtor argument. The macro’s className argument should be the name of the class that is to be used in a free list.

- **C_FREE_LIST(void)**
  Constructor for the C_FREE_LIST class.

- **void CopyType(char* newTypeName, int newTypeId, int sourceTypeId)**
  Can be used instead of set_type(char* typeName, int typeId, ARRAY_OF_CHILDREN* (*)(int) arrayCtor, int classSize, int howMany) to define a new type. All of the type information previously given for the class associated with sourceTypeId is duplicated, except for the type name, and associated with newTypeId. The newTypeName argument determines the name of the new free list type.

- **void delete_list(C_QUEUE* queue)**
  Takes a C_QUEUE filled solely with objects that have been returned by the new_object method and empties it, returning all the objects to their proper free list. If the C_QUEUE is NOT solely filled with objects that were returned by the new_object method, undefined behavior may occur.

- **void delete_object(void* freeListObject)**
  Returns an instance that is no longer needed to its free list instead of deleting it.

- **int get_size(int typeId)**
  Returns the size, in bytes, of the class associated with typeId.
• **int get_size(C_ITEM* freeListObject)**
  Returns the size, in bytes, of the given object’s actual class, assuming that the object is a part of a free list. If it is not, -1 is returned instead.

• **char* GetName(int typeId)**
  Returns the name of the class that is associated with typeId.

• **char** GetNames(void)
  Returns an array of strings, which contains the names of the classes that have free lists. The size of the array is accessed from the GetNtypes method.

• **int GetNtypes(void)**
  Returns the number of different classes that have free lists. This is also the value passed to the last invocation of the set_ntypes method.

• **void** new_object(int typeId)
  Returns an instance of the class that is associated with typeId. Ideally, the instance is taken off of its free list and does not have to be dynamically allocated. If, however, the free list is empty, more instances will be allocated, added to the free list, and one of the new instances will be returned.

• **void print(ostringstream& out = cout)**
  Prints status information to the given ostream. If no ostream is supplied, cout is used as the default.

• **void set_ntypes(int numTypes)**
  The first time this method is called, enough space for numTypes different free lists is allocated. If this method is called more than once, subsequent calls will increase or decrease the previously allocated space to fit the new value of numTypes. This method should be called before any other method on the C_FREE_LIST class, as most of the other methods depend on this space being allocated.

• **void set_type(char* typeName, int typeId, void (*)(int) oldArrayCtor, int classSize, int howMany)**
  Sets the name, typeId, array constructor, size in bytes, and allocation size for a class that is to have its own free list. The typeId argument should be between 0 and (numTypes - 1) inclusive, where numTypes is the integer value passed to the last invocation of the set_ntypes method. Each class that has a free list must have its own typeId that is unique to this instance of C_FREE_LIST. The oldArrayCtor argument must be a function that takes an integer value and returns an array of the class that is to have a free list. The integer value will specify the length of the array. The howMany argument specifies how many instances of the class will be allocated for its free list. This version of the set_type method is obsolete, and included solely for backwards compatibility purposes.

• **void set_type(char* typeName, int typeId, ARRAY_OF_CHILDREN*(*)(int) arrayCtor, int classSize, int howMany)**
3.19 C_FREE_LIST: Class for Maintaining Multiple Free Lists (obsolete)

Sets the name, typeId, array constructor, size in bytes, and allocation size for a class that is to have its own free list. This version of the set_type method is intended to be used in conjunction with the FREE_DEFINE_CLASS macro, detailed above. The typeId argument should be between 0 and (numTypes - 1) inclusive, where numTypes is the integer value passed to the last invocation of the set_ntypes method. Each class that has a free list must have its own typeId that is unique to this instance of C_FREE_LIST. The specifics of the arrayCtor argument are unimportant (ARRAY_OF_CHILDREN is an internal class); simply pass in the nameForCtorFunc that was given to the FREE_DEFINE_CLASS macro for the class. The howMany argument specifies how many instances of the class will be allocated for its free list.

3.19.2 Examples

The following illustrates how to set up a free list for a class:

```c
#include "SpFreeList.H"

class MyClass : public C_ITEM {
   ...
};
FREE_DEFINE_CLASS(AllocateMyClassArray, MyClass);
```

The FREE_DEFINE_CLASS macro will create a function named AllocateMyClassArray which we will use when we define the MyClass free list, shown here:

```c
C_FREE_LIST MyFreelists;

MyFreelists.set_ntypes(1); // We only have 1 type,  // but we could have more
   MyFreelists.
      .set_type("MyClass", 0,  // We determine the typeId
         AllocateMyClassArray, // function generated by macro
         sizeof(MyClass), 10);  // allocate 10 MyClass instances to start
   ...
   // Get a new instance of MyClass
   MyClass* classInstance = (MyClass* ) MyFreelists.new_object(0);
   ...
   // Once we are finished with it, we return it to the free list
   MyFreelists.delete_object(classInstance);
```

3.19.3 See also

C_ITEM 3.2, and SpMempool 3.1.
3.20 SpHashTree and RB_SpHashTree: Hash Table of Trees

3.20.1 Description

The SpBinaryTree and RB_SpBinaryTree provide an \( O(N \log(N)) \) speed for lookup, insertion, and removal. The SpDynPtrArray and RB_SpDynPtrArray provide constant time lookup but have much slower \( O(N) \) inserts and removals if order must be maintained and empty slots filled. At the expense of greater memory usage, the SpHashTree and RB_SpHashTree try to balance these needs by providing faster operations than both of these classes.

Rather than using templates, these hash trees contain void* values. This means that users are responsible for memory management of items inserted or removed from these hash trees.

- **SpHashTree(int arraySize = 10)**
  These constructors set up a hash table (an array) of binary trees. Whenever the user inserts an item into these classes, it is first hashed into a specific tree and then inserted into that tree. The goal is to have a small number of items in any specific tree, reducing the overall time required to search through each tree. Choose the initial size so that, in the ideal situation, the number of items in each tree (often referred to as a hashing bucket) is “small”.

- **RB_SpHashTree(int arraySize = 10)**
  Same as SpHashTree(int arraySize = 10), above.

- Items of any type can be inserted into the tree. Simply pass the address and the key. Duplicates are allowed and are then found in the order in which they are inserted. That is, if one inserts two items with the same key, the first one inserted will be found first by Find or Remove, etc.

  It is not allowed to mix two types of inserts. The SpHashTree and RB_SpHashTree do not allow inserting using doubles and switching to using strings (or vice-versa) on a non-empty hash tree. This is due to the fact that there is not a standard ordering for the set of strings and doubles. If, however, a hash tree is filled using double keys and then emptied, that hash tree can then be used with string keys or vice-versa. Also, do not insert NULL into these classes, because NULL is frequently used as an end of list indicator for iteration.

  - **void Insert(void* item, char* keyName)**
    Inserts item using the key, keyName. keyName is then copied and stored internally with the item in the hash tree.

  - **void Insert(void* item, double keyNum)**
    Inserts item using the key, keyNum. Use of integer keys is handled by this call, as well, through C++’s type promotion from integer to double.

- Removal. The following methods provide several options for removing items from a hash tree. One can remove a specific item by passing in a key and the item is returned upon removal. If the item is not found, NULL is returned.

  - **void* Remove(char* keyName)**
    Removes and returns the first item inserted and still in the hash tree with the key, keyName.

  - **void* Remove(double keyNum)**
    Removes and returns the first item inserted and still in the hash tree with the key, keyNum.
3.20 SpHashTree and RB_SpHashTree: Hash Table of Trees

- `void* RemoveFirstElement()`  
  Removes and returns the first item in the hash tree. If the hash tree is empty, returns NULL.  
  If the key (or keyName) is desired, retrieve this value prior to removal by calling a Get-FirstElement GetCurrentKeyName combination first.

- The following methods find an item in the hash tree with a given key. If multiple items are in the hash tree with the key being sought, then the item that was inserted first will be returned (subsequent items can be found by calling FindIterator and then GetNextElement on the returned iterator). If a requested element cannot be found, NULL is returned.
  
  - `void* Find(const char* keyName)`  
    Finds and returns an element with key, keyName.
  
  - `void* Find(const double keyNum)`  
    Finds and returns an element with key, keyNum.
  
  - `SpIterator_SpHashTree FindIterator(const char* keyName)`  
    Returns a SpIterator_SpHashTree pointing to the key, keyName.
  
  - `SpIterator_RB_SpHashTree FindIterator(const char* keyName)`  
    Returns a SpIterator_RB_SpHashTree pointing to key, keyName.
  
  - `SpIterator_SpHashTree FindIterator(double keyNum)`  
    Returns a SpIterator_SpHashTree pointing to the key, keyNum.
  
  - `SpIterator_RB_SpHashTree FindIterator(double keyNum)`  
    Returns a SpIterator_RB_SpHashTree pointing to the key, keyNum.

- `int GetNumElements()`  
  Returns the number of elements in the hash tree.

- There are many methods for traversing a hash tree and examining all of its elements. The iterator methods are affected by manipulating the contents of a hash tree, so do not call GetNextElement or GetPreviousElement after changing the hash tree or before a call to GetFirstElement or GetLastElement. However, the methods below are deprecated and SpIterator_RB_SpHashTree and SpIterator_SpHashTree are preferred (see Section 3.23).

  It has a consistent order, but it is an arbitrary order rather than a sorted order. For example, iterating forwards gives the exact opposite order from iterating backwards.

  - `void* GetCurrentElement()`  
    Returns the current element of the hash tree.
  
  - `void* GetFirstElement()`  
    Returns the first element in the hash tree and sets the built-in iterator to point to it. If the container is empty, returns NULL.
  
  - `void* GetLastElement()`  
    Returns the last element in the hash tree and sets the built-in iterator to point to it. If the container is empty, returns NULL.
  
  - `void* GetNextElement()`  
    Advances to the next element in the hash tree and returns it. If the end of the hash tree has been reached, returns NULL.
  
  - `void* GetPreviousElement()`  
    Moves back to the previous element and returns it. If the start of the hash tree has been reached, returns NULL.
- **void** operator ++()
  Advances to the next element in the hash tree and returns it. If the end of the hash tree has been reached, returns NULL. This operator is the same as GetNextElement.

- **void** operator ++(int)
  Returns the current element and advances the built-in iterator to next element in the container.

- **void** operator --()
  Moves back to the previous element and returns it. If the start of the hash tree has been reached, returns NULL. This operator is the same as GetPreviousElement.

- **void** operator --(int)
  Returns the current element and moves the built-in iterator back to the previous element in the container.

- **double** GetCurrentKey()
  Returns the key of the current element for a hash tree using double keys.

- **char** GetCurrentKeyName()
  Returns the key of the current element for a hash tree using string keys. Do not delete the string returned by this call.

- The RB_SpHashTree and the SpHashTree support several modes of operation relating to the way the trees are balanced.
  - NormalTreeMode does not ever attempt to balance the tree (default mode). If data is to be inserted or removed from the tree in a completely random order, this method will perform very well.
  - BalancedTreeMode does not keep the tree perfectly balanced but uses a heuristic to balance the tree whenever it becomes too unbalanced. This will help maintain a relatively well-balanced tree if the data is well ordered when it is inserted into the tree or if the data is always removed from one end of the tree.
  - SplayTreeMode implements a splay tree algorithm. Whenever an item is inserted into the tree, that item is rotated to the root of the tree. Similarly, when an item is removed, it is first rotated to the root of the tree and then removed. Finally, whenever an item is found in the tree through one of the Find calls, it is rotated to the root of the tree. This algorithm has a similar effect as does a “caching” algorithm. Whenever an item is accessed, the items close to it are moved into a position where they can more quickly be accessed as well. If the items in the tree are inserted, removed, or accessed in a completely random fashion, this mode will yield poor performance.

- The methods for changing and examining the balancing heuristic are:
  - **void** SetBalancedTreeMode()
    Sets the trees to balanced tree mode.
  - **void** SetNormalTreeMode()
    Sets the trees to normal tree mode.
  - **void** SetSplayTreeMode()
    Sets the trees to splay tree mode.
  - **int** IsBalancedTreeMode()
    Returns 1 if the trees are in balanced tree mode, 0 otherwise.
3.20 SpHashTree and RB_SpHashTree: Hash Table of Trees

- `int IsNormalTreeMode()`
  Returns 1 if the trees are in normal tree mode, 0 otherwise.
- `int IsSplayTreeMode()`
  Returns 1 if the trees are in splay tree mode, 0 otherwise.

3.20.2 Bugs

Free lists are used to allocate the tree items and the keys for the tree. These free lists do not clean themselves up at exit, leading to what appears to be a memory leak.

3.20.3 See also

3.21 SpHostUser: Internal Host Router Interface

3.21.1 Description

The SpHostUser class is the internal SPEEDES interface to the host router (a part of the SpeedesServer), which allows communication with the world outside of SPEEDES. From external modules, the only way to get information from inside of SPEEDES is to use the SpEmHostUser class, which simply gets its information from the host router. The way this information is sent to the host router from SPEEDES is through the use of the SpHostUser interface. This interface allows one to get information out of SPEEDES, as well as get information into SPEEDES.

- **SpHostUser()**
  Creates the HostUser and connects to the host router. This constructor will look in the current directory for a speedes.par file, and attempt to extract the hostname, port number, and simulation group id for the host router from it. Called automatically by ExecuteSpeedes.

- **SpHostUser(char* oldSimData)**
  Creates the HostUser and connects to the host router. One can record all messages from the outside world and then pipe them back into a simulation using this call. This can be very useful for debugging your applications. Currently, no API exists for using this constructor.

- **SpHostUser(char* serverHost, int portNumber, int simGroupId)**
  Creates the HostUser and connects to the host router. With this constructor, the user can specify the hostname, port number, and the simulation’s group id directly, instead of getting that data from a par file.

- **SpHostUser(SpDataParser* parser)**
  Creates the HostUser and connects to the host router. The parser passed into this constructor must have already read in the par file that contains the hostname, port number, and simulation group id for the host router.

- **void Flush()**
  Sends a flush message to the host router. In order to make sure that the host router has received previously sent data, a call to this method acts similar to a ping. It sends a message to the host router and then waits for the return. This assumes that TCP/IP (or a similar transport mechanism) is being used that guarantees an in order delivery of messages.

- **GENERIC_MSG_HEADER* GetMessage()**
  Returns a message from the host router, if there is one. This method first checks to see if any messages have been queued up from a prior call to the ReadMessages method. If so, the first message that was read in is returned. If there are no messages waiting in the queue, there is a check to see if there are any messages waiting to be read in. If not, NULL is returned.

- **GENERIC_MSG_HEADER* GetMessage(int msgType)**
  Returns a message of the specified type from the host router, if there is one. This method first checks to see if any such messages have been queued up from a prior call to the ReadMessages method. If so, the first message of the desired type that was read in is returned. If there are no such messages waiting in the queue, there is a check to see if there are any such messages waiting to be read in. If not, NULL is returned.
• **void QueryReply(SpNameValueList* queryResponse, int targetExternalId)**  
  Sends a SpNameValueList in response to a query message sent by the external module specified by targetExternalId. The name value list can contain anything the user wishes.

• **void RB_SendData(SpSimTime sendTime,  
  int    targetExternalId,  
  char*   data,  
  int    dataBytes,  
  int    senderGlobalId)**  
  Rollbackable version of SendData.

• **void RB_SendNamedData(char*  
  dataName,  
  SpSimTime sendTime,  
  int    targetExternalId,  
  char*   data,  
  int    dataBytes,  
  int    senderGlobalId)**  
  Rollbackable version of SendNamedData.

• **void RB_SendSubscribedData(SpSimTime sendTime,  
  char*    subscriptionName,  
  char*    data,  
  int    dataBytes,  
  int    senderGlobalId)**  
  Rollbackable version of SendSubscribedData.

• **void ReadMessages()**  
  Reads all messages that have arrived from the host router since the last time this method was invoked, and sticks them in a queue.

• **void SendData(SpSimTime sendTime,  
  int    targetExternalId,  
  char*   data,  
  int    dataBytes,  
  int    senderGlobalId)**  
  Sends the supplied data to the external module specified by targetExternalId. The global id of the current simulation object should be passed in as the senderGlobalId argument. This method is not rollbackable.

• **void SendNamedData(char*  
  dataName,  
  SpSimTime sendTime,  
  int    targetExternalId,  
  char*   data,  
  int    dataBytes,  
  int    senderGlobalId)**  
  Sends the supplied data to the external module specified by targetExternalId. The global id of the current simulation object should be passed in as the senderGlobalId argument. This method is not rollbackable.
• void SendSubscribedData(SpSimTime sendTime,
  char* subscriptionName,
  char* data,
  int dataBytes,
  int senderGlobalId)

Sends the supplied data to the external module specified by targetExternalId. The data also has the supplied subscription name associated with it, which is used by external modules based around the SpStateMgr (e.g. Object Proxy State Manager (OPSM)). The global id of the CurrentSimObj should be passed in as the senderGlobalId argument. This method is not rollbackable.

3.21.2 See also

3.22 SpList and RB_SpList: List Container Classes

3.22.1 Description

SpList and RB_SpList are container classes with constant time inserts at the ends and removals from the ends, but with linear time searches or removals from the middle. As with the other container classes, these both share a common API.

Rather than using templates, these trees contain void* values. This means that users are responsible for memory management of items inserted or removed from these trees.

- **void Insert(void* item, int topOrBot = SpList::BOT_FLAG)**
  Inserts an item at the top or bottom of the list. If topOrBot is equal to SpList::TOP_FLAG (or RB_SpList::TOP_FLAG), it is inserted at the top. Insertion at the bottom is done through the use of SpList::BOT_FLAG (or RB_SpList::BOT_FLAG). Do not insert NULL.

- **Removal.** The following methods provide several options for removing items from a list. One can remove a specific item by passing in the element’s address and the item is returned upon removal. If the item is not found, NULL is returned.
  - **void* Remove(void* element)**
    Removes and returns the element with the same address as is passed in.
  - **void* RemoveFirstElement()**
    Removes and returns the first item in the list. If the list is empty, returns NULL.
  - **void* RemoveLastElement()**
    Removes and returns the last item in the list. If the list is empty, returns NULL.

- There are many methods for traversing a list and examining all of its elements. The iterator methods are affected by manipulating the contents of a list. Do not call GetNextElement or GetPreviousElement after a Remove or Insert call has been made or before a call to GetFirstElement or GetLastElement.
  - **void* GetCurrentElement()**
    Returns the current element of the list.
  - **void* GetFirstElement()**
    Returns the first element in the list. If the list is empty, returns NULL.
  - **void* GetLastElement()**
    Returns the last element in the list. If the list is empty, returns NULL.
  - **void* GetNextElement()**
    Returns the next element in the list. If the end of the list has been reached, returns NULL.
  - **int GetNumElements()**
    Returns the number of elements in the list.
  - **void* GetPreviousElement()**
    Returns the previous element in the list. If the start of the list has been reached, returns NULL.
  - **void* operator ++()**
    Same as GetNextElement but prefix operator.
  - **void* operator --()**
    Same as GetPreviousElement but prefix operator.
3.22.2 See also

SpBinaryTree 3.13, RB_SpBinaryTree 3.13, SpDynPtrArray 3.18, RB_SpDynPtrArray 3.18, SpHashTree 3.20, RB_SpHashTree 3.20, and Independent Iterators 3.23.
3.23 Independent Iterators

3.23.1 Description

An iterator is an object which can be set to point to a particular place in a container class. Multiple iterator objects may independently and simultaneously iterate on the same container object. There are independent iterators for the SpList, RB_SpList, SpBinaryTree, RB_SpBinaryTree, SpHashTree, and RB_SpHashTree.

- **SpIterator** <Container>()
  Constructor for an iterator on <Container> class where <Container> is replaced with the container class name. The iterator will not point anywhere, but may be used for iterating after setting it to another valid iterator.

- **<Container>** Iterator(const <Container> &container)
  Construct an iterator for the passed in container. The iterator initially points to the first element or NULL if the container is empty.

- **<Container>** Iterator(const <Container> &container, int topOrBot)
  Same as previous except that it takes either <Container>::TOP_FLAG to start at the first element or <Container>::BOT_FLAG to start at the last element.

- **void** GetFirstElement()
  Set the iterator to the first element in the container and return that element or return NULL if the container is empty.

- **void** GetLastElement()
  Set the iterator to the last element in the container and return that element or return NULL if the container is empty.

- If changes are made to a container, then all iterators on that container become invalid for further use of the current position. An invalid position will become valid again if the iterator is set equal to an iterator with a valid position or if GetFirstElement or GetLastElement is called on the iterator. Specifically, do not call the following without resetting the iterator with a GetFirstElement or GetLastElement call or through assignment from a valid iterator.

  - **operator void**( ) const
    Return the current element or NULL if there are no elements left.

  - **void** GetCurrentElement() const
    Same as operator void*.

  - **double** GetCurrentKey() const
    Return the key of the current element for a container using double keys. Return -1 if there are no elements left. This only works on SpMapIterator and its descendents.

  - **char** GetCurrentKeyName() const
    Return the key of the current element for a container using string keys. Return NULL if there are no elements left. Do not delete the string returned by this call. This only works on SpMapIterator and its descendents.

  - **void** GetNextElement()
    Advance the iterator to the next element in the container and return that element or return NULL if there are no more elements.
3.23.2 Examples

Here is an example of traversing with an iterator:

```c++
RB_SpBinaryTree temp;
//... Add items to the tree
RB_cout << "Start printing" << endl;
for (SpIterator_RB_SpBinaryTree iterator(temp); iterator; ++iterator) {
    RB_cout << iterator << ' ' << iterator.GetCurrentKey() << endl;
}
RB_cout << "end printing" << endl;
```

3.23.3 See also

RB_SpBinaryTree 3.13, RB_SpHashTree 3.20

3.23.4 Notes

For SpHashTree, and RB_SpHashTree, iterating has a consistent order, but it is an arbitrary order rather than a sorted order. For example, iterating forwards gives the exact opposite order from iterating backwards.
3.24 SpMsg: Event Message Base Class

3.24.1 Description

All events are scheduled through sending a message either to a local object or to an object on another node. The unified API generates a message for each event which inherits from this class and adds in the additional data for the argument for the event method or handler.

All the data for this class is public and consists of network safe classes (NET_INT and NET_DOUBLE) to ensure compatibility between different endian nature machines.

- The following information is used by all events:
  - **SpSimTime ScheduleTime**
    Time tag for the message (future event).
  - **NET_INT bytes**
    Bytes in data for variable size message, for critical path.
  - **NET_INT cancel_id**
    Cancel event id.
  - **NET_INT evtype**
    Event type.
  - **NET_DOUBLE EarliestStartTime**
    Earliest start time of the event scheduled from this message.
  - **NET_INT ExternalId**
    The id of the external module.
  - **NET_INT objid**
    Destination object id.
  - **NET_INT obtype**
    Destination object type.
  - **NET_INT rbid**
    Rollback id.

- The following three are used only for event handlers:
  - **NET_INT type**
    Event id.
  - **NET_INT undirected**
    Boolean for directed or undirected.
  - **NET_INT varsizem**
    Size of appended variable-length data, because normal “data” is the string name of the handler.

- The following two are used within the process model:
  - **NET_INT ProcId**
    The id for the particular process in order to recover local variables.
  - **NET_INT ProcReentryType**
    The reason for the process model waking up (either beginning of a process, timeout, or a semaphore being triggered).
3.24.2 Examples

All of the DEFINE_*_EVENT macros create a special SpMsg class that is used to pass the parameters of the method. For example, the macro, DEFINE_*_EVENT_3_ARG(MyEvent, MyEventClass, MyMethod, int, double, myClass), generates the class:

class M_MyEvent_ARG : public SpMsg {
  public:
    M_MyEvent_ARG() {}  
    M_MyEvent_ARG(int p1, double p2, myClass p3) : 
      v1(p1), 
      v2(p2), 
      v3(p3) {} 
    int    v1; 
    double v2; 
    myClass v3; 
};

Since this is the actual class used to transfer data from the scheduler to the event, everything in this class must be “flat”, meaning that dynamic memory and references are not allowed. Also, classes must have a copy constructor due to their setting in the member initialization list.

If heterogeneous computing is desired, all parts of the message must be either NET_INT, NET_DOUBLE or classes built out of these types. Alternatively, the data marshaling could be performed on a method-by-method basis.
3.25 SpNameValueList: List of Associated Names and Double Values

3.25.1 Description

The SpNameValueList class is used to associate names and floating point values within SPEEDES. This is used in external modules to query the state of the simulation, as well as in several internal methods and routines to exchange name/value pairs.

- **SpNameValueList()**
  Default constructor which sets the name of the list to “UnNamedList”.

- **SpNameValueList(char* buffer)**
  Constructs a new name value list from another SpNameValueList, which has been packed into a char* buffer.

- **SpNameValueList(int n)**
  Constructs a new name value list, which will have n name/value pairs.

- **char* GetName(int i)**
  Returns a pointer to the i\textsuperscript{th} name/value pair in the list. Do not delete.

- **char* GetNameOfList()**
  Returns a pointer to the name of the list. Do not delete.

- **int GetNameValue(int i, char*& n, double& v)**
  Returns the name/value pair in n and v for the i\textsuperscript{th} pair and 0 is returned. If i is invalid (negative or greater than the number of elements in the list), 1 is returned. In either case, do not delete n.

- **GetNitems()**
  Returns the number of pairs in the list.

- **int GetSizeOfBuffer()**
  Returns the size of the SpNameValueList when it is packed into a buffer.

- **double GetValue(char* n, int& nameFound)**
  Returns the value for the name, n, in the list. If n is not found, nameFound is set to 0 upon return.

- **double GetValue(int i)**
  Returns the value of the i\textsuperscript{th} name/value pair in the list.

- **char* MakeBuffer(int& size)**
  Packs the name value list into a buffer and returns the size of the buffer in the parameter size. Delete the returned value with “delete [] buffer” when finished with it.

- **int PackBuffer(char* buffer, int size)**
  Pack the list into buffer that contains size bytes. If size does not agree with the size of the name value list, 1 is returned and an error message is printed. Otherwise, 0 is returned.

- **void Print(ostream& out = cout)**
  Prints the list to the stream.

- **void SetNameOfList(char* name)**
  Defines the name of the list to be name.
• **void SetNameValue(int whichValue, char* n, double v)**
  Sets the whichValue\(^{th}\) item to have the name, n, and the value, v. The list will automatically grow to accommodate new elements.

• **void SetNitems(int n)**
  Defines the name value list to have n name/value pairs.

• **void UnPackBuffer(char* buffer)**
  Unpacks into the current list, the packed SpNameValueList that is contained in buffer.

### 3.25.2 Examples

This class is used to fill out the response to an external module’s query call. For example:

```c++
#include "SpNameValueList.H"
#include "SpSimObj.H"

class S_MySimObj : public SpSimObj {
public:
  virtual void Query(double time_tag, SpnameValueList& nameValueList) {
    extern int Debug;
    nameValueList.SetNameValue(0, "Weight", Weight);
    nameValueList.SetNameValue(1, "Length", Length);
    nameValueList.SetNameValue(2, "Allegence", Allegence);
    if (Debug == 1) {
      for (int i = 0; i < 3; ++i) {
        cout << "Item: " << i
            << nameValueList.GetName(i) << " : "
            << nameValueList.GetValue(i) << endl;
      }
    }
  }
private:
  RB_double Weight;
  RB_double Length;
  RB_int Allegence;
};
```

### 3.25.3 Used in

Query 2.15, and SpEmHostUser 6.1.
3.26 SpObjHandle: Simulation Object Handle

3.26.1 Description

An object handle is a reference to a specific simulation object. It contains three integers that uniquely map each object in the simulation. This triplet is the node, object manager type, and local id of the object and is encapsulated in this class.

- **int GetNodeId()**
  Returns the node id of the object specified by this object handle.

- **int GetSimObjLocalId()**
  Returns the local id of the object specified by this handle.

- **int GetSimObjMgrId()**
  Returns the manager id of the object specified by this handle.

- **void SetNodeId(int n)**
  Sets the node for this object handle to n.

- **void SetSimObjLocalId(int l)**
  Sets the local id for this object handle to l.

- **void SetSimObjMgrId(int o)**
  Sets the manager id for this object handle to o.

3.26.2 Examples

Generally, object handles are obtained from the global functions in SpGlobalFunctions.H but, occasionally, one can build them from scratch. For example, if you wanted to schedule an event on an object for which you have its proxy (see Section 4.35), the handle can be obtained like this:

```c++
//...
F_SpProxyItem* pItem =
  (F_SpProxyItem *) GetRemoteObjectProxies()->get_top();
SpObjProxy* proxy = pItem->GetObjProxy();
SpObjHandle otherHandle;
otherHandle.SetNodeId(proxy->GetProxyNode());
otherHandle.SetSimObjMgrId(proxy->GetProxySimObjMgrId());
otherHandle.SetSimObjLocalId(proxy->GetProxySimObjLocalId());
```

3.26.3 See also

3.27 **SpParmSet and SpParmSetElement: Parameter Set for Interactions**

### 3.27.1 Description

SpParmSet is a container class that holds elements of known types (instead of void pointers, as in SpBinaryTree) referenced by either string or integer key values. In addition to the regular Insert and Remove methods, this class supports C-style array usage (with the key value replacing the array index). It is also capable of packing and unpacking itself into and out of a buffer.

- Creating a SpParmSet.
  - **SpParmSet(), SpParmSet(SpParmSet& copySet)**
    Basic and copy constructors.
  - **SpParmSet(char* buff, int size)**
    Creates a SpParmSet from an array of bytes generated by the GenerateBuff or PackBuff methods. See the “Packing into a buffer” section below.

- Insertion. Elements of any known type (integer, float, double, string, buffer) can be inserted into the set along with a key value using the proper method from the list below. String and buffer data inserted into the set will be copied, rather than retaining a pointer to the user’s data. As with SpBinaryTree, key types cannot be mixed. Once an element has been inserted with an integer key, all following inserts must also use integer keys. There is no similar restriction on element types. Key values do not need to be unique; duplicate keys will be found in the order in which they are inserted when iterating through the set.
  - **void InsertBuffer(int intKey, char* buff, int size),**
    void InsertBuffer(char* stringKey, char* buff, int size)**
    Inserts a buffer into the SpParmSet with the given key and size.
  - **void InsertDouble(int intKey, double value),**
    void InsertDouble(char* stringKey, double value)**
    Inserts a double into the SpParmSet with the given key.
  - **void InsertFloat(int intKey, float value),**
    void InsertFloat(char* stringKey, float value)**
    Inserts a float into the SpParmSet with the given key.
  - **void InsertInt(int intKey, int value),**
    void InsertInt(char* stringKey, int value)**
    Inserts an integer into the SpParmSet with the given key.
  - **void InsertString(int intKey, char* value),**
    void InsertString (char* stringKey, char* value)**
    Inserts a string into the SpParmSet with the given key.

- Removal. The following methods will remove the element associated with the supplied key value, and return it to the caller as the specified type (e.g. double in the case of RemoveDouble). If the supplied key is not present in the SpParmSet, 0 or NULL will be returned instead. The caller is responsible for deleting returned string or buffer data.
  - **char* RemoveBuffer(int intKey, int& size),**
    char* RemoveBuffer(char* stringKey, int& size)**
    Returns the char* (i.e. buffer) specified by the key. The size of the buffer is returned in size.
3.27 SpParmSet and SpParmSetElement: Parameter Set for Interactions

- double RemoveDouble(int intKey),
  double RemoveDouble(char* stringKey)
  Returns the double specified by the key.

- float RemoveFloat(int intKey),
  float RemoveFloat(char* stringKey)
  Returns the float specified by the key.

- int RemoveInt(int intKey),
  int RemoveInt(char* stringKey)
  Returns the integer specified by the key.

- char* RemoveString(int intKey),
  char* RemoveString(char* stringKey)
  Returns the char* (i.e. NULL-terminated string) specified by the key.

- Retrieval. The following methods will return the value of the element associated with the supplied key value. If the supplied key is not present in the SpParmSet, 0 or NULL will be returned instead. If the value was inserted as one type and is being retrieved as another (e.g. inserted as double but retrieved as int) the value will be cast to the retrieved type, with the caveat that string and buffer data cannot be cast as any of the numerical types.

- char* GetBuffer(int intKey, int& size),
  char* GetBuffer(char* stringKey, int& size)
  Retrieves the char* (i.e. buffer) value for the input key. The size of the buffer is returned in size.

- double GetDouble(int intKey),
  double GetDouble(char* stringKey)
  Retrieves the double value for the input key.

- float GetFloat(int intKey),
  float GetFloat(char* stringKey)
  Retrieves the float value for the input key.

- int GetInt(int intKey),
  int GetInt(char* stringKey)
  Retrieves the integer value for the input key.

- char* GetString(int intKey),
  char* GetString(char* stringKey)
  Retrieves the char* (i.e. NULL-terminated string) value for the input key.

- Iterating. The user can use the GetFirst and GetNext methods to iterate through the key values in the SpParmSet, and can then lookup those keys with the retrieval methods detailed above.

- SpParmSetDataType GetDataType(int key),
  SpParmSetDataType GetDataType(char* stringKey)
  Returns the data type of the element for the supplied key. The return value is one of the enumerated values INT_DATA, FLOAT_DATA, DOUBLE_DATA, STRING_DATA, BUFFER_DATA, or NONE_DATA. A return value of NONE_DATA should be ignored.

- int* GetFirstIntParmId(),
  int* GetNextIntParmId()
  Returns a pointer to the integer key of the first or next element. A return value of NULL indicates that the end of the set has been reached.
– char* GetFirstStringParmId(),
  char* GetNextStringParmId()
  Returns a pointer to the string key for the first or next element. A return value of NULL indicates that the end of the set has been reached.

– int GetNumElements()
  Returns the number of elements in the set.

– SpParmSetIdType GetParmIdType()
  Returns the type of keys used in this set. This is one of ID_TYPE_NONE, ID_TYPE_INT, or ID_TYPE_STRING. A return value of ID_TYPE_NONE means that the SpParmSet is empty.

• Alternative Access Methods. The [ ] operator provides an easier way to insert and retrieve data from a SpParmSet.

  – SpParmSetElement& operator [ ] (int intKey)

  – SpParmSetElement& operator [ ] (char* stringKey)

Because the SpParmSetElement class has conversion operators for the supported data types (int, float, double, string), as well as an overloaded = operator, it is never directly seen by the user. Instead, it functions as a stand-in for the actual data stored in the SpParmSet and can be modified just as if it were that data. See the Examples section below for information on how the [ ] operator is used.

• Packing into a Buffer. The following methods are provided for those wishing to take a SpParmSet and flatten it into a byte array.

  – SpParmSet* Clone()
    Returns a deep copy of the SpParmSet.

  – char* GenerateBuff(int& size)
    Generates and returns a buffer that contains the SpParmSet data. The length of the buffer is returned in the parameter size. It is up to the caller to delete the returned buffer.

  – int GetPackLength()
    Returns the size of the SpParmSet when it is packed into a buffer.

  – void PackBuff(char* buff)
    Causes the set to pack itself into a user-supplied byte array. The byte array must have enough space to hold the set, the size of which can be determined with the GetPackLength method.

  – void SetNative(),
    void SetNET_TYPES(),
    void SetXDR()
    These methods allow the caller to choose which packing method the SpParmSet will use when it flattens itself into a buffer. NET_TYPES tells the set to use SPEEDES NET_INTs and NET_DOUBLEs when packing numerical values. Native means that the numerical values are packed as is, which may cause problems if the SpParmSet is going to be unpacked on a computer with a different architecture. External Data Representation (XDR) tells the set to use SUN’s XDR encoding protocol.
3.27 Examples

Using SpParmSet’s [] operator is very similar to the retrieval methods detailed above. The caller passes in an integer or string key (whichever is appropriate for the set), and a reference to the corresponding SpParmSetElement is returned. The SpParmSetElement class has conversion methods for all the supported data types. So, the following two statements are identical:

```c
int value1 = set.GetInt("key");
int value2 = set["key"];```

Alternatively, the caller can use the [] operator to insert data into the set. If an unknown key is passed in, a new SpParmSetElement is created and inserted into the set. The SpParmSetElement class has the = operator overloaded for all the supported data types. So, the following two statements are identical:

```c
set.InsertInt("key", 5);
set["key"] = 5;
```

Data of type, buffer, are a special case, since they need an extra size parameter. In this case, you must use the GetBuffer and SetBuffer methods from the SpParmSetElement class:

```c
set["key"].SetBuffer(buff, size);
buff = set["key"].GetBuffer(size);
```

3.27.3 See also

SpBinaryTree 3.13 and SpSet 3.31.

3.27.4 Used in

SpDefineHandler.H 2.7.

3.27.5 Notes

SpParmSet uses a balanced SpBinaryTree to store its elements.
3.28 SpParser: SPEEDES Parser

3.28.1 Description

SpParser is a powerful parser that reads in .par files, which can be used to configure your simulation via text files rather than source code modifications. Only the API and a brief discussion of the parameter file format is shown below. Usage of the SpParser is described in the SPEEDES User’s Guide.

Parameter File Format

Here is an example of a simple parameter file, demonstrating many of the basic features of this parser:

```plaintext
// The "//" string indicates the start of a comment in a parameter file.
Set_0 {  // Set 0 definition
    int SetZerosDefaultValue 6  // Integer definition
}

Set_1 {  // Set 1 definition
    Set_4 {  // Set 4 definition
        inherit Set_0  // This set also contains set 0
        int SetZerosDefaultValue 17  // Overrides SetZerosDefaultValue
        int setFoursInt 6  // Integer definition
        Set_6 {  // Set 6 definition
            include "SetSixIncludedValues.par"  // Set 6 contains all of the definitions found in SetSixIncludedValues.par
            reference Set_2  // Reference to non embedded set
            string myString "hello"
                "Good bye"  // String array definition
            int x 1 2 3  // Integer array definition
            float double1 6.4 1e12 17  // Float array definition
            logical boolean1 f F t T  // Boolean array definition
        }
    }

Set_2 {  // Set 2 definition
    string string1 StringsWithOutSpacesDoNotNeedQuotes  // String entry with one element
    string string2 "String" "Array" "Requires" "Quotes"  // String array definition
}
```

The general format consists of named sets enclosed by curly braces ("{" and "}") which contain the data. Keywords include “int, float, logical, string, and float” for the primitive types of integers, booleans, strings, and floating point numbers. There are the additional keywords of “inherit, reference, and include”. These keywords allow for:

1. Sets to inherit values from previously defined sets.
2. Sets to include (reference) sets that are defined later.
3.28 SpParser: SPEEDES Parser

3. Sets to include values from other parameter files. Note that sets that include other parameter files cannot be base sets for inheritance.

Values of parent sets can be overridden by simply restating the inherited item along with its new value. This is demonstrated in the above example when Set\textsubscript{4} overrides the value of SetZerosDefaultValue which was inherited from Set\textsubscript{0}.

SpParser API

- **SpSet** Parse(SpSet* set = NULL)
  Parse the file or stream and return the top level set. Parse can be called multiple times and will always return the set returned from the first call to Parse. If a set is passed as an argument, the data parsed from the file or stream is added to that set. Once the set is obtained, all the utilities from the SpSet class (see Section 3.31 for more information) can be used to extract the information from the parameter file.

- **SpParser** (istream* inflie)
  Creates a SpParser from an istream.

- **SpParser** (char* filename)
  Creates a SpParser from a file.

3.28.2 See also

SpSet 3.31.
3.29 SpPriorityTree and RB_SpPriorityTree: Binary Tree Based Priority Queue

3.29.1 Description

The SpPriorityTree and RB_SpPriorityTree are classes that wrap the standard SpBinaryTree and RB_SpBinaryTree to provide functionality similar to that of a priority queue.

- `int GetNumElements()`
  Returns the number of elements in the tree.

- `SpRetractionHandle* Insert(void* item, double priority)`
  Inserts item using the key, keyNum. Use of integer keys is handled by this call, as well, through C++'s type promotion from integer to double. The return from the insert is a retraction handle, so that the insert can be retracted at a later point in time.

- `void* Remove()`
  Removes and returns the item with the lowest priority.

- `void* Remove(double& priority)`
  Removes and returns the item with the lowest priority. The priority of the item removed is returned in the field priority.

- `int Retract(RetractionHandle* retractionHandle)`
  Retracts an item that was previously inserted. Returns non-zero on success and 0 on failure.

- The `RB_SpBinaryTree` and the `SpBinaryTree` support several modes of operation relating to the way the tree is balanced.
  - `NormalTreeMode` does not ever attempt to balance the tree (default mode). If data is to be inserted or removed from the tree in a completely random order, this method will perform very well.
  - `BalancedTreeMode` does not keep the tree perfectly balanced but uses a heuristic to balance the tree whenever it becomes too unbalanced. This will help maintain a relatively well-balanced tree if the data is well ordered when it is inserted into the tree or if the data is always removed from one end of the tree.
  - `SplayTreeMode` implements a splay tree algorithm. Whenever an item is inserted into the tree, that item is rotated to the root of the tree. Similarly, when an item is removed, it is first rotated to the root of the tree and then removed. Finally, whenever an item is found in the tree through one of the Find calls, it is rotated to the root of the tree. This algorithm has a similar effect as does a “caching” algorithm. Whenever an item is accessed, the items close to it are moved into a position where they can more quickly be accessed as well. If the items in the tree are inserted, removed, or accessed in a completely random fashion, this mode will yield poor performance.

- The methods for changing and examining the balancing heuristic are:
  - `void SetBalancedTreeMode()`
    Sets the trees to balanced tree mode.
  - `void SetNormalTreeMode()`
    Sets the trees to normal tree mode.
- **void** SetSplayTreeMode()
  Sets the trees to splay tree mode.

- **int** IsBalancedTreeMode()
  Returns 1 if the trees are in balanced tree mode, 0 otherwise.

- **int** IsNormalTreeMode()
  Returns 1 if the trees are in normal tree mode, 0 otherwise.

- **int** IsSplayTreeMode()
  Returns 1 if the trees are in splay tree mode, 0 otherwise.
3.30 SpRandom and RB_SpRandom: Random Number Generators

3.30.1 Description

SpRandom and RB_SpRandom provide a rollbackable and a non-rollbackable version of a powerful random number generator, as well as several different useful distributions.

- **double Erf(double x)**
  Generates the erf (area under Gaussian curve from 0 to x) function for a value of x.

- **double GenerateBeta(int n1, int n2)**
  Generates a random number using the distribution:
  \[ f(x) = \frac{(n_1 + n_2 + 1)!}{n_1!n_2!}[t_0^1(1 - t_0^n)]. \]

- **int GenerateBit()**
  Generates a single random bit.

- **int GenerateBits(int numBits)**
  Generates a given number of random bits.

- **double GenerateCauchy(double alpha)**
  Generates a Cauchy distribution.

- **double GenerateDensityFunction(SpDensityFunction* densityFunction)**
  Generates a random number using a user-defined density function and the rejection method. The density function must inherit from the following base class:

  ```
  class SpDensityFunction {
  public:
    virtual double f(double x) = 0; // Function at x
    virtual double F(double x) = 0; // Function at x
    virtual double GetMaxAmplitude() = 0; // Max amplitude of function
    virtual double GetLoLimit() = 0; // Low x value of function
    virtual double GetHiLimit() = 0; // High x value of function
  protected:
    private:
  }
  ```

- **double GenerateDouble(double lo = 0.0, double hi = 1.0)**
  Generates a random double in the range [lo, hi].

- **double GenerateExponential(double timeConstant)**
  Generates a random number using an exponential distribution:
  \[ f(t) = ce^{-t/c}. \]

- **int GenerateInt(int lo, int hi)**
  Generates a random int in [lo, hi].

- **double GenerateGaussian(double mean, double sigma)**
  Generates a Gaussian distribution which is described by:
  \[ f(x) = \frac{1}{\sqrt{2\pi}\sigma}e^{-\frac{(x-\mu)^2}{2\sigma^2}}. \]
double GenerateLaplace(double timeConstant)
Generate a random double using a Laplace distribution (same as exponential but the time constant
can be either positive or negative).

double GeneratePower(double power)
Generates a random number with the distribution:
\[ f(x) = (p + 1)x^p \text{ where } 0 \leq x \leq 1. \]

double GenerateRayleigh(double alpha)
Generates a random number using a Rayleigh distribution:
\[ f(x) = \frac{x}{\alpha^2} e^{-\frac{x^2}{2\alpha^2}} \text{ where } 0 \leq x. \]

double GenerateReversePower(double power)
Generates a random number with the distribution:
\[ f(x) = (p + 1)(1 - x)^p \text{ where } 0 \leq x \leq 1. \]

double GenerateTriangleDown()
Generates a random number using the distribution:
\[ f(x) = 2(1 - x) \text{ where } 0 \leq x \leq 1. \]

double GenerateTriangleUp()
Generates a random number using the distribution:
\[ f(x) = 2x \text{ where } 0 \leq x \leq 1. \]

void GenerateVector(double vector[3], double magnitude = 1.0)
Generates a random vector with the given magnitude.

double GenerateUniform()
Generates a double uniform int [0, 1].

void GenerateVector(double vector[3], double meanMagnitude, double sigmaMagnitude)
Generates a random vector with given mean and standard deviation on the magnitude.

int GetSeed()
Gets the seed for the random number generator.

double InvErf(double erfx)
Generates the inverse erf value (x for which erf(x) = erfx).

void SetSeed()
Sets the seed for the random number generator.

3.30.2 Bugs
A seed of MIN_INT is not valid.

3.30.3 Notes
RB_SpFastRandom and SpFastRandom are classes with the same API. These classes provide faster
random number computations at the expense of less randomness in the numbers.
3.31 SpSet: Streamable Set of Sets And Elements

3.31.1 Description

Set syntax consists of an identifier representing the set name, followed by a braced set of elements (or followed by nothing if a reference). Elements that comprise sets include nested sets, inherited sets, referenced sets, integers, integer arrays, floats, float arrays, logicals (booleans), logical arrays, strings, string arrays, enums, and user defines. Any element type, which includes a name (or user-defined keyword) and a value, including a set, may be dynamically added to any set or nested set, by simply calling methods on that set.

In fact, SpSet inherits from SpSetElement, which is exactly how nested sets are implemented.

- **SpSet(char* buffer)**
  Reconstructs a set from a character buffer returned from the pack method.

- **SpSet* Embed(char* name)**
  Preferred method for embedding one set inside another.

- Accessor Methods. The first of each of the following methods will return the value identified by key, if present, and set the status flag to non-zero and return the default, if not. The second of each of these methods will return the value identified by key, if available. Otherwise, it prints a message to stderr and aborts.
  
  - int GetInt(char* key)
  
  - int GetInt(char* key, int defaultValue, int& status)
  
  - double GetFloat(char* key)
  
  - double GetFloat(char* key, double defaultValue, int& status)
  
  - bool GetLogical(char* key)
  
  - bool GetLogical(char* key, bool defaultValue, int& status)
  
  - char* GetString(char* key)
    Do not delete returned value.

- Array accessor methods. These methods return a pointer to private data for the sake of performance. Do not free it! The second of each of them returns also the length of the array. If no array corresponding to “key” exists, each returns NULL.
  
  - SpSetArray* GetArray(char* key)
  
  - double* GetFloatArray(char* key)

  - double* GetFloatArray(char* key, int& length)
3.31 SpSet: Streamable Set of Sets And Elements

- `int* GetIntArray(char* key)`
- `int* GetIntArray(char* key, int& length)`
- `bool* GetLogicalArray(char* key)`
- `bool* GetLogicalArray(char* key, int& length)`
- `char** GetStringArray(char* key)`
- `char** GetStringArray(char* key, int& length)`
- `void* GetVoidPtr(char* key)`

- **SpSet* GetSet(char* name)**
  Method for retrieving an embedded or referenced set.

- **SpSetElement* find_in_scope(char* key)**
  Finds the element named key anywhere in the scope of this set. Returns NULL if not found.

- **Methods for packing a set and all its dependencies into a buffer suitable for sending in messages or saving off to disk.**
  - `char* Pack()`
    Allocates a buffer and returns a buffer to the allocated space.
  - `char* Pack(char* buffer)`
    Packs set into this buffer.
  - `char* Pack(char* buffer, SpSet* top)`
    Allocates a buffer and returns a buffer to the allocated space. The top argument leaves any references visible in the scope of this set (which are, therefore, not defined in this set but somewhere under top) undefined.
  - `char* Pack(int& bufferLength)`
    Allocates a buffer and returns a buffer to the allocated space. The size of the buffer is returned in bufferLength.
  - `char* Pack(int& bufferLength, char* buffer)`
    Packs set into buffer and returns number of bytes written in bufferLength.
  - `char* Pack(int& bufferLength, char* buffer, SpSet* top)`
    Packs set into buffer and returns a buffer to the allocated space. The top argument leaves any references visible in the scope of this set (which are therefore not defined in this set but somewhere under top) undefined. Number of bytes packed is returned in bufferLength.
  - **SpSetElement* Restore(char* buffer)**
    Returns a buffer that has been restored from a buffer.
  - `int Size()`
    Returns size of set when packed into a buffer.
– **int Size(SpSet* top)**
  Returns size of set when packed into a buffer. Generally, you should never need to pass an argument for top. It is only necessary if you want the size to be calculated based on packing up only unresolved references to elements not defined under this set, but which are defined somewhere under top.

- **SpSetArray& AddArray(char* key, ElementType type, int arrayLength)**
  Adds an array of length, arrayLength, key, and ElementType, type. ElementType can be one of:
  - **ARRAY_ET**
    Array type.
  - **ENUM_ET**
    Enumerated type.
  - **FLOAT_ET**
    Floating point type.
  - **INT_ET**
    Integer type.
  - **LOGICAL_ET**
    Logical type.
  - **NONE_ET**
    No type.
  - **SET_ET**
    Set type.
  - **STRING_ET**
    String type.
  - **UNRES_ET**
    Unresolved type.
  - **USER_DEFINED_ET**
    User-defined type.
  - **VOIDPTR_ET**
    Void pointer type.

- **void AddRef(char* key)**
  Adds a reference to the first element identified by key in the scope of this set.

- **SpSetElement* Clone()**
  Clones this set as a new top-level set (references in the scope of this set will get deep-copied).

- **SpSet* GetAncestor(char* setName)**
  Finds the ancestor set with the name setName. If none can be found, returns NULL.

- **SpSet* GetAncestor(int numLevels = 1)**
  Returns the ancestor that is numLevels up or NULL if there are not that many levels.

- **SpSet* GetParent()**
  Return the parent of this set. If this set is a top level set, returns NULL.

- **void Insert(bool value, char* key)**
  Inserts a boolean value with key.
3.31 SpSet: Streamable Set of Sets And Elements

- **void Insert(char* value, char* key)**
  Inserts a char* string value with key.

- **void Insert(double value, char* key)**
  Inserts a double value with key.

- **void Insert(int value, char* key)**
  Inserts an int value with key.

- **SpSet* Insert(SpSet* value)**
  Inserts a reference to a set visible within the scope of this set.

- **void Insert(void* value, char* key, int length)**
  Inserts a void* value with size of length with key.

- **void Remove(char* key)**
  Removes the element named key. If NULL, destroys self.

- **SpSet* Root()**
  Returns the top level set for this set.

- **SpSet* Union(SpSet* otherSet)**
  Forms the union of this set with otherSet.

- **Set iterators.** The basic rule for looping through either sets, user-defines, or all elements is that only one loop can be operating at a time. This means you cannot be calling GetFirstSet, GetNextSet, etc. intermixed with calling GoToFirstUserDefine, GoToNextUserDefine, etc. or intermixed with calling GetFirstElement, GetNextElement, etc. This limitation will be alleviated once iterators for this class are developed.

  - **SpSet* GetFirstSet()**
    Returns first set or reference.

  - **SpSet* GetNextSet()**
    Returns next set or reference.

  - **int GetNumSets()**
    Returns number of sets or references.

- **int GetEnumIntID(char* key, char* enumString)**
  Returns the enum element integer id for which the string equivalent of one of the enum elements is known.

- **int GetEnumIntID(char* key, char* enumString, int defaultValue, int& status)**
  Returns the enum element integer id for which the string equivalent of one of the enum elements is known. If the key is not present, status will be set to 0 and, if found, set to 1.

- **char* GetEnumString(char* name, int intId)**
  Returns the string equivalent for which the enum element integer id of one of the enum elements is known. Do not delete the returned value.

- **int GetNumElements()**
  Gets the number of elements in this set.
3.31.2 See also

SpParmSet 3.27.
3.32 SpStopWatch: Stopwatch Timer Class

3.32.1 Description

Class for single and/or cumulative timing of any time intervals in the simulation. Timing always includes both wall (actual/real/total) time and clock (CPU) time. Use objects of this class to time any aspects of the simulation. A SpStopWatch can be named during construction, reset, and repeatedly started and stopped. If not reset, they only keep track of cumulative time. Either clock or wall time can be set as the default of what is returned and/or printed, but SpStopWatches always keep track of both. Thus, the flag for clock or wall time returning and printing can be changed at any time during the cumulative timing.

- **get_CumulativeClockTime()**
  The first time StartTimer and StopTimer are called after object construction or after calling Reset, this class begins cumulative timing from zero for both wall timing and clock timing. Otherwise, it continues accumulating both wall and clock time from past StartTimer/StopTimer intervals. Regardless of the Clock Flag, the class cumulatively stores both types of time.

- **double get_CumulativeTime()**
  The first time StartTimer and StopTimer are called after object construction or after calling Reset, this class begins cumulative timing from zero for both wall timing and clock timing. Otherwise, it continues accumulating both wall and clock time from past StartTimer/StopTimer intervals. Regardless of the Clock Flag, the class cumulatively stores both types of time.

- **const char* get_name()**
  Returns the name of this stop watch.

- **void Reset()**
  Resets cumulative wall time and and clock time The first time StartTimer and StopTimer are called after object construction or after calling Reset, this class begins cumulative timing from zero for both wall timing and clock timing. Otherwise, it continues accumulating both wall and clock time from past StartTimer/StopTimer intervals. Regardless of the Clock Flag, the class cumulatively stores both types of time.

- **double StopTimer()**
  Stops timing the interval since the last StartTimer call, stops cumulative wall time and clock time, and returns the time value of the time interval since the last StartTimer call. This returns the time interval beginning with the most recent StartTimer call. The time value returned depends on the current setting of the Clock Flag (the default is wall time), which must be set previously, but can be set before or after the most recent StartTimer call with the methods UseWallTime and UseClockTime. The first time StartTimer and StopTimer are called after object construction or after calling Reset, this class begins cumulative timing from zero for both wall timing and clock timing. Otherwise, it continues accumulating both wall and clock time from past StartTimer/StopTimer intervals. Regardless of the Clock Flag, the class cumulatively stores both types of time.

- **void UseClockTime()**
  The clock system call is used for the clock (CPU) timer.

- **void UseConstant()**
  All StopTimer calls return 1.0.

- **void StartTimer()**
  Begins timing a new time interval for either wall time or clock time, and continues cumulative
wall time and clock time. Objects of this type will return the time interval beginning with this call when the next StopTimer is called. The time value returned depends on the setting of the Clock Flag (the default is wall time) when StopTimer is called, which can be set before or after calling StartTimer with the methods, UseWallTime and UseClockTime. The first time this method is called after object construction or after calling Reset, this method begins cumulative timing from zero for both wall timing and clock timing. Otherwise, it continues accumulating both wall and clock time from past StartTimer/StopTimer intervals. Regardless of the Clock Flag, the class cumulatively stores both types of time.

- **void UseWallTime()**
  The gettimeofday system call is used for the wall (actual/real/total) timer.

### 3.32.2 Notes

If the corresponding C++ file is compiled on a Silicon Graphics, Incorporated computer with the macro SGL_HARDWARE_TIMER defined, the built-in hardware timer is used when UseWallTime is called. This hardware counter wraps unexpectedly and can result in incorrect results. However, it does have a significantly higher resolution, does not “drift” like the gettimeofday call, and has lower overhead.
3.33 SpString80: Fixed-Length String Class

3.33.1 Description

The SpString80 is a character array that is 80 bytes long. Since the typed arguments for those events should be of a fixed size, the SpString80 can be useful for interface events (also known as typed events).

- **char* operator =(char* stringValue)**
  The first 80 bytes of the supplied string are copied into the class’ internal string value.

- **operator char*()**
  Conversion operator. Returns a pointer to the internal string value, which should not be altered in any way.

- **SpString80()**
  Default constructor. The string value is initialized to be an empty string.

- **SpString80(const char* stringValue)**
  Constructor with an initial value. The first 80 bytes of the supplied string are copied into the class’ internal string value.

3.33.2 See also

3.34 SpTag: Trace File Debug Class

3.34.1 Description

When an event is processed, a SpTag can be obtained through the call, SpGetTag (defined in SpGlobalFunctions.H), and data can be added to the tag. This data will be added to the trace file if tracing is enabled. The class, SpTag, supports standard “put to” operators and behaves like normal stream classes.

- `osfstream & operator << (char* val)`
  Adds the string to the tag.
- `osfstream & operator << (double val)`
  Adds the double to the tag.
- `osfstream & operator << (float val)`
  Adds the float to the tag.
- `osfstream & operator << (int val)`
  Adds the integer to the tag.
- `osfstream & operator << (long val)`
  Adds the long value to the tag.
- `osfstream & operator << (short val)`
  Adds the short value to the tag.
- `osfstream & operator << (unsigned int val)`
  Adds the unsigned integer to the tag.
- `osfstream & operator << (unsigned long val)`
  Adds the unsigned long value to the tag.
- `osfstream & operator << (unsigned short val)`
  Adds the unsigned short value to the tag.
- `osfstream & operator << (void* p)`
  Adds the pointer to the tag.

3.34.2 Examples

During any event, one can call SpGetTag to access the tag and add data to it. For example:

```cpp
class S_Ship::ProcessDetonation(double size, double distance) {
    int killed = 0;
    if (rand() > RAND_MAX * 0.9) {
        killed = 1;
        SCHEDULE_CheckForDamage(SpGetTime(), SpGetObjHandle());
    }
    if (DEBUG) {
        SpTag& currTag = *SpGetTag();
        currTag << "S_Ship::ProcessDetonation size: " << size
          << " distance: " << distance;
```
3.34 SpTag: Trace File Debug Class

```cpp
if (killed) {
    currTag << " Killed, scheduled check for damage" << endl;
} else {
    currTag << " Not Killed" << endl;
}
```

3.34.3 Bugs

The SpTag can contain a maximum of 256 characters of information. If more is needed, change the `#define` at the top of SpTag.H to a larger value and recompile. SpTag does not have direct support for manipulators or other operators, such as `endl`. To work around this problem, stack operators, putting the manipulators or other operators after one of the allowed operators. For Example:

```cpp
SpTag* myTag = SpGetTag();
(*myTag) << endl; // Will not compile
(*myTag) << " " << endl; // Will compile as operator<< (char*)
// returns an ostrstream&
3.35 **C_SQ_ITEM: Base Class for Items in RB_hash and RB_queue (obsolete)**

### 3.35.1 Description

Classes must inherit from this class to be inserted into RB queues or C_QUEUES. This class is simply a generic queue item that can be sorted according to simulated time tags by a RB queue (doubly-linked) or C_QUEUE (singly-linked). They can also be used in a RB queue or a C_QUEUE used as a stack. That is, when consistently popping from the top of the queue (which is the only way to pop in a singly linked C_QUEUE), pushing can be to the top or the bottom of the queue. When consistently pushing to the top, the RB_queue or C_QUEUE acts as a stack. When consistently pushing to the bottom, the RB_queue or C_QUEUE acts as a queue. Pushing effects are vice-versa when consistently popping from the bottom (rather than the top) of the queue (for RB queues or C_DBL_QUEUEs only). Sorting and inserting operations always put the low time value on top so that consecutive pops off the top of the sorted queue or stack occurs in simulated time order.

- **C_SQ_ITEM* get_backlink()**
  Gets the back link for this C_SQ_ITEM (i.e. the link to the previous item in the doubly-linked RB_queue data structure).

- **void set_backlink(C_SQ_ITEM* bl)**
  Sets the back link for this C_SQ_ITEM (i.e. the link to the previous item in the doubly-linked RB_queue data structure).

### 3.35.2 Examples

The C_SQ_ITEM is often noticed as a base class to the F_SpProxyItem. F_SpProxyItems are the holders of proxies on the remote proxy list of an S_SpHLA class (see Section 4.1 for more information). Iterating through this list requires use of one of the above methods or the methods C_ITEM (see section 3.2). For example, the following shows how to search through the list of remote proxies in reverse (bottom to top) order:

```cpp
#include "S_SpHLA.H"
#include "F_SpProxyItem.H"
#include "RB_queue"

class S_Ship : public S_SpHLA {
public:
    S_Ship();
    void IterateThroughProxies() {
        RB_queue* list = GetRemoteObjectProxies();
        int len = list->get_length();
        F_SpProxyItem* pItem = (F_SpProxyItem*)list->get_bot();
        for (int i = 0; i < len; ++i) {
            ProcessProxy(pItem->GetProxy());
            pItem = (F_SpProxyItem*) pItem->get_backlink();
        }
    }
private:
```

void ProcessProxy(SpObjProxy* proxy);

3.35.3 See also

RB_queue 3.9, RB_hash 3.7, and C.ITEM 3.2.
Chapter 4

Object Management (proxies)
4.1 **F_SpProxyItem: Rollbackable Reference to an Object Proxy**

### 4.1.1 Description

The F_SpProxyItem class is used in SPEEDES’s DDM as a reference to a simulation object proxy. It inherits from C_SQ_ITEM, which allows F_SpProxyItems to be inserted into a RB_queue.

- **virtual char* GetName()**
  Returns the proxy name of the object proxy to which the F_SpProxyItem has a reference.

- **int GetNode()**
  Returns the Nodeld of the simulation object to which the F_SpProxyItem’s proxy corresponds.

- **SpObjProxy* GetObjProxy()**
  Returns a pointer to the object proxy to which the F_SpProxyItem has a reference.

- **int GetSimObjGlobalId()**
  Returns the SimObjGlobalId of the simulation object to which the F_SpProxyItem’s proxy corresponds.

- **int GetSimObjLocalId()**
  Returns the SimObjLocalId of the simulation object to which the F_SpProxyItem’s proxy corresponds.

- **int GetSimObjMgrId()**
  Returns the SimObjMgrId of the simulation object to which the F_SpProxyItem’s proxy corresponds.

- **void SetObjProxy(SpObjProxy* proxy)**
  Assigns the F_SpProxyItem’s reference to the given object proxy.

### 4.1.2 See also

SpSimObj 2.15, S_SpHLA 4.33, SpObjProxy 4.35, RB_queue 3.9, and C_SQ_ITEM 3.35.
4.2 BASE_ATTRIBUTE: Object Proxy Attribute Base Class

4.2.1 Description

BASE_ATTRIBUTE is the class from which all ATTRIBUTE (proxyable) classes are derived. This means *ATTRIBUTE are the objects within a simulation object which have the ability to be monitored by other classes through object proxies. These objects are inherently rollbackable. Users should never use this class directly, but should use the derived classes such as INTATTRIBUTE, DYNAMICATTRIBUTE, STRING_ATTRIBUTE, etc.

In general, you should treat attribute objects just like you would treat any rollbackable members. They are fully rollbackable and have the added feature of being publishable. The two main derived types of the BASE_ATTRIBUTE class are *ATTRIBUTE and DYNAMIC_{* ATTRIBUTE}. The *ATTRIBUTE are attributes that have a single non-time dependent value. DYNAMIC_{* ATTRIBUTE} types have values at different points in time. The static attributes’ (i.e. non-dynamic) values are accessible through simple accessor methods. The dynamic attributes’ values are accessible through accessor methods, which require input of specified data retrieval time.

Most of the public methods should never be called, but it is at least instructive to describe what some of them do:

- **voidScheduleUpdate( ),
  voidScheduleUpdate(ADD_REMOVE, ATTRIBUTE_LIST_ELEMENT)**
  These methods are called whenever an attribute changes its values. This indicates to the SPEEDES framework that the new attribute values need to be packaged up and sent to all objects that are subscribed to this object and ensures that all updates performed during an event are sent as one update.

- **int AttributeIndex**
  This protected data member represents the index of this attribute within Objects.par. For example, if the Objects.par had the form:

  ```
  MovingEntity {
    define dynamic_position Position
  }
  
  Ship {
    reference INHERIT MovingEntity
    define float Vmax
  }
  ```

  then the AttributeIndex of Position is 0 and the AttributeIndex of Vmax is 1. This information can be useful in debugging crashes due to malformed Objects.par files.

- **int GetAttributeIndex()**
  Returns AttributeIndex.

4.2.2 Bugs

The *ATTRIBUTE classes should never be instantiated as temporary variables. These classes should be treated just like rollbackable classes and used only to maintain the state of simulation objects.
4.2.3 See also

4.3 BASE_DYNAMIC_ATTRIBUTE: Base Class for Dynamic Attributes

4.3.1 Description

BASE_DYNAMIC_ATTRIBUTE inherits from LIST_ATTRIBUTE (see Section 4.26) and extends this class to represent time related data. A dynamic type is a type which, rather than having a single static value, may have different values at different times. There are many possible applications for this, including using different equations of motion to represent different waypoints in a motion or dynamic integers to represent the available quantity of a given resource.

This has great value in a discrete-event simulation, as it reduces the number of times that an attribute must be changed. Since each change of an attribute requires that all subscribers be notified of the change, being able to provide a prediction of what the values will be can greatly reduce the number of events and rollbacks that will occur.

There are a number of methods for accessing the dynamic items that are on the BASE_DYNAMIC_ATTRIBUTE:

- **SpDynItem* FindDynamicItem(double t)**
  Since the BASE_DYNAMIC_ATTRIBUTE class inherits from the LIST_ATTRIBUTE type, it has a list which can store a variety of DYNAMIC_ATTRIBUTEs. Each of these attributes has a start time and an end time. These are the times for which a dynamic item is considered to be valid and this returns the one that is valid for the passed in time. If there is no item in the list that is valid for the passed in time, an error message is printed and NULL is returned. Be careful to pass in a double value! Otherwise, this method could get confused with the method (non-user method), FindDynamicItem(int), which performs a different service.

- **double GetEndT ime()**
  Returns the largest time for any item in the list.

- **double GetStartT ime()**
  Returns the smallest time for any item in the list.

- **void GetTimeInterval(double& startT ime, double& endT ime)**
  Returns the start and end time set for the dynamic attribute. The start time is the smallest time for any item in the list and the end time is the largest time for any item in the list.

- **int GetTimeInterval(double t, double& startT ime, double& endT ime)**
  Finds the start and end time for the segment that crosses t. Returns 1 if an item spanning t is found. Returns 0 otherwise.

- **int ValidLastRef()**
  Returns 1 if the last evaluation had a valid time (i.e. there is a dynamic item whose interval includes the time). Returns 0 otherwise.

- **void operator +=(SpDynItem* v)**
  Adds the item, v, to the bottom of the list and reflects the attributes to all the proxies. Dynamic items added should always be obtained through a call to FreeDynamicAttributes—>new_object or its rollbackable equivalent.

- **void operator -= (SpDynItem* v)**
  Removes the item, v, from the list. If v is not found, no error message is returned, but an update of the proxy is still broadcast to all subscribers.
4.3.2 See also

BASE ATTRIBUTE 4.2, LIST ATTRIBUTE 4.26, and SpFreeDynamicAttributes 4.34.
4.4 BINARY_BUFFER_ATTRIBUTE: Binary Buffer Attribute

4.4.1 Description

A BINARY_BUFFER_ATTRIBUTE is a publishable raw buffer attribute for use in object proxies. If you want to publish non-time dependent raw binary data about your object, this is the class to use. Whenever the data in this class is updated, then the updated data is sent to all objects which subscribed to this object.

- **CopyIntoBuffer(char* srcBuff, int srcBytes)**
  Assigns the buffer located at srcBuff with size, srcBytes, to the BINARY_BUFFER_ATTRIBUTE. This is the only method available for changing the value of the BINARY_BUFFER_ATTRIBUTE.

- **const char* GetBuffPtr() const**
  Returns the char* string attribute value for the BINARY_BUFFER_ATTRIBUTE.

- **int GetBuffSize()**
  Returns the size of the buffer in bytes.

- **int operator ==(const char* buff)**
  Compares the passed in buffer to the attribute. Returns 1 if they are equal, 0 otherwise. Note that length of the argument, buff, is not passed in nor checked. This operator will go away in a later version and will be replaced with an “equals” method.

- **int operator ==(BINARY_BUFFER_ATTRIBUTE& rhs)**
  Compares the two buffer attributes for equality. Returns 1 if they are equal, 0 otherwise.

- **int operator !=(const char* a)**
  Compares the passed in string, a, to the attribute. Returns 1 if they are not equal, 0 if they are equal, but does not check length.

4.4.2 Examples

This class works well when data needs to be exchanged between simulation objects but it must be exchanged either in a proprietary format, such as XDR, or perhaps encrypted for security purposes. Complex data structures could be exchanged by streaming from the publisher and unstreaming at the recipient. Consider the following example:

```c++
// S_Ship.H
#include "S_BaseEntity.H"
#include "SpExportAttribute.H"

// Here, S_BaseEntity inherits from S_SpHLA and adds in the
// methods char* EncryptMessage(const char* msg).
class S_Ship : public S_BaseEntity {
    public:
        S_Ship() : S_BaseEntity("S_Ship") {}  
        void Init();
        void SetMessage(const char* message);

    private:
```
BINARY_BUFFER_ATTRIBUTE EncryptedData;
};

// S_Ship.C

void S_Ship::Init() {
// Here, the Objects.par description does not state that the
// message is encrypted or even what is the format of the
// encryption. It only states that it is a message. This
// class
DEFINE_ATTRIBUTE(EncryptedData, "Message");
int msgSize;
char* encryptedMessage = EncryptMessage("", msgSize);
EncryptedData.CopyIntoBuffer(encryptedMessage, msgSize);
}

4.4.3 Bugs

- The == and != operators do not check the length of the passed in argument when compared to strings.

4.4.4 See also

BASE_ATTRIBUTE 4.2.
CIRCULAR ORBIT: Circular Orbit Dynamic Position

CIRCULAR ORBIT creates a very simple circular orbit about the earth moving from West to East. The gravitational constant is assumed to be 398601.3 meters/second² and the object is allowed to orbit at any altitude (even negative altitudes do not cause an error). The API is similarly small and follows the same pattern as other dynamic position classes:

- **Initialization method.**
  - `init(double lat, double lon, double alt, double phase, double t)`
    Initializes the CIRCULAR ORBIT with latitude, lat, longitude, lon, and altitude, alt (units are radians and kilometers) at time, t, with phase, phase. phase is the number of radians the object has moved along the path described by the motion at time, t. Start time is set to −1.0e20 and the end time is initialized to 1.0e20 (use the parent class SetStartTime(double) and SetEndTime(double) to change these). The coordinate system is initialized with an undefined state so the parent classes, SetEARTH, SetECI or SetECR, must be called to set the appropriate coordinate system.

- **Methods for evaluating positions.**
  - `void operator () (double t, int pt, double x[3])`
    Returns the position in x for the time, t, and the coordinate system, pt. pt is one of ECR, ECI, or EARTH, and the conversion is done using the functions in Section 3.16.
  - `void operator () (double t, int pt, double x[3], double v[3])`
    Returns the position in x and the velocity in v for the time, t, and the coordinate system, pt. pt is one of ECR, ECI, or EARTH, and the conversion is done using the functions in Section 3.16.
  - `void operator () (double t, int pt, double x[3], double v[3], double a[3])`
    Returns the position in x, the velocity in v, and the acceleration in a for the time, t, and the coordinate system, pt. pt is one of ECR, ECI, or EARTH, and the conversion is done using the functions in Section 3.16.

**4.5.1 See also**

4.6 CONSTANT_MOTION: Constant Position Motion Dynamic Position

4.6.1 Description

CONSTANT_MOTION is a dynamic position class for defining a constant position. It may seem odd to think of a dynamic position as being constant but this is, in fact, a very natural situation. Consider a simulation of an airport. At any point in time, there are a number of airplanes that are stationed at a constant position (the gate, repair, hanger, …) for an extended period of time until their position is changed by either taking off or being shuttled around the airport.

CONSTANT_MOTION also makes an excellent choice for extending an equation of motion to cover all periods of time from $-\infty$ to $+\infty$. This is often required in order to ensure that dynamic position values always return valid information for all valid input times, such as for DDM or other models.

CONSTANT_MOTION has a very simple API to correspond to its simple function:

- **Methods for changing the position.**
  - `void SetPosition(double x0, double x1, double x2)`
    Sets the location of the constant position to (x0, x1, x1). There are no methods for setting the coordinate system, so the default coordinate system is undefined. Therefore, one of SetEARTH, SetECI, or SetECR (inherited from DYNAMIC_POSITION_ITEM) should be called before adding this item to a DYNAMIC_POSITION_ATTRIBUTE. Similarly, the start time and end time are initialized to -1.0e20 and should be set using the SetStartTime(double time) and SetEndTime(double time) methods that are inherited from SpDynItem.
  - `void SetPosition(double x[3])`
    Sets the location of the constant position to (x[0], x[1], x[2]).

- **Accessor methods.**
  - `void operator () (double t, int pt, double x[3])`
    Returns the position in x for the given time, t, and the given position type, pt (i.e. ECR, ECI, or EARTH in Section 3.16).
  - `void operator () (double t, int pt, double x[3], double v[3])`
    Returns the position in x and the velocity in v (velocity is always 0 for the given time, t, and the given position type, pt (i.e. ECR, ECI, or EARTH in Section 3.16)).
  - `void operator () (double t, int pt, double x[3], double v[3], double a[3])`
    Returns the position in x, and the velocity and acceleration in v and a (velocity and acceleration are always 0 for the given time, t, and the given position type, pt (i.e. ECR, ECI, or EARTH in Section 3.16)).

4.6.2 See also

4.7 DOUBLE_ATTRIBUTE: Double Attribute

4.7.1 Description

DOUBLE_ATTRIBUTE behaves just like an RB<double with the additional feature that any changes are reflected to all objects subscribed to this object. To this end, a DOUBLE_ATTRIBUTE can be treated just like a normal double.

- **operator double() const**
  Returns the double value of the DOUBLE_ATTRIBUTE.

- **double operator =(double rhs)**
  Assigns the value, rhs, to the DOUBLE_ATTRIBUTE.

- **double operator +=(double a)**
  Increments the value of the DOUBLE_ATTRIBUTE by a and returns the new value.

- **double operator -= (double a)**
  Decrements the value of the DOUBLE_ATTRIBUTE by a and returns the new value.

- **double operator *= (double a)**
  Multiplies the value of the DOUBLE_ATTRIBUTE by a and returns the new value.

- **double operator /= (double a)**
  Divides the value of the DOUBLE_ATTRIBUTE by a and returns the new value.

4.7.2 See also

BASE_ATTRIBUTE 4.2 and RB<double 3.4.
4.8 DYNAMIC_COMPLEX_EXPONENTIAL: Complex Exponential Dynamic Double

4.8.1 Description

This yields an equation of motion in one dimension that follows the path of a complex exponential. A complex exponential is of the form:

\[ y = ae^{i\omega(t-t_0)} \]

where \( a \) is an amplitude, \( \omega \) is a parameter, \( t_0 \) is the time offset, \( t \) is the value at which we are evaluating, and \( i \) and \( e \) are the standard mathematical constants \( \sqrt{-1} \) and \( 2.718281828\ldots \). An interesting result of this is that:

\[ ae^{i\omega(t-t_0)} = a[\cos(\omega(t-t_0)) + i\sin(\omega(t-t_0))] \]

This means that if RealFlag is set through the method, SetRealFlag, we return the real portion of the equation, \( a \cos(\omega(t-t_0)) \). Otherwise, we return the imaginary portion, \( a \sin(\omega(t-t_0)) \).

A mathematical curiosity that arises from these equations is known as Euler’s formula: \( e^{ix} - 1 = 0 \). This equation is significant, since it involves the five most important numbers in all of mathematics in a single short equation.

As with other DYNAMIC_DOUBLE_ITEMS, an item of this type is obtained from the free list, modified, and added to a DYNAMIC_DOUBLE_ATTRIBUTE. The API for this item is:

- **Methods to set the attribute.**
  - `void Set(double a, double o, double t)`
    Sets the amplitude to \( a \), \( \omega \) to \( o \), and the time offset to \( t \). The defaults are 0, 0, and 0.
  - `void SetAmplitude(double a)`
    Sets the amplitude to \( a \).
  - `void SetFrequency(double f)`
    Sets the frequency to \( f \). That is, set \( \omega = 2\pi f \).
  - `void SetImag()`
    Sets the attribute so the imaginary portion is returned. This setting is not reflected.
  - `void SetOmega(double o)`
    Sets \( \omega \) to \( o \).
  - `void SetPeriod(double t)`
    Sets the period to \( t \). That is, set \( \omega = 2\pi / t \).
  - `void SetReal()`
    Sets the attribute so the real portion is returned. This setting is not reflected and real mode is the default setting.
  - `void SetTimeOffset(double t)`
    Sets the value of the time offset to \( t \).

- **Accessor methods.**
  - `double GetAmplitude()`
    Returns the amplitude.
– double GetFrequency()
  Returns the frequency \((2\pi/\omega)\).

– double GetOmega()
  Returns \(\omega\).

– double GetPeriod()
  Returns the period \((\omega/(2\pi))\).

– int GetRealFlag()
  Returns whether the dynamic complex exponential returns the real portion or the imaginary portion. If real is set, 1 is returned, otherwise, 0.

– double GetTimeOffset()
  Returns the time offset.

– double operator () (double t)
  Returns the real or imaginary part of the complex exponential at time, \(t - \text{StartTime}\), as specified by SetReal or SetImag.

– double operator () (double t, double& v)
  Returns the real or imaginary part of the complex exponential at time, \(t - \text{StartTime}\), and sets \(v\) to the first derivative (velocity), as specified by SetReal or SetImag.

– double operator () (double t, double& v, double& a)
  Returns the real or imaginary part of the complex exponential at time, \(t - \text{StartTime}\), sets \(v\) to the first derivative (velocity), and sets \(a\) to the second derivative (acceleration), as specified by SetReal or SetImag.

4.8.2 See also

BASE_ATTRIBUTE 4.2 and DYNAMIC_DOUBLE_ATTRIBUTE 4.9.
4.9 DYNAMIC_DOUBLE_ATTRIBUTE: Dynamic Double Attribute

4.9.1 Description

The DYNAMIC_DOUBLE_ATTRIBUTE is a dynamic attribute that can be used to produce piecewise defined functions. For example, if we wanted an attribute to represent the function:

\[ f(t) = \begin{cases} 
  t & t < 0 \\ 
  t^2 & 0 < t < 10 \\ 
  e^t & t > 10 
\end{cases} \]

then this exact functionality could be obtained by using a DYNAMIC_DOUBLE_ATTRIBUTE and using three separate functions, one to represent each interval. The API for a DYNAMIC_DOUBLE_ATTRIBUTE is relatively straightforward:

- **double operator () (double t)**
  Evaluates the piecewise defined function that is valid for time, t, and returns the value at that point. If a function is not defined for t, -1.0e20 is returned and a call to ValidLastRef (see Section 4.3) will return 1.

- **double operator () (double t, double& v)**
  Evaluates the piecewise defined function that is valid for time, t, and returns the value at that point. The parameter, v, is filled in with the first derivative (velocity) at t. If a function is not defined for t, -1.0e20 is returned, v is set to -1.0e20, and a call to ValidLastRef (see Section 4.3) will return 1.

- **double operator () (double t, double& v, double& a)**
  Evaluates the piecewise defined function that is valid for time, t, and returns the value at that point. The parameter, v, is filled in with the first derivative (velocity) and a is filled in with the second derivative (acceleration) at t. If a function is not defined for t, -1.0e20 is returned, v and a are set to -1.0e20, and a call to ValidLastRef (see Section 4.3) will return 1.

4.9.2 Examples

Here is a short example of how to initialize and/or modify a dynamic double attribute.

```c
// S_Ship.H
#include "S_SpHLA.H"
#include "SpExportAttribute.H"

class S_Ship : public S_SpHLA {
public:
    S_Ship():S_SpHLA("S_Ship") {}
    void Init();
    void PunctureFuelTank();

private:
    DYNAMIC_DOUBLE_ATTRIBUTE FuelLevel;
};

// S_Ship.C
```
4.9 DYNAMIC_DOUBLE_ATTRIBUTE: Dynamic Double Attribute

```cpp
#include "SpFreeDynamicAttributes.H"
#include "SpExtrapolate.H"
#include "SpConstant.H"
#include "RB_SpFrameworkFuncs.H"

void S_Ship::Init() {
    // Always get dynamic objects from FreeDynamicAttributes
    DYNAMIC_EXTRAPOLATE* newEquation = (DYNAMIC_EXTRAPOLATE*)
        RB_FREE_NEW(FreeDynamicAttributes, DYNAMIC_EXTRAPOLATE_ID);

    DEFINE_ATTRIBUTE(FuelLevel, "FuelLevel");
    // Fuel tank starts with 1000 liters and will be empty at
    // time 3600 when the simulation ends.
    newEquation->Init(0, 3600, 1000, 1000 / 3600);
    FuelLevel += newEquation;
}

void S_Ship::PunctureFuelTank() {
    double currFuelLevel = FuelLevel(SpGetTime());
    double newRate = (1000 / 3600) + 3; // Fuel leaks at 3 liters/sec
    double emptyAtTime = currFuelLevel/newRate;
    DYNAMIC_DOUBLE_CONSTANT* constant = (DYNAMIC_DOUBLE_CONSTANT*)
        RB_FREE_NEW(FreeDynamicAttributes, DYNAMIC_DOUBLE_CONSTANT_ID);

    // Remove the first (and only) item from the equation
    newEquation = FuelLevel.GetFirstElement();
    newEquation -= newEquation;

    // Initialize the equation with the new rate
    newEquation->Init(SpGetTime(), emptyAtTime,
                      currFuelLevel, newRate);
    // Reinsert it in the list (one cannot just change it while
    // it is on the list)
    FuelLevel += newEquation;

    // Now add on another equation indicating that the tank is
    // empty for the rest of the simulation.
    constant->Set(0);

    // The following are from SpDynItem, the base dynamic item class
    constant->SetStartTime(emptyAtTime);
    constant->SetEndTime(3600);
    FuelLevel += constant;
}

4.9.3 Bugs

Behavior at the endpoints of dynamic attributes is undetermined. It is recommended that the piecewise
functions be made continuous at the endpoints or that evaluation is not performed at the endpoints.

4.9.4 See also

4.9.5 Notes

The list of potential equations that can be added to a DYNAMIC_DOUBLE_ATTRIBUTE include: DYNAMIC_COMPLEX_EXPONENTIAL, DYNAMIC_DOUBLE_CONSTANT, DYNAMIC_EXPONENTIAL, DYNAMIC_POLY_N, DYNAMIC_SPLINE_3, DYNAMIC_SPLINE_6, DYNAMIC_EXTRAPO-LATE, and SpFreeDynAttributes.
4.10 DYNAMIC_DOUBLE_CONSTANT: Constant Dynamic Double

4.10.1 Description

A DYNAMIC_DOUBLE_CONSTANT represents the function that maintains a constant value. It can be used to provide endpoint values to ensure that a DYNAMIC_DOUBLE_ATTRIBUTE has valid values for all values of time or can be used to create step functions. As this is a simple function, the API is similarly simple:

- Only one method to set.
  - void Set(double v)
    Sets the constant value to v.

- Methods for accessing.
  - double operator () (double)
    Returns the constant value.
  - double operator () (double, double& v)
    Returns the constant value and sets v to 0.0.
  - double operator () (double, double& v, double& a)
    Returns the constant value and sets both v and a to 0.0.

4.10.2 See also

BASE_ATTRIBUTE 4.2 and DYNAMIC_DOUBLE_ATTRIBUTE 4.9.
4.11 DYNAMIC_EXPONENTIAL: Exponential Equation Dynamic Double

4.11.1 Description

This yields an equation of motion in one dimension that follows the path of an exponential. The equation that it uses is

\[ y = ae^{\lambda(t-T_0)} \]

where \( a \), \( \lambda \), and \( T_0 \) are constants and \( e \approx 2.718281828 \ldots \) \( a \) is the amplitude, \( \lambda \) is just a constant multiplier of the variable (it can be thought of as a horizontal stretch/shrinking of the graph), and \( T_0 \) is a phase shift (horizontal shift) of the function and is set to the value from parent class method, GetStartTime, and is set by the parent class method, SetStartTime(double t).

- Methods for setting the item.
  - `void Set(double amp, double lambda)`
    Sets the amplitude to amp and the value for \( \lambda \) to lambda. The defaults are 0.0 and 1.0, which means that the initial setting is the zero function.
  - `void SetAmplitude(double a)`
    Sets Amplitude to a.
  - `void SetTimeConstant(double t)`
    Sets \( \lambda \) to \( 1.0/t \). This ensures that \( f(t) = a \) at time, t.

- Methods for accessing the item.
  - `double GetAmplitude()`
    Returns the amplitude.
  - `double GetTimeConstant()`
    Returns the value of t for which \( f(t) = a \).
  - `double operator () (double t)`
    Returns the value of the exponential at time, \( t - \text{StartTime} \).
  - `double operator () (double t, double& v)`
    Returns the value of the exponential at time, \( t - \text{StartTime} \), and sets v to the first derivative (Velocity) at \( t - \text{StartTime} \).
  - `double operator () (double t, double& v, double& a)`
    Returns the value of the exponential at time, \( t - \text{StartTime} \), sets v to the first derivative (velocity) and sets a to the second derivative (acceleration) at \( t - \text{StartTime} \).

4.11.2 See also

BASEATTRIBUTE 4.2 and DYNAMIC_DOUBLE_ATTRIBUTE 4.9.
4.12 DYNAMIC_EXTRAPOLATE: Extrapolation Dynamic Double

4.12.1 Description

DYNAMIC_EXTRAPOLATE can take an initial position and velocity (optionally the acceleration and the jerk [third derivative] as well), and then extrapolate the values as far as required. This is often called dead reckoning.

- Initializer method.
  - void Init(double ts, double te, double x0, double v0, double a0 = 0.0, double j0 = 0.0)
    Sets the start time to ts, the end time to te, the initial position to x0, initial velocity to v0, and, optionally, the initial acceleration and jerk to a0 and j0, respectively.

- Evaluation methods.
  - double operator () (double t)
    Extrapolates the initial conditions and returns the position at t-StartTime.
  - double operator () (double t, double& v)
    Extrapolates the initial conditions and returns the position at t-StartTime. The first derivative (velocity) is evaluated at the same time and returned in v.
  - double operator () (double t, double& v, double& a)
    Extrapolates the initial conditions and returns the position at t-StartTime. The first and second derivatives (velocity and acceleration) are evaluated at the same time and returned in v and a.

4.12.2 See also

BASE_ATTRIBUTE 4.2, SpFreeDynAttributes 4.34, and DYNAMIC_DOUBLE_ATTRIBUTE 4.9.
4.13 DYNAMIC_INT_ATTRIBUTE: Dynamic Integer Attribute

4.13.1 Description

The DYNAMIC_INT_ATTRIBUTE is a dynamic attribute that can be used to produce piecewise step functions. For example, if we wanted an attribute to represent the function:

\[ f(t) = \begin{cases} 
4 & t < 0 \\
5 & 0 < t < 10 \\
0 & t > 10 
\end{cases} \]

then this functionality could be obtained by using a DYNAMIC_INT_ATTRIBUTE and using three separate steps, one to represent each interval. The API for a DYNAMIC_INT_ATTRIBUTE is relatively straightforward:

- **int operator () (int t)**
  
  Evaluates the step function that is valid for time, t, and returns the value at t. If a function is not defined for t, -1.0e20 is returned and a call to ValidLastRef (see Section 4.3) will return 1.

4.13.2 Bugs

Behavior at the endpoints of dynamic attributes is undetermined. It is recommended that the piecewise functions be made continuous at the endpoints or that evaluation is not performed at the endpoints.

4.13.3 See also


4.13.4 Notes

There is currently only one dynamic int step function that can be added to this list: DYNAMIC_INT_CONSTANT in Section 4.14.
4.14 DYNAMIC_INT_CONSTANT: Constant Dynamic Integer

4.14.1 Description

DYNAMIC_INT_CONSTANTs are the only dynamic items that are currently supported with a DYNAMIC_INT_ATTRIBUTE (see Section 4.13). The DYNAMIC_INT_CONSTANT is quite simple.

- Modification methods.
  
  - int operator = (int v)
    
    Assigns the value, v, to the dynamic int constant.
  
  - void Set(int v)
    
    Assigns the value, v, to the dynamic int constant.

- Accessor method.

  - int operator () (double)
    
    Returns the value of the dynamic int item.

4.14.2 See also

4.15 DYNAMIC_LOGICAL_ATTRIB: Dynamic Logical Attribute

4.15.1 Description

The DYNAMIC_LOGICAL_ATTRIB is a dynamic attribute that can be used to produce predefined, changing logical values. The API for a DYNAMIC_LOGICAL_ATTRIB is relatively straightforward:

- **int operator () (logical t)**
  Evaluates the logical function that is valid for time, t, and returns the value at t. If a function is not defined for t, -1 is returned and a call to ValidLastRef (see Section 4.3) will return 1.

4.15.2 Bugs

Behavior at the endpoints of dynamic attributes is undetermined. It is recommended that the piecewise functions be made continuous at the endpoints or that evaluation is not performed at the endpoints.

4.15.3 See also

BASE_ATTRIB 4.2, LIST_ATTRIB 4.26, and BASE_DYNAMIC_ATTRIB 4.3.
4.16 DYNAMIC_LOGICAL_CONSTANT: Constant Dynamic Logical

4.16.1 Description

DYNAMIC_LOGICAL_CONSTANTs are the only dynamic items that are currently supported with a DYNAMIC_LOGICAL_ATTRIBUTE (see Section 4.15). The DYNAMIC_LOGICAL_CONSTANT is quite simple.

- Modification methods.
  - int operator = (int v)
    Assigns the value, v, to the dynamic logical constant.
  - void Set(int v)
    Assigns the value, v, to the dynamic logical constant.

- Accessor method.
  - int operator () (double)
    Returns the value of the dynamic logical item.

4.16.2 See also

BASE_ATTRIBUTE 4.2 and DYNAMIC_LOGICAL_ATTRIBUTE 4.15.
4.17 DYNAMIC_POLY_N: Polynomial Equation Dynamic Double

4.17.1 Description

There are variants of this class that range from DYNAMIC_POLY_1 through DYNAMIC_POLY_10 and provide an n-degree polynomial that best fits the given points. If the user provides more than n points, this class will construct the best fit polynomial through those points. The user is allowed to specify sigma squared (the standard deviation squared) for each of the points to specify how close a fit is desired to match those points.

While every possible precaution is taken, the user should be warned that the system of equations that must be solved can be notoriously ill conditioned for larger values of n (greater than 8 or 9). Every precaution is taken to try to solve this system, but it still can give incorrect values.

It should also be noted that high degree polynomials often have wild oscillations, even when the data appears to be smooth. For this reason, rather than using a high degree (greater than 6) polynomial, it may be wiser to use a spline.

- Methods for setting a dynamic polynomial
  - void AddPoints(double ts, double te, double t, double v, double err2, ...)
    AddPoints can take any number of arguments equal to 2 plus a multiple of 3. For example, if the user wanted the line that went through the points (0, 1) and (2, 7) from t=0 through 3, he would first use the child class DYNAMIC_POLY_1 and then would call:
    AddPoints(0.0, 3.0, 0.0, 1.0, 17.0, 2.0, 7.0, 17.0, END_POLY);
    Notice the extra number “17.0” that is in the equation. This is supposed to be the standard deviation (squared) allowed for each point in the line fit. Since we are giving the exact number of points required to fit a line, these values are ignored. This list of points is always terminated with the argument “END_POLY”
    Suppose the user wanted to find the best fit quadratic to the points (1, 2), (3, 7) (4, 12), and (8, 6) from t=0 to t=13. It is also desired that the line should be a better fit for the first, third, and fourth numbers than for the second. In this case, use the child class, DYNAMIC_POLY_2, and would call the method:
    AddPoints(0, 13,
    1, 2, 0.1,
    3, 7, 10,
    4, 12, 0.1,
    8, 6, 0.1,
    END_POLY)
    Since the sigma squared is smaller in the first, third, and fourth points, the line will fit those points better, while it will be a worse fit for the second point.
    Three important points of note. First, you must always terminate the argument list with END_POLY. This is the signal to the framework that you have entered your last point. Second, sigma squared (err2) must never be 0 or negative. 0 will result in either NaN’s or a core dump, and negative numbers can result in garbage results.
  - void Reset()
    Frees the work areas and removes all points that were added. This does not undo the polynomial that you have created but allows you to recreate it with new points.
– void MakePoly()
  Generates the polynomial for the points you entered. If too few points have been added to
  generate the polynomial you desire, an error message is printed.

• Methods for examining or evaluating the polynomial.

  – NET_DOUBLE* GetCoeff()
    Returns the coefficients of the polynomial.

  – int GetDim()
    Returns the degree of the polynomial.

  – double operator () (double t)
    Returns the value of the polynomial at time, t - StartTime.

  – double operator () (double t, double& v)
    Returns the value of the polynomial at time, t - StartTime. v is set to the first derivative
    (velocity) at that time.

  – double operator () (double t, double& v, double& a)
    Returns the value of the polynomial at time t - StartTime. v is set to the first derivative
    (velocity) and a is set to the second derivative (acceleration) at that time.

4.17.2 See also

BASE_ATTRIBUTE 4.2, and DYNAMIC_DOUBLE_ATTRIBUTE 4.9.
4.18 DYNAMICPOSITION_ATTRIBUTE: Dynamic Position Attribute

4.18.1 Description

The DYNAMICPOSITION_ATTRIBUTE is a dynamic attribute that can be used to produce predefined, changing equations of motions. This class inherits from BASE DYNAMIC_ATTRIBUTE and many of the methods needed for day-to-day use can be found in that class (Section 4.3). The API for a DYNAMICPOSITION_ATTRIBUTE is relatively straightforward:

- **int CheckMotion()**
  Performs a cursory check for realistic derivative values for each equation of motion. A finite difference calculation is performed and then compared against the actual derivative values to check for numbers within reason. The return value is 0 for no errors and is incremented by 1 for every error found. Note that it only checks individual segments and does not check for derivatives across different equations of motion.

- **int CheckTime()**
  Once the user has defined a dynamic position attribute, it is important that it provides a position for every time between its start time and its end time. If it is not continuous, it prints an error message and returns a non-zero status. It does not check to make sure that the velocity and acceleration are continuous. Returns 0 for no errors in continuity and increments by 1 for every gap in continuity.

- **int operator () (double t, int pt, double x[3])**
  Evaluates the equation of motion that is valid for time, t, and sets x to the value at t for the coordinate system, pt, which is one of ECR, ECI, or EARTH (coordinate transformations are done using the functions in Section 3.16 with the t value passed for ECI conversions). If a function is not defined for t, -1 is returned and a call to ValidLastRef (see Section 4.3) will return 1.

- **int operator () (double t, int pt, double x[3], double v[3])**
  Evaluates the equation of motion that is valid for time, t, sets x to the value and v to the velocity at t for the coordinate system pt which is one of ECR, ECI, or Earth (coordinate transformations are done using the functions in Section 3.16 with the t value passed for ECI conversions). If a function is not defined for t, -1 is returned and a call to ValidLastRef (see Section 4.3) will return 1.

- **int operator () (double t, int pt, double x[3], double v[3], double a[3])**
  Evaluates the equation of motion that is valid for time, t, and sets x to the value at t for the coordinate system, pt, which is one of ECR, ECI, or Earth (coordinate transformations are done using the functions in Section 3.16 with the t value passed for ECI conversions). If a function is not defined for t, -1 is returned and a call to ValidLastRef (see Section 4.3) will return 1.

4.18.2 Examples

Dynamic position attributes come in a wide variety and are described in Section 4.18.5.

Here is a short example of how to add equations of motion to a dynamic position attribute:
/ S_Ship.H
#include "S_SpHLA.H"
#include "SpExportAttribute.H"

class S_Ship : public S_SpHLA {
public:
    S_Ship() : S_SpHLA("S_Ship") {}
    void Init();
    void ChangeMotion();

private:
    DYNAMIC_POSITION_ATTRIBUTE MyPosition;
};

// S_Ship.C
#include "SpFreeDynamicAttributes.H"
#include "SpExtrapolateMotion.H"
#include "SpGreatCircle.H"
#include "RB_SpFrameworkFuncs.H"

void S_Ship::Init() {
    // Always get motion objects from FreeDynamicAttributes

    newMotion = (EXTRAPOLATE_MOTION *)
        RB_FREE_NEW(FreeDynamicAttributes, EXTRAPOLATE_MOTION_ID);

double startPos[3] = {0, 0, 0} // Lat, Lon, Alt

double startVel[3] = {0, 0, 0} // Velocity of above WRT time

double startAcc[3] = {0, 0, 0.1} // Acceleration of pos WRT time

double startJerk[3] = {0, 0, -0.0003} // Jerk of pos WRT time

    newMotion->Init(0, 3600, startPos, startVel,
        startAcc, startJerk, EARTH);
    MyMotion += newMotion;
}

void S_Ship::ChangeMotion(){
    double currItemStart;
    double currItemEnd;
    SpDynItem* currItem = MyMotion.FindDynamicItem(SpGetTime());

double endPos[3];
double currPos[3];
double currVel[3];

    // If there is an item on the list for the current time,
    if (MyMotion.GetTimeInterval(SpGetTime(),
        currItemStart, currItemEnd)) {
        // Compute positions now and at end of equation of motion
        MyMotion(SpGetTime(), EARTH, currPos, currVel);
        MyMotion(currItemEnd, EARTH, endPos);
        // Remove equation of motion.
        MyMotion -= currItem;

        // Rollbackably return to free list so that memory is not lost
        // or misused in some other fashion.
        RB_FREE_DELETE(FreeDynamicAttributes, currItem);

        newMotion = (GREAT_CIRCLE *)
            RB_FREE_NEW(FreeDynamicAttributes, GREAT_CIRCLE_ID);
double speed = sqrt(currVel[0] * currVel[0] +
currVel[1] * currVel[1] +
currVel[2] * currVel[2])
newMotion->init_Vconstant(currPos[0], currPos[1], currPos[2],
                      SpGetTime(),
                      endPos[0], endPos[2], endPos[3],
                      speed);

4.18.3 Bugs

Behavior at the endpoints of dynamic attributes is undetermined. It is recommended that the piecewise
functions be made continuous at the endpoints or that evaluation is not performed at the endpoints.

4.18.4 See also


4.18.5 Notes

The equations of motion that can be added to a dynamic attribute are: SPLINE5_MOTION, SPLINE3-
_MOTION, RHUMB_LINE, POLY_N_MOTION, LOITER_MOTION, GREAT_CIRCLE, EXTRAPOLATE_MOTION,
Elliptical, CONSTANT_MOTION, and CIRCULAR_ORBIT.
4.19 **DYNAMIC_SPLINE_3: Cubic Spline Equation Dynamic Double**

### 4.19.1 Description

A DYNAMIC_SPLINE_3 is a third order spline that fits a position and velocity at an initial and final time. As opposed to DYNAMIC_POLY_N (see Section 4.17), this is an exact fit and does not perform any least-squares fitting.

- **Initializer method.**

  ```
  void init(double t0, double x0, double v0, double t1, double x1, double v1)
  
  Fits the cubic to the initial time, t0, initial position, x0, initial velocity, v0, and final time, t1, final position, x1, and final velocity, v1.
  ```

- **Evaluation methods.**

  ```
  double operator () (double t)
  
  Evaluates the spline at time, t-StartTime, and returns the value.
  
  double operator () (double t, double& v)
  
  Evaluates the spline at time, t-StartTime, and returns the value. The first derivative (velocity) is returned in v and is evaluated at the same time.
  
  double operator () (double t, double& v, double& a)
  
  Evaluates the spline at time, t-StartTime, and returns the value. The first derivative (velocity) is returned in v and second derivative (acceleration) is returned in a. Both are evaluated at the same time.
  ```

### 4.19.2 See also

BASE_ATTRIBUTE 4.2, DYNAMIC_DOUBLE_ATTRIBUTE 4.9, and DYNAMIC_SPLINE_6 4.20.
4.20 DYNAMIC_SPLINE_6: Quintic (fifth order) Spline Equation Dynamic Double

4.20.1 Description

A DYNAMIC_SPLINE_6 is a fifth order spline that fits a position, velocity, and acceleration at an initial and final time. Unlike DYNAMIC_POLYN (see Section 4.17), this is an exact fit and does not perform any least squares fit.

- Initializer method.
  - void init(double t0,
    double x0,
    double v0,
    double a0,
    double t1,
    double x1,
    double v1,
    double a1)
    Fits a fifth order spline to the initial time, t0, initial position, x0, and initial velocity, v0, and final time, t1, final position, x1, and final velocity, v1.

- Evaluation methods.
  - double operator () (double t)
    Evaluates the spline at time, t-StartTime, and returns the value.
  - double operator () (double t, double& v)
    Evaluates the spline at time, t-StartTime, and returns the value. The first derivative (velocity) is returned in v and is evaluated at the same time.
  - double operator () (double t, double& v, double& a)
    Evaluates the spline at time, t-StartTime, and returns the value. The first derivative (velocity) is returned in v and second derivative (acceleration) is returned in a. Both are evaluated at the same time.

4.20.2 See also

BASE_ATTRIBUTE 4.2, DYNAMIC_DOUBLE_ATTRIBUTE 4.9, and DYNAMIC_SPLINE_3 4.19.
4.21 Elliptical: Elliptical Orbits Dynamic Position

4.21 Elliptical: Elliptical Orbits Dynamic Position

4.21.1 Description

Elliptical is an equation of motion that follows the elliptical path of an object in orbit about another object fixed in space. This is generally used to compute the path of an object that is orbiting about the earth. In fact, the standard initialization method assumes an orbit about the earth, although there exist methods to set the mass of the earth and the radius of the earth.

As with other dynamic positions, generally, an Elliptical equation of motion is obtained from the free list, SpFreeDynAttributes, initializers are called, and then the equation is added to a DYNAMIC_POSITION_ATTRIBUTE. This is then reflected through all of the proxies.

Many methods exist for setting and examining the value of the equation of motion. These are:

- Methods for modifying the values of an Elliptical.
  - `void SetRadiusEarth(double re)`
    Sets the radius of the earth to re. The default value for the radius of the earth is 6378.145 meters.
  - `void SetMassEarth(double me)`
    Sets the mass of the earth to me. The default value for the mass of the earth is $5.9742 \times 10^{24}$ kg.
  - `void InitPosVel(double timeInit, int pt, double* x, double* v, double timeFinal = 1.0e20)`
    Initializes the elliptical equation with the given initial time, the position type, pt (one of ECR, ECI, or EARTH), the initial position, x, and the initial velocity. The default final time is 1e20 but can be set by this initialization method. Calling this method also initializes the impact position.

- Methods for examining the value of an Elliptical.
  - `double GetAngularVelocity()`
    Returns the average angular velocity for the equation of motion.
  - `double GetPeriod()`
    Returns the amount of time for one orbit of the elliptical object.
  - `void GetPosImpact(double x[3])`
    Fills in x with the value of the impact position for the elliptical orbit. If the object does not impact the earth, the value 0, 0, 0 is returned in the ECI coordinate system.
  - `void GetPosVelImpact(double x[3], double v[3])`
    Fills in x and v with the value of the position and velocity at impact. If impact is not achieved, position is 0, 0, 0.
  - `void GetPosVelAccImpact(double x[3], double v[3], double a[3])`
    Fills in x, v, and a with the position, velocity, and acceleration at impact. If impact is not achieved, position is 0, 0, 0.
– **double GetTimeImpact()**
  Returns the time of the impact. If the elliptical orbit does not impact, the time, 1e20, is returned.

– **virtual void operator () (double t, int pt, double x[3])**
  Returns the position, x, for the given time, t, and the coordinate system, pt (one of ECR, ECI, or EARTH). The conversion from the initial position type is performed using the conversion functions of Section 3.16.

– **virtual void operator () (double t, int pt, double x[3], double v[3])**
  Returns the position, x, and velocity, v, for the given time, t, and the coordinate system, pt (one of ECR, ECI, or EARTH).

– **virtual void operator () (double t, int pt, double x[3], double v[3], double a[3])**
  Returns the position, x, the velocity, v, and the acceleration, a, for the given time, t, and the coordinate system, pt (one of ECR, ECI, or EARTH).

**4.21.2 See also**

4.22 EXTRAPOLATE_MOTION: Dead Reckoning Dynamic Position

4.22.1 Description

Simulations often do not have a great deal of information about an object and this is often the case with federations. External federates (especially Distributed Interactive Simulation (DIS) federates) will often provide a position and velocity but no more information. It is then up to the simulation to propagate the position of this object until another update is provided.

This is an ideal situation for the EXTRAPOLATE_MOTION dynamic position type. EXTRAPOLATE_MOTION can be initialized with the position at a given time with as many as three derivatives, and it will propagate the position forward for as long as is necessary.

- **Initializer methods.**
  - `void Init(double ts, double te, double x0[3], double v0[3], int pt)`
    Initializes an EXTRAPOLATE_MOTION equation using a position, x0, and velocity, v0, that are valid at time, ts, for coordinate system, pt (one of EARTH, ECI, or ECR in Section 3.16). ts is the end time of this equation of motion.
  - `void Init(double ts, double te, double x0[3], double v0[3], double a0[3], int pt)`
    Initializes an EXTRAPOLATE_MOTION equation using a position, x0, velocity, v0, and acceleration, a0 that are valid at time, ts, for coordinate system, pt (one of EARTH, ECI, or ECR in Section 3.16). ts is the end time of this equation of motion.
  - `void Init(double ts, double te, double x0[3], double v0[3], double a0[3], double j0[3], int pt)`
    Initializes an EXTRAPOLATE_MOTION equation using a position, x0, velocity, v0, acceleration, a0, and jerk (third derivative), j0, that are valid at time, ts, for coordinate system, pt (one of EARTH, ECI, or ECR in Section 3.16). ts is the end time of this equation of motion.

- **Methods for evaluating the dead reckoner.**
  - `virtual void operator () (double t, int pt, double x[3])`
    Sets x to the position of the dead reckoner at time, t, for the coordinate system, pt, using the conversion functions from Section 3.16.
  - `virtual void operator () (double t, int pt, double x[3], double v[3])`
    Sets s to the position and v to the velocity for the dead reckoner at time, t, for the coordinate system, pt, using the conversion functions from Section 3.16.
  - `virtual void operator () (double t, int pt, double x[3], double v[3], double a[3])`
    Sets x to the position, v to the velocity and a to the acceleration for the dead reckoner at time, t, for the coordinate system, pt, using the conversion functions from Section 3.16.

4.22.2 See also

4.23 **SpFreeObjProxy: A Class for Managing Freelists of Proxies**

4.23.1 **Description**

The class, SpFreeObjProxy, is defined by SPEEDES but the implementation of the constructor must be implemented by users when proxies are used. The SpFreeObjProxy class inherits from C_FREE_LIST (see Section 3.19) and most of the functionality is seen there.

- **SpFreeObjProxy(int nTypes)**
  Constructor of an SpFreeObjProxy that must be implemented by users. nTypes is the number of classes identified from the Objects.par file.

4.23.2 **Examples**

The minimal required SpFreeObjectProxy constructor is actually quite small:

```cpp
#include "SpFreeObjectProxy.H"

SpFreeObjProxy::SpFreeObjProxy(int n) {
    set_ntypes(n);
}
```

This minimal constructor will allow the simulation to run but does not allow for user-defined object proxies. That is, every object proxy that is discovered will be given an empty SpObjProxy class. If there are any user-defined proxies, then a more extensive constructor must be implemented, as in the next example:

```cpp
#include "SpObjProxy.H"
#include "SpFreeObjProxy.H"

class MovingEntity : public SpObjProxy {
    //...
};

class MissileProxy : public MovingEntity {
    //...
};

class ShipProxy: public MovingEntity {
    //...
};

enum {
    MovingEntity_ID,
    MissileProxy_ID,
    ShipProxy_ID
};

void* NewMovingEntity(int n) { return new MovingEntity[n]; }
void* NewMissileProxy(int n) { return new MissileProxy[n]; }
void* NewShipProxy(int n) { return new ShipProxy[n]; }
```
SpFreeObjProxy::SpFreeObjProxy(int n) {
    set_ntypes(n);

    set_type("Missile", // Name from objects.par
        MissileProxy_ID, // Enumerated value from above
        NewMissileProxy, // Function to create proxies
        sizeof(MissileProxy), // Size of this proxy class
        10); // Number to create at once.

    set_type("Ship",
        ShipProxy_ID,
        NewShipProxy,
        sizeof(ShipProxy),
        10);

    set_type("MovingEntity",
        MovingEntity_ID,
        NewMovingEntity,
        sizeof(MovingEntity),
        10);
}

If any proxies are not defined in the SpFreeObjectProxy constructor, the inheritance chain from Objects.par will be followed to determine what sort of proxy to create. In the above example, suppose objects.par looked like this:

MovingEntity {
    //...
}

Ship {
    reference INHERIT MovingEntity
    //...
}

Missile {
    reference INHERIT MovingEntity
    //...
}

Aircraft {
    reference INHERIT MovingEntity
    //...
}

City {
    //...
}

In this example, if an Aircraft were created, SPEEDES would determine that no Aircraft proxy has been defined and would try to create a MovingEntity proxy for the user since that has been defined. City has also not been defined and, since it inherits from nothing, a blank SpObjProxy will be returned.
4.23.3 See also

C\_FREE\_LIST 3.19 and SpObjProxy 4.35.
Traditionally, we define great circle motion on the earth as motion on a plane that goes through the center of the earth. The motion is circular and has a radius equal to that of earth so that the object stays on the surface. For example, an object that follows the path of the equator or path of the line on the surface passing through the north and south poles has a great circle trajectory.

In this extension of great circle motion, we do not restrict the object’s motion to the surface. We allow the object to have a non-zero altitude that changes with a constant rate of speed. Allowing the altitude to vary translates into allowing the radius of the great circle to vary. An object with an extended great circle trajectory may travel above the surface, but its projection on the surface of the earth is that of traditional great circle.

### 4.24.1 Description

The methods of interest include:

- There are two methods to initialize a great circle equation of motion. The coordinate system is initialized with an undefined state so the parent classes, SetEARTH, SetECI, or SetECR, must be called to set the appropriate coordinate system.

  - **double init_Vconstant**(double latInit, double lonInit, double altInit, double timeInit, double latFin, double lonFin, double altFin, double v)
    
    Initializes the great circle with the initial latitude, longitude, altitude, and time, followed by the final latitude, longitude, altitude, and velocity for the flight.

  - **double init_Tconstant**(double latInit, double lonInit, double altInit, double timeInit, double latFin, double lonFin, double altFin)
    
    Initializes the equation for the given initial latitude, longitude, altitude, and time, as well as the final latitude, longitude, altitude, and time.

- As with other equations of motion, there are the standard overloaded parentheses operators for evaluation:

  - **virtual void operator ()**(double t, int pt, double x[3])
    
    Returns the position of the great circle motion in x for time, t, and position type, pt (one of ECI, ECR, or EARTH using the conversion functions in Section 3.16).
– virtual void operator () (double t, int pt, double x[3], double v[3])
  Returns the position of the great circle motion in x and the velocity in v for time, t, and position type, pt (one of ECI, ECR, or EARTH using the conversion functions in Section 3.16).

– virtual void operator () (double t, int pt, double x[3], double v[3], double a[3])
  Returns the position of the great circle motion in x, the velocity in v and the acceleration in a for time, t, and position type, pt (one of ECI, ECR, or EARTH using the conversion functions in Section 3.16).

4.24.2 See also

4.25 INT_ATTRIBUTE: Integer Attribute

4.25.1 Description

INT_ATTRIBUTE behaves just like an RB_int with the additional feature that any changes are reflected to all objects subscribed to this object. Therefore, an INT_ATTRIBUTE can be treated just like a normal int.

- **operator int() const**
  Returns the integer value of the INT_ATTRIBUTE.

- **int operator =(int rhs)**
  Assigns the value, rhs, to the INT_ATTRIBUTE.

- **int operator ++()**
  Prefix increment operator. Increments the value of the INT_ATTRIBUTE and returns the new value.

- **int operator ++(int)**
  Postfix increment operator. Increments the value of the INT_ATTRIBUTE by 1 and returns the pre-increment value.

- **int operator --()**
  Prefix Decrement operator. Decrements the value of the INT_ATTRIBUTE by 1 and returns the new value.

- **int operator --(int)**
  Postfix decrement operator. Decrements the value of the INT_ATTRIBUTE by 1 and returns the pre-decrement value.

- **int operator +=(int a)**
  Increments the value of the INT_ATTRIBUTE by a and returns the new value.

- **int operator -==(int a)**
  Decrements the value of the INT_ATTRIBUTE by a and returns the new value.

- **int operator *==(int a)**
  Multiplies the INT_ATTRIBUTE by the value a and returns the new value.

- **int operator /==(int a)**
  Divides the INT_ATTRIBUTE by a and returns the new value.

- **int operator %==(int a)**
  Assigns the INT_ATTRIBUTE the remainder when the attribute is divided by a and returns the new value.

- **int operator >>==(int a)**
  Right shifts the attribute by a and returns the new value.

- **int operator <<==(int a)**
  Left shifts the attribute by a and returns the new value.

- **int operator ^==(int a)**
  “Not’s” the INT_ATTRIBUTE with a and returns the new value.
• int operator &=(int a)
  “And’s” the INT_ATTRIB with a and returns the new value.

• int operator !=(int a)
  “Or’s” the INT_ATTRIB with a and returns the new value.

4.25.2 Bugs

On some ANSI compliant compilers, an INT_ATTRIB cannot be used to index an array. To work around this, assign the INT_ATTRIB to an ordinary int and use that to index the array.

4.25.3 See also

BASE_ATTRIB 4.2 and RB_int 3.6.
4.26 LIST_ATTRIBUTE: List Attributes

4.26.1 Description

The LIST_ATTRIBUTE is the only dynamic data structure supported within SPEEDES object proxies. LIST_ATTRIBUTES are used to support all of the dynamic ATTRIBUTE types, such as the DYNAMIC_POSITION_ATTRIBUTE, the DYNAMIC_DOUBLE_ATTRIBUTE, and others. All items that can be placed on a LIST_ATTRIBUTE must be derived from the base class, OBJECT_ATTRIBUTE. The LIST_ATTRIBUTE supports many of the same methods as the RB_SpList and SpList classes.

- **void* GetFirstElement()**
  Returns a pointer to the first element in the list. Returns NULL if the list is empty.

- **void* GetLastElement()**
  Returns a pointer to the last element in the list. Returns NULL if the list is empty.

- **int GetNumElements()**
  Returns the number of elements in the attribute.

- **OBJECT_ATTRIBUTE* operator() (char* n)**
  Returns the attribute that has the name, n. If none are found, NULL is returned.

- **void* operator ++()**
  Returns a pointer to the next element in the list. Returns NULL at the end of the list.

- **void* operator --()**
  Returns a pointer the the previous element of the list. Returns NULL once the start of the list is passed.

- **void operator +=(OBJECT_ATTRIBUTE* v)**
  Adds the OBJECT_ATTRIBUTE, v, to the bottom of the list.

- **void operator -=(OBJECT_ATTRIBUTE* v)**
  Removes the OBJECT_ATTRIBUTE, v, from the list. No error is given if the object is not in the list.

Iterating through the list using ++ and -- cannot be nested.

4.26.2 See also

4.27 LOGICAL_ATTRIBUTE: Logical Attribute

4.27.1 Description

The LOGICAL_ATTRIBUTE is intended to make boolean values available on the proxy. Changes to a LOGICAL_ATTRIBUTE are reflected to all objects subscribed to this object.

- **operator int() const**
  Returns the integer value of the LOGICAL_ATTRIBUTE. Since C++ does not differentiate between integers and booleans, LOGICAL_ATTRIBUTEs are simply implemented with integers.

- **logical operator =(int rhs)**
  Assigns the value, rhs, to the LOGICAL_ATTRIBUTE. Zero is considered to be false, while any non-zero value is considered to be true.

4.27.2 See also

BASE_ATTRIBUTE 4.2.
4.28 LOITER_MOTION: Circle about a Fixed Ground Point Dynamic Position

4.28.1 Description

LOITER_MOTION is an equation described by flying a circle at constant altitude about a fixed point on the surface of the earth. For example, this could describe the path of an airplane that has been instructed to circle the airport for a period of time before landing.

As with other equations of motion, a LOITER_MOTION is generally obtained from the SpFreeDyn-Attributes free list and then added to a DYNAMIC_POSITION ATTRIBUTE in order to be reflected to object proxies. It is added to the attribute only after calling the Init method, since, as with all other equations of motion, changing the value of the LOITER_MOTION while it is on the DYNAMIC_POSITION ATTRIBUTE will not result in updates being propagated to the object proxies.

Several methods exist for examining and modifying the state of the equation of motion:

- **Modifying methods.**
  - `void Init(double ts, double te, double x0, double x1, double x2, double r, double s, int pt)`
    
    Initializes the equation of motion with the given start time, ts, end time, te, and position about which to loiter, (x0, x1, and x2). r is the radius of the loiter motion, s is the speed, and pt is the position type (one of ECR, ECI, or EARTH). If a position type of EARTH is selected, the input points are converted to ECR and the stored position type becomes ECR.
    
    It is not possible to set the initial angle through which the object has moved in the given LOITER_MOTION. However, ts can be adjusted in order to achieve this same effect. Suppose the radius of the LOITER_MOTION is 5000 meters, the speed is 500 meters per second, the object starts 20% of the way around the route and has to travel this route from 900 to 1200 seconds. The object travels all the way around the circle in \( \frac{2 \pi}{500} \) seconds. This means it is 20% of the way around after 125.66 seconds. Therefore, if the equation of motion is initialized with ts equal to 900 - 125.66 = 774.336 seconds, the correct rotation of an additional 20% of the way around the LOITER_MOTION will be achieved.

  - `void Init(double ts, double te, double x[3], double r, double s, int pt)`
    
    Behaves identically to the prior Init method, except the center of the circle for the LOITER_MOTION is given in the triple x.

- **Evaluation methods.**
  - `virtual void operator () (double t, int pt, double x[3])`
    
    Returns the position of the object in x for time, t, and position type, pt (one of ECR, ECI, or EARTH) using the conversion functions in Section 3.16.

  - `virtual void operator () (double t, int pt, double x[3], double v[3])`
    
    Returns the position of the object in x and velocity of the object in v for time, t, and position type, pt (one of ECR, ECI, or EARTH) using the conversion functions in Section 3.16.
virtual void operator () (double t, int pt, double x[3], double v[3], double a[3])

Returns the position of the object in x, the velocity in v and the acceleration in a for time, t, and position type, pt (one of ECR, ECI, or EARTH) using the conversion functions in Section 3.16.

4.28.2 See also

4.29 OBJECT_ATTRIBUTE: Object Attributes

4.29.1 Description

An OBJECT_ATTRIBUTE is a mechanism for creating infinitely generic objects within proxies that can be as varied as desired by users. OBJECT_ATTRIBUTEs can contain other ATTRIBUTE classes and can even contain other OBJECT_ATTRIBUTEs. For example, suppose you had the following class:

class STANDARD_INFO {
public:
  STANDARD_INFO() {
    SetTime(NULL);
    SetStartTime(-INFINITY);
  }
  char* GetName();
  void SetName(char* Name);
  double GetStartTime();
  void SetStartTime(double startTime);
private:
  char* Name;
  double StartTime;
  double TempVariable;
};

This class needs to be distributed through the proxy. One method to do so would be to add all the basic information to the standard proxy. However, this does not support encapsulation; so, this class could be turned into a proxyable class through inheritance and the addition of some SPEEDES calls:

class STANDARD_INFO : public OBJECT_ATTRIBUTE {
public:
  STANDARD_INFO() {
    SetTime(NULL);
    SetStartTime(-INFINITY);
    DEFINE_ATTRIBUTE(StartTime, "StartTime");
    DEFINE_ATTRIBUTE(Name, "Name");
    SetClassName("STANDARD_INFO"); // Required
  }
  char* GetName();
  void SetName(char* Name);
  double GetStartTime();
  void SetStartTime(double startTime);
  virtual int GetSize() {return sizeof (*this);} // Required
private:
  STRING_ATTRIBUTE Name;
  DOUBLE_ATTRIBUTE StartTime;
  double TempVariable;
};

RB_DEFINE_CLASS(STANDARD_INFO); // Required for free lists
The major changes you may notice are to change standard variables to ATTRIBUTE variables, to inherit from OBJECT_ATTRIBUTE, and to add DEFINE_ATTRIBUTE calls to the constructor. These DEFINE_ATTRIBUTE calls allow SPEEDES to know what this class looks like and gives SPEEDES a way to know how to package this attribute for updates. Each class must also call SetClassName in the constructor. This class name must match the name within Objects.par. Finally, the virtual method, GetSize, must be implemented and needs to return the size of this class.

This new class is added to Objects.par just like any other simulation object.

```
// Objects.par
...
STANDARD_INFO {
    define string Name
    define double StartTime
}
```

There are a great number of public methods for the OBJECT_ATTRIBUTE class of which only a small number are for public consumption:

- **SpObjProxy* GetObjProxy()**
  Returns the object proxy with the OBJECT_ATTRIBUTE. Every OBJECT_ATTRIBUTE contains a SpObjProxy that actually holds the values of the proxied values. This SpObjProxy is obtained and then methods can be called on it to return values of the various attributes within that proxy.

- **void SetClassName(char* objectType, int createProxyWithAttribs = 0)**
  Sets the name of the objectAttribute which must match the type within Objects.par. The second optional argument will cause some additional work to be performed and can be left at the default value.

### 4.29.2 Examples

Consider the Objects.par:

```
TEST_OBJECT {
    define string TestObjectName
}
// Base class for all simulation objects.
ENTITY {
    define object TestObject
}
...
```

Along with the simulation code:

```
// Entity.H
#include "S_SpHLA.H"
#include "SpObjProxy.H"
class TEST_OBJECT : public OBJECT_ATTRIBUTE {
```
public:
    STRING_ATTRIBUTE TestObjectName;
    TEST_OBJECT() /*: OBJECT_ATTRIBUTE("TEST_OBJECT")*/ {
        SetClassName("TEST_OBJECT");
        DEFINE_ATTRIBUTE(TestObjectName, "TestObjectName");
    }
    voidSetName(char*n) {TestObjectName = n;}
    const char* GetName() {return TestObjectName;}
    int GetSize() {return sizeof(*this);}
};

class S_Entity : public S_SpHLA {
    public:
        S_Entity() : S_SpHLA("ENTITY") {
            DEFINE_ATTRIBUTE(TestObject, "TestObject");
        }
        S_Entity(char* type) : S_SpHLA(type) // For inheritance
            DEFINE_ATTRIBUTE(TestObject, "TestObject");
        }
        const char* GetTestObjectName(SpObjProxy* proxy) {
            OBJECT_ATTRIBUTE* obj =
                (OBJECT_ATTRIBUTE)proxy->GetAttribute("TestObject");
            return obj->GetObjProxy()->GetString("TestObjectName");
        }

    private:
        TEST_OBJECT TestObject;
        ...
};

4.29.3 See also

SpObjProxy 4.35 and BASE_ATTRIBUTE 4.2.
4.30 POLY_N_MOTION: Least Squares Fit Polynomial Dynamic Position

4.30.1 Description

There are ten classes inheriting from POLY_N_MOTION that represent differing degrees of polynomials for equations of motion. Each class implements the same virtual methods in POLY_N_MOTION to provide the same interface, despite the different degrees of the polynomial. The derived classes available are POLY_1_MOTION through POLY_10_MOTION. Further ones could be created by simply deriving from POLY_N_MOTION and overriding the virtual function GetDim. However, using higher dimensions than 10 will almost certainly result in an almost impossible to solve system of equations and is not recommended.

Each polynomial is created by adding at least one more point than the degree of the polynomial, calling MakePoly, and then adding the equation to a DYNAMIC_POSITION_ATTRIBUTE. If more than \( n + 1 \) points (where \( n \) is the degree of the polynomial) are added, a least squares calculation is performed to find the best fit polynomial for the given number of points. If too few points are added or too many co-linear points are added, an error message is printed during the MakePoly call and undefined results may be encountered.

Two primary methods are provided for modifying the state of the POLY_N_MOTION:

- **void AddPoints(double ts, double te, double t, double v0, double v1, double v2, double err2, ...)**
  
  This method adds points to be used in the calculation of the best fit equation of motion. ts and te are the start and end times for this equation of motion.

  This is always followed by a variable number (but always at least one) of sets of times, point triplets, and errors. These sets follow the form of the time, t, for which the point is valid, the point itself in v0, v1, and v1, and the standard deviation allowed for this point. The standard deviation can never be zero and using too many points with too low of a standard deviation will result in a difficult to solve equation and may result in errors even if enough points are provided.

  A single call to this method using a large number of points or many successive calls may be made, adding more points.

- **void MakePoly()**

  Once all points have been added, MakePoly must be called in order to determine the best fit polynomial for the given list of points. There are several situations under which this call can fail:

  - Too few points have been added. For an \( n^{th} \) degree polynomial, at least \( n + 1 \) points must be added.
  - Too many colinear or nearly colinear points are added. This can result in a singular system of equations that cannot be properly solved.
  - Too many points with a small standard deviation have been added. This results in an unstable system of equations to be solved, resulting in errors.
In all of these situations, an error message will be printed to the screen and unexpected results will occur when the equation is evaluated.

Four different methods are provided for evaluating the POLY_N_MOTION, as well as the 10 derived classes.

- **int GetDim()**
  Returns the dimension (degree) of the equation of motion.

- **void operator () (double t, int pt, double x[3])**
  Returns the position, x, for the given time, t-StartTime, and the coordinate system, pt (one of ECR, ECI, or EARTH). The conversion from the initial position type is performed using the conversion functions of Section 3.16.

- **void operator () (double t, int pt, double x[3], double v[3])**
  Returns the position, x, and velocity, v, for the given time, t-StartTime, and the coordinate system, pt (one of ECR, ECI, or EARTH). The conversion from the initial position type is performed using the conversion functions of Section 3.16.

- **void operator () (double t, int pt, double x[3], double v[3], double a[3])**
  Returns the position, x, the velocity, v, and the acceleration, a, for the given time, t-StartTime, and the coordinate system, pt (one of ECR, ECI, or EARTH). The conversion from the initial position type is performed using the conversion functions of Section 3.16.

### 4.30.2 Bugs

Higher dimensional polynomials can be difficult to calculate and can result in incorrect behavior or slightly incorrect equations.

### 4.30.3 See also

4.31 POSITION_ATTRIBUTE: Position Attributes

4.31.1 Description

A POSITION_ATTRIBUTE is a publishable triplet and position type for object proxies. If the user wishes to publish non-time dependent position data about your object, this is the class to use. Whenever this class is updated in value, the update is then sent to everyone subscribed to the object proxy. The default position type is EARTH (latitude, longitude, altitude). The built-in position types available are:

- EARTH: Latitude, longitude, altitude.
- ECR: X, Y, Z in a rotating earth coordinate system.
- ECI: X, Y, Z in an inertial coordinate system.

The methods available are:

- **int GetPositionType()**
  Returns the type of coordinate system that is currently in use.

- **void SetEARTH(double lat, double lon, double alt)**
  Sets the latitude, longitude, and altitude to the passed in values and also sets the position type to EARTH. This is not error checked so, if you are in EARTH coordinates, it will happily allow you to make your position 74922 radians longitude 87823 radians latitude, and an altitude of -569279 kilometers.

- **void SetECI(double x[3])**
  Sets a new position in an ECI coordinate system.

- **void SetECR(double x[3])**
  Sets a new position in an ECR coordinate system.

- **void GetEARTH(double& lat, double& lon, double& alt, double t = 0.0)**
  Returns the latitude, longitude, and altitude of the position in the values lat, lon, and alt. If the current position type is ECI, the value, t, is used in the conversion function shown in Section 3.16.

- **void GetECI(double x[3], double t = 0.0)**
  Computes the position in ECI coordinates and returns those values in the first parameter. Notice that this does not change the stored state. For this reason, if you are going to be making successive calls to GetECI, it may be wise to call SetECI first or just save the values and not call the method again. The functions in Section 3.16 are used to change the coordinate system.

- **void GetECR(double x[3], double t = 0.0)**
  Returns the position in the ECR coordinate system. If the stored coordinate system is ECI, value, t, is used in the conversion function shown in Section 3.16.

4.31.2 See also

4.32 RHUMB_LINE: Rhumb Line Motion Dynamic Position

4.32.1 Description

An object following a Rhumb Line motion trajectory maintains a constant bearing. Bearing is the angle between the object’s velocity vector and compass north. The SpRhumbLine class will calculate a rhumb line going between two fixed positions with a fixed starting time. The end time can either be fixed, resulting in constant time, or the velocity can be fixed for the length of the equation. Even if the altitude changes between the start and end of the equation, both result in mathematically correct motion.

As with the other dynamic position attributes, they are usually obtained from a free list and then an init method is called on them to set up their data. Once that has been done, they may be added to a dynamic position attribute as a given equation of motion.

The methods of interest include:

- The init methods all take the same first seven arguments. These are the initial latitude, longitude, altitude, and time, followed by the final latitude, longitude, and altitude.
  
  - double init_Vconstant(double latInit,  
    double lonInit,  
    double altInit,  
    double timeInit,  
    double latFin,  
    double lonFin,  
    double altFin,  
    double vel)
  
    This method initializes the rhumb line equation assuming consistent velocity between the begin and endpoints. This velocity is the eighth input parameter, and the return value is the end time of this equation of motion.

  - double init_Tconstant(double latInit,  
    double lonInit,  
    double altInit,  
    double timeInit,  
    double latFin,  
    double lonFin,  
    double altFin,  
    double timeFin)

    This method initializes the rhumb line equation and calculates the constant speed required to traverse the distance in the time required. The speed is returned from this method.

- Operators for evaluating equation. The first argument is always the time the equation should be evaluated, and the second argument is always one of “ECR”, “ECI”, or “EARTH”. If the position was originally stored in ECR or EARTH and ECI is requested, or visa-versa, the “t” parameter is passed in to the conversion function (see Section 3.16).

  - void operator () (double t, int pt, double x[3])
    Returns the position at t - StartTime and in the given coordinate system.

  - void operator () (double t, int pt, double x[3], double v[3])
    Returns the position and velocity at t - StartTime and in the given coordinate system.
4.32.2 See also


4.32.3 Notes

A document detailing the mathematical derivation of the rhumb line may be made available on request.
4.33 S_SpHLA: Generic Self-Distributing Simulation Object

4.33.1 Description

S_SpHLA is an abstract base class from which simulation objects can inherit (instead of SpSimObj) in order to get access to an added layer of functionality: SPEEDES Distribution Management (DM) and DDM. Each simulation object that inherits from S_SpHLA (hereafter referred to as an S_SpHLA object) has an object proxy (which is an instance of the SpObjProxy class) that mirrors part of the simulation object’s state. The state reflected by the proxy is listed in the Objects.par file, which defines the proxies for all the simulation object classes that inherit from S_SpHLA in the simulation.

DM allows an individual S_SpHLA object to register an interest in the proxies of another class, one that also inherits from S_SpHLA. This is known as subscribing. After an object subscribes to a specific type, it will receive the proxies for each instance of that type (known as discovering) and subsequently receive updates to those proxies if changes are made to the simulation state mirrored by the proxies (known as reflecting or updating). S_SpHLA objects can subscribe, and unsubscribe as well, while the simulation is running.

The complement to subscribing is publishing. Each S_SpHLA object automatically publishes itself, which means that it tells the rest of the simulation that it exists so that if some other object subscribes to that type, the proxy can be delivered to it. An object can unpublish itself as well, which will cause any S_SpHLA objects that have discovered it to “undiscover” it. S_SpHLA objects will be informed of proxy discovers, reflects, and undiscovers through event handlers.

DDM is similar to DM, but instead of subscribing to a type of object, the S_SpHLA object subscribes to a “space”. A DDM space can be just about anything, although the easiest example is a physical region of the battlefield. When another S_SpHLA object enters that space, the subscriber will discover it and receive updates until the object leaves the region.

Using proxies, DM, and DDM can be very complicated, and the information presented here is only a brief overview. Please refer to the SPEEDES User’s Guide for detailed information on using these features.

- **S_SpHLA(char* objectType)**
  The constructor for the S_SpHLA class. The objectType argument must be the name of the class type in Objects.par that lists the state to be mirrored by this object’s proxy.

- **virtual double GetLookAheadSec()**
  Returns the lookahead value, in seconds, for range-based filtering. All objects participating in range-based filtering must use the same value of lookahead, which should be smaller than the minimum reschedule time (see the GetMinRescheduleTimeSec method). This virtual method must be implemented by the user if range-based filtering is being used, as the default implementation will produce an error.

- **virtual double GetMaxSensorRangeKm()**
  Returns the maximum range of the “sensor”, in kilometers, for range-based filtering. That is, the maximum range at which the “sensor” should discover proxies of other objects doing range-based filtering. For a sensor, this is a sensor range but for an ordinance, it may be the range of the ordinance. This virtual method must be implemented by the user if range-based filtering is being used, as the default implementation will produce an error.

- **virtual double GetMaxSpeedKmPerSec()**
Returns the maximum speed, in kilometers per second, that this object can obtain, for use in range-based filtering. This virtual method must be implemented by the user if range-based filtering is being used, as the default implementation will produce an error.

- **virtual double GetMinExpansionKm()**
  Returns the minimum expansion of the sensor range, in kilometers, for use in range-based filtering. Smaller values result in better filtering at the expense of more frequent events to determine the proxy distribution. This virtual method must be implemented by the user if range-based filtering is being used, as the default implementation will produce an error.

- **virtual double GetMinRescheduleTimeSec()**
  Returns the minimum reschedule time, in seconds, for range-based filtering. Smaller values result in better filtering but may result in too large an amount of CPU time being expended for the DDM alone. This virtual method must be implemented by the user if range-based filtering is being used, as the default implementation will produce an error.

- **virtual voidGetPosition(double time, double x[3], double v[3], double a[3])**
  Returns the position, velocity, and acceleration for this object (the x, v, and a arguments are output parameters) at the given time. For use in range-based filtering. This virtual method must be implemented by the user if range-based filtering is being used, as the default implementation will produce an error.

- **virtual int GetPositionTimes(double time, double& startT ime, double& endT ime)**
  Returns the bounds of the equation of motion (through the startT ime and endT ime parameters) at the given time. The actual integer return value will abort the simulation if it is equal to zero, and is ignored otherwise. If this object has a fixed position, fill in \(-\infty\) and \(+\infty\) for startT ime and endT ime. For use in range-based filtering. This virtual method must be implemented by the user if range-based filtering is being used, as the default implementation will produce an error.

- **virtual void SetName(char* objectName)**
  This overrides the implementation of SetName inherited from SpSimObj. It will set the name of this simulation object (retrievable via the GetName method), and also sets the proxy name of this object’s object proxy (retrievable via the proxy’s GetProxyName method).

- **void ClrActive()**
  Turns updates for High Level Architecture (HLA) object in the HLA gateway off.

- **void ClrGwHlaUpdate()**
  Turns periodic updates off for objects in the HLA Gateway.

- **SpPublishSpace* FindPublishSpace(const char* pubSpaceName)**
  Retrieves a pointer to the SpPublishSpace with the given name. NULL is returned if no such publish space is found. Space names are listed in the InterestSpaces.par file.

- **SpSubscribeSpace* FindSubscribeSpace(const char* subSpaceName)**
  Retrieves a pointer to the SpSubscribeSpace with the given name. NULL is returned if no such publish space is found. Space names are listed in the InterestSpaces.par file.

- **int GetActive() const**
  Returns 1 if this object is active as far as the HLA Gateway is concerned and 0 otherwise.

- **int GetGwHlaUpdate() const**
  Returns status of object periodic updates in the HLA Gateway. 0 indicates periodic updates are off and 1 indicates that they are on.
SpProxy* GetObjProxy()
Retrieves this simulation object’s object proxy.

SpPublishSpace* GetPublishSpace(const char* pubSpaceName)
Retrieves a pointer to the SpPublishSpace with the given name. The simulation is aborted if the space is not found. Space names are listed in the InterestSpaces.par file.

RB_queue* GetRemoteObjectProxies()
Returns a queue of the object proxies discovered by this object. The proxies are stored on the queue inside of F_SpProxyItems.

SpSubscribeSpace* GetSubscribeSpace(const char* subSpaceName)
Retrieves a pointer to the SpSubscribeSpace with the given name. The simulation is aborted if the space is not found. Space names are listed in the InterestSpaces.par file.

int Publish()
Declares this object’s existence to the rest of the simulation for the purposes of DM, which allows other S_SpHLA objects to discover it and receive attribute updates. This method is invoked automatically when the simulation object is initialized, but it may be needed to resume publishing after using the UnPublish method.

SpPublishSpace* PublishSpace(const SpSimTime& time, const char* pubSpaceName)
The named publish space is returned so that the caller can modify it. Modifications made to the space will take place at the given time. NULL is returned if no such publish space is found. Space names are listed in the InterestSpaces.par file.

void SetActive(int flag = 1)
Allows HLA objects to have their data published by the HLA Gateway.

void SetGwHlaUpdate(int flag = 1)
Turns periodic updates on for objects in the HLA Gateway.

int Subscribe(char* objectType)
Subscribes this S_SpHLA object to the given object type. Note that the object type is a name of a class from Objects.par, and not a C++ class name.

SpSubscribeSpace* SubscribeSpace(const SpSimTime& time, const char* subSpaceName)
The named subscribe space is returned so that the caller can modify it. Modifications made to the space will take place at the given time. NULL is returned if no such subscribe space is found. Space names are listed in the InterestSpaces.par file.

void UnPublish()
Hides this object’s existence from the rest of the simulation for the purposes of DM. S_SpHLA objects that have already discovered this object will undiscover it following the call to UnPublish.

int UnSubscribe(char* objectType)
Unsubscribes this HLA object from the given object type. Note that the object type is a name of a class from Objects.par, and not a C++ class name.

4.33.2 Examples

Suppose we wanted to create a simulation of ships and each ship always needs to see the draft of the other ships. The S_SpHLA class satisfies this need by allowing objects to see attributes of other objects. The code for S_Ship could look like this:
#include "S_SpHLA.H"
#include "SpExportAttribute.H"

class S_Ship : public S_SpHLA {
public:
    S_Ship():S_SpHLA("S_Ship") {}  
    void S_Ship::Init() {
        // The following line defines a mapping between Objects.par
        // and the actual attribute
        DEFINE_ATTRIBUTE(MyDraft, "Draft"); // Names need not be same
        MyDraft = 17.0;
    }
    //...
private:
    INT_ATTRIBUTE MuDraft;
};

This class definition must then always have a corresponding Objects.par entry:

    // Objects.par
    S_Ship {
        define int Draft // Ships have a publishable
        reference SUBSCRIBE S_Ship // Ships subscribe to ships
    }
} ...

4.33.3 See also

4.34 SpFreeDynAttributes: Dynamic Attribute Types Free List

4.34.1 Description

This class has but one purpose and that is to provide dynamic attributes to object proxies. Generally, one obtains an item from the global variable, SpFreeDynAttributes* FreeDynamicAttributes, using the RB_FREE_NEW function from RB_SpFrameworkFuncs.H and returns it using the RB_FREE_DELETE function from the same file. There are a large number of global itemized values that correspond to the specific dynamic attributes that are served by this list. The names of the enums are obtained by taking the name of the class and appending the string “_ID”:

- Dynamic logicals.
  - DYNAMIC_LOGICAL_CONSTANT_ID

- Dynamic integers.
  - DYNAMIC_INT_CONSTANT_ID

- Dynamic floats.
  - DYNAMIC_COMPLEX_EXPONENTIAL_ID
  - DYNAMIC_DOUBLE_CONSTANT_ID
  - DYNAMIC_EXPONENTIAL_ID
  - DYNAMIC_EXTRAPOLATE_ID
  - DYNAMIC_POLY_1_ID
  - DYNAMIC_POLY_2_ID
  - DYNAMIC_POLY_3_ID
  - DYNAMIC_POLY_4_ID
  - DYNAMIC_POLY_5_ID
  - DYNAMIC_POLY_6_ID
  - DYNAMIC_POLY_7_ID
  - DYNAMIC_POLY_8_ID
  - DYNAMIC_POLY_9_ID
  - DYNAMIC_POLY_10_ID
  - DYNAMIC_SPLINE_3_ID
  - DYNAMIC_SPLINE_6_ID

- Dynamic Positions.
  - CIRCULAR_ORBIT_ID
  - CONSTANT_MOTION_ID
  - Elliptical_ID
  - EXTRAPOLATE_MOTION_ID
  - GREAT_CIRCLE_ID
- LOITER_MOTION_ID
- POLY_1_MOTION_ID
- POLY_2_MOTION_ID
- POLY_3_MOTION_ID
- POLY_4_MOTION_ID
- POLY_5_MOTION_ID
- POLY_6_MOTION_ID
- POLY_7_MOTION_ID
- POLY_8_MOTION_ID
- POLY_9_MOTION_ID
- POLY_10_MOTION_ID
- RHUMB_LINE_ID
- SPLINE3_MOTION_ID
- SPLINE6_MOTION_ID

### 4.34.2 See also

4.35 SpObjProxy: Base Class for Obtaining Proxy Values

4.35.1 Description

The SpObjProxy class contains all data values for objects published through S_SpHLA. These classes are automatically delivered to subscribers through the DM or the DDM system. They are accessible through the method, GetRemoteObjProxies, on S_SpHLA.

- Methods for accessing attributes. Many of these accessor methods have an array index argument. If the default value is used, then this argument is ignored and the zeroth item is returned. If the attribute is looked up by name, an error message is printed if that attribute cannot be found.

  - **BASE_ATTRIBUTE* GetAttribute(int i, int j)**
    Returns a pointer to the $j^{th}$ array value of the attribute with index, $i$. No error checking is performed to see if $j$ is an array bounds error.

  - **BASE_ATTRIBUTE* Find(const int reference, int arrayIndex = 0) const**
    Same as GetAttribute(int i, int j) with a default for j.

  - **BASE_ATTRIBUTE* Find(const char* name, int arrayIndex = 0) const**
    Returns a pointer to the attribute with name, $n$, and array index, arrayIndex. If the attribute named $n$ is not found, NULL is returned.

  - **BASE_ATTRIBUTE* GetAttribute(int id)**
    Returns a pointer to the attribute with reference value, id.

  - **int GetReference(const char* n) const**
    Returns an integer reference to the attribute with name, $n$. If the attribute is not found, -1 is returned and an error message is printed.

  - **int GetReference(const char* name, char* t) const**
    Returns an integer reference to the attribute with name, $n$, and type, $t$. If the attribute is not found, -1 is returned and an error message is printed.

- Methods for accessing time interval for dynamic attributes. No error checking is performed to ensure reference or name is a dynamic position item.

  - **void GetTimeInterval(const char* name,**
    double& startTime,**
    double& endTime,**
    int index = 0)***
    Returns the start time in startTime and the end time in endTime for the dynamic attribute given by name and array item, index.

  - **void GetTimeInterval(const char* name,**
    double time,**
    double& startTime,**
    double& endTime,**
    int index = 0)***
    Returns the start time in startTime and the end time in endTime for the dynamic item dynamic at time, $t$, within the attribute given by name and array item, index.
– void GetTimeInterval(int reference, double& startTime, double& endTime, int index = 0)

Returns the start time in startTime and the end time in endTime for the dynamic attribute
given by reference and array item, index.

– int GetTimeInterval(int reference, double t, double& startTime, double& endTime, int index = 0)

Returns the start time in startTime and the end time in endTime for the dynamic item dy-
namic at time, t, within the attribute given by reference and array item, index.

- Methods for accessing integers.

– int GetDynamicInt(char* name, double t, int index = 0)

Returns the integer value for the attribute with name, n, at time, t, with array offset, index.

– int GetDynamicInt(int reference, double t, int index = 0)

Returns the integer value for the attribute with reference, ref, time, t, and array offset, index.

– int GetInt(const char* name, int index = 0) const

Returns the integer value for the attribute with name, n, and array offset, index.

– int GetInt(const int ref, int index = 0) const

Returns the integer value for the attribute with reference, ref, and array offset, index.

- Methods for accessing doubles.

– double GetDynamicFloat(char* n, double t, int index = 0)

Returns the floating point value for the attribute with name, n, at time, t, with array offset,
index. Note this actually returns a double, not a float.

– double GetDynamicFloat(char* n, double t, double& v, int index = 0)

Returns the floating point value for the attribute with name, n, at time, t, with array offset,
index. The value of v is set to the first derivative (velocity) at time, t. Note this actually
returns a double, not a float.

– double GetDynamicFloat(char* n, double t, double& v, double& a, int index = 0)

Returns the floating point value for the attribute with name, n, at time, t, with array offset
index. The value of v is set to the first derivative (velocity) at time, t, and a is set to the
second derivative (acceleration) at time, t. Note this actually returns a double, not a float.

– double GetDynamicFloat(int ref, double t, int index = 0)

Returns the floating point value for the attribute with reference, ref at time, t, with array
offset, index. Note this actually returns a double, not a float.

– double GetDynamicFloat(int ref, double t, double& v, int index = 0)

Returns the floating point value for the attribute with reference ref at time, t, with array
offset, index. The value of v is set to the first derivative (velocity) at time, t. Note this
actually returns a double, not a float.
double GetDynamicFloat(int ref, double t, double& v, double& a, int index = 0)
Returns the floating point value for the attribute with reference, ref, at time, t, with array offset, index. The value of v is set to the first derivative (velocity) at time, t, and a is set to the second derivative (acceleration) at time, t. Note this actually returns a double, not a float.

double GetFloat(const char* n, int index = 0) const
Returns the floating point value for the attribute with name, n, and array offset, index. Note this actually returns a double, not a float.

double GetFloat(const int reference, int index = 0) const
Returns the floating point value for the attribute with reference, ref, and array offset, index. Note this actually returns a double, not a float.

Methods for accessing strings.

const char* GetString(const char* n, int index = 0) const
Returns a reference to the string for the attribute with name, n, and array index, index.

const char* GetString(const int reference, int index = 0) const
Returns a reference to the string for the attribute with reference, ref, and array index, index.

Methods for accessing binary buffers

char* GetBinaryBuffer(int& numBytesInBuffer, const char* n, int index = 0) const
Returns a pointer to the binary buffer attribute with name, n, and array index, 0. numBytesInBuffer is filled in with the size of the buffer attribute.

char* GetBinaryBuffer(int& numBytesInBuffer, const int reference, int index = 0) const
Returns a pointer to the binary buffer attribute with reference, ref, and array index, 0. numBytesInBuffer is filled in with the size of the buffer attribute.

Methods for accessing logicals

int GetDynamicLogical(char* name, double t, int index = 0)
Returns the logical attribute value for name, n, time, t, and array offset, index.

int GetDynamicLogical(int reference, double t, int index = 0)
Returns the logical attribute value for reference, ref, time, t, and array offset, index.

int GetLogical(const char* n, int index = 0) const
Returns the logical attribute value for name, n and array offset, index.

int GetLogical(const int reference, int index = 0) const
Returns the logical attribute value for reference, ref, and array offset, index.

Methods for accessing an object attribute.

DYNAMIC_OBJECT* GetDynamicObject(char* name, double t, int index = 0)
Returns the object proxy contained within the object attribute with name, n, and array offset index at time, t.

DYNAMIC_OBJECT* GetDynamicObject(int reference, double t, int index = 0)
Returns the object proxy contained within the object attribute with reference, ref, and array offset, index, at time, t.
– SpObjProxy* GetObjProxy(char* n, int index = 0)
  Returns the object proxy contained within the object attribute with name, n, and array offset, index.

– SpObjProxy* GetObjProxy(int ref, int index = 0)
  Returns the object proxy contained within the object attribute with reference, ref, and array offset, index.

• Methods for accessing positions. All of these methods accept an argument int, pt, which is the coordinate system in which the position should be evaluated. pt must be one of ECR, ECI, or EARTH and the conversion functions are described in SpConvert.H 3.16. For conversions to ECI, the optional double t = 0.0 parameter is passed to the conversion function to account for the Earth’s rotation.

– void GetDynamicPosition(char* n,
  double x[3],
  int pt,
  double t = 0.0,
  int index = 0)
  Fills in x with the position of the dynamic position attribute with name, n.

– void GetDynamicPosition(char* n,
  double x[3],
  double v[3],
  int pt,
  double t = 0.0,
  int index = 0)
  Fills in x with the position and v with the velocity of the dynamic position attribute with name, n.

– void GetDynamicPosition(char* n,
  double x[3],
  double v[3],
  double a[3],
  int pt,
  double t = 0.0,
  int index = 0)
  Fills in x with the position, v with the velocity, and a with the acceleration of the dynamic position attribute with name, n.

– void GetDynamicPosition(int ref,
  double x[3],
  int pt,
  double t = 0.0,
  int index = 0)
  Fills in x with the position of the dynamic position attribute with reference, ref.

– void GetDynamicPosition(int ref,
  double x[3],
  double v[3],
  int pt,
  double t = 0.0
  int index = 0)
Fills in \( x \) with the position and \( v \) with the velocity of the dynamic position attribute with reference, \( \text{ref} \).

\[
\text{void GetDynamicPosition(int } \text{ref}, \ \\
\text{double } x[3], \ \\
\text{double } v[3], \ \\
\text{double } a[3], \ \\
\text{int } \text{pt}, \ \\
\text{double } t = 0.0, \ \\
\text{int } \text{index} = 0) \]

Fills in \( x \) with the position, \( v \) with the velocity, and \( a \) with the acceleration of the dynamic position attribute with reference, \( \text{ref} \).

\[
\text{void GetPosition(char* } \text{n}, \text{ double } x[3], \text{ int } \text{pt}, \text{ double } t = 0.0, \text{ int } \text{index} = 0) \]

Fills in \( x \) with the position of the static position attribute with name, \( \text{n} \).

\[
\text{void GetPosition(char* } \text{n}, \ \\
\text{double } x[3], \ \\
\text{double } v[3], \ \\
\text{int } \text{pt}, \ \\
\text{double } t = 0.0, \ \\
\text{int } \text{index} = 0) \]

Fills in \( x \) with the position and \( v \) with the velocity of the static position attribute with name, \( \text{n} \).

\[
\text{void GetPosition(char* } \text{n}, \ \\
\text{double } x[3], \ \\
\text{double } v[3], \ \\
\text{double } a[3], \ \\
\text{int } \text{pt}, \ \\
\text{double } t = 0.0, \ \\
\text{int } \text{index} = 0) \]

Fills in \( x \) with the position, \( v \) with the velocity, and \( a \) with the acceleration of the static position attribute with name, \( \text{n} \).

\[
\text{void GetPosition(int } \text{ref}, \text{ double } x[3], \text{ int } \text{pt}, \text{ double } t = 0.0, \text{ int } \text{index} = 0) \]

Fills in \( x \) with the position of the static position attribute with reference, \( \text{ref} \).

\[
\text{void GetPosition(int } \text{ref}, \ \\
\text{double } x[3], \ \\
\text{double } v[3], \ \\
\text{int } \text{pt}, \ \\
\text{double } t = 0.0, \ \\
\text{int } \text{index} = 0) \]

Fills in \( x \) with the position and \( v \) with the velocity of the static position attribute with reference, \( \text{ref} \).

\[
\text{void GetPosition(int } \text{ref}, \ \\
\text{double } x[3], \ \\
\text{double } v[3], \ \\
\text{double } a[3], \ \\
\text{int } \text{pt}, \ \\
\text{double } t = 0.0, \ \\
\text{int } \text{index} = 0) \]
Fills in x with the position, v with the velocity, and a with the acceleration of the static position attribute with reference, ref.

- Methods for accessing lists.
  - `LIST_ATTRIBUTE* GetList(char* n, int index = 0)`
    Returns the list attribute with name, n, and array offset, index.
  - `LIST_ATTRIBUTE* GetList(int ref, int index = 0)`
    Returns the list attribute with reference, ref, and array offset, index.
  - `SpObjProxy* GetListName(char* n, char* itemName, int index = 0)`
    Returns the item with name, itemName, on the list with name, n, and array index, index.
  - `SpObjProxy* GetListName(int ref, char* itemName, int index = 0)`
    Returns the item with name, itemName, on the list with reference, ref, and array index, index.

- Methods for Active and HLA Gateway flags.
  - `int GetActive() const`
    Returns 1 if this object is active as far as the HLA Gateway is concerned and 0 otherwise.
  - `int GetGwHlaUpdate() const`
    Returns 1 if the HLA Gateway is reflecting attributes on a periodic basis and returns 0 otherwise.

- `int GetProxySimObjGlobalId()`
  Returns the global id of the simulation object to which this object proxy belongs.

- `int GetProxySimObjLocalId()`
  Returns the local id of the simulation object to which this object proxy belongs.

- `int GetProxySimObjMgrId()`
  Returns the simulation object manager id (object type) of the simulation object to which this object proxy belongs.

- `char* GetProxyName()`
  Returns the name of the simulation object to which this object proxy belongs.

- `int GetProxyNode()`
  Returns the node of the simulation object to which this object proxy belongs.

4.35.2 Examples

SpObjProxy is frequently subclassed in order to provide convenience methods that are specific to the class being proxied. The following is a small example of a class and its proxy:

```cpp
#include "S_SpHLA.H"
#include "SpObjProxy.H"

class S_Ship : public S_SpHLA {
  public:
    S_Ship():S_SpHLA("S_Ship") {}  
}
```
class ShipProxy : public SpObjProxy {
public:
    ShipProxy();
    int GetCrewSize() {return GetInt(CrewSizeId);}
    double GetDraft() {return GetDouble(DraftId);}
    double GetMaxSpeed() {
        return sqrt(sqr(GetDouble(MaxVelocityId, 0)) + 
            sqr(GetDouble(MaxVelocityId, 1)) + 
            sqr(GetDouble(MaxVelocityId, 2)));
    }
    double GetNumPersonsPerSquareFootOfDeckArea() {
        return ((double) GetInt(CrewSizeId)) / 
            GetDouble(DeckAreaInSquareFeetId);
    }
private:
    static int CrewSizeId;
    static int DraftId;
    static int MaxVelocityId;
    static int DeckAreaInSquareFeet;
    double sqr(double x) {return x * x;}
};

// S_Ship.C
#include "S_Ship.H"

int ShipProxy::CrewSizeId = -1;
int ShipProxy::DraftId = -1;
int ShipProxy::MaxVelocityId = -1;
int ShipProxy::DeckAreaInSquareFeetId = -1;

void S_Ship::Init() {
    DEFINE_ATTRIBUTE(CrewSize, "CrewSize");
    DEFINE_ATTRIBUTE(Draft, "Draft");
    DEFINE_ATTRIBUTE(MaxVelocity, "MaxVelocity");
    DEFINE_ATTRIBUTE(DeckAreaInSquareFeet, "DeckAreaInSquareFeet");
}

ShipProxy::ShipProxy() {
    if (CrewSizeId == -1)
        CrewSizeId = GetReference("CrewSizeId", "S_Ship");
    if (DraftId == -1)
        DraftId = GetReference("DraftId", "S_Ship");
    if (MaxVelocityId == -1)
        MaxVelocityId = GetReference("MaxVelocityId", "S_Ship");
    if (DeckAreaInSquareFeet == -1)
        DeckAreaInSquareFeet = GetReference("DeckAreaInSquareFeet", "S_Ship");
}
4.35.3 See also

4.36 SPLINE3_MOTION: Dynamic Position Fitting Position and Velocity

4.36.1 Description

If both the position and velocity of an object are known at two different points in time, SPLINE3_MOTION is an excellent choice of a dynamic position class. This equation of motion will compute the polynomial that exactly fits two points and velocities. This produces a third order polynomial and, if the data is smooth, results in a smooth curve connecting the points. If the data source is not smooth, this can result in a very erratic polynomial.

The API for this equation is simple with only one initialization method and three evaluator methods:

- **Initializer method**
  
  ```c
  void init(double t0, 
  double x0[3], 
  double v0[3], 
  double t1, 
  double x1[3], 
  double v1[3])
  ```
  
  Initializes the equation with the time, t0, position, x0 at t0, velocity, v0 at t0, time, t1, position, x1 at t1, and velocity, v1 at t1. This call generates the coefficients for the third order polynomial satisfying these constraints. The coordinate system is initialized with an undefined state so the parent classes, SetEARTH, SetECI, or SetECR, must be called to set the appropriate coordinate system.

- **Evaluation methods**
  
  ```c
  void operator () (double t, int pt, double x[3])
  
  Sets the values of x to the position of the cubic at time, t - StartTime, for coordinate system, pt, where pt is one of ECR, ECI, or EARTH. The conversion is performed using the functions in Section 3.16.
  ```

  ```c
  void operator () (double t, int pt, double x[3], double v[3])
  
  Sets the values of x to the position and the values of v to the velocity of the cubic at time, t - StartTime, for coordinate system, pt, where pt is one of ECR, ECI, or EARTH. The conversion is performed using the functions in Section 3.16.
  ```

  ```c
  void operator () (double t, int pt, double x[3], double v[3], double a[3])
  
  Sets the values of x to the position, the values of v to the velocity, and the values of a to the acceleration of the cubic at time, t - StartTime, for coordinate system, pt, where pt is one of ECR, ECI, or EARTH. The conversion is performed using the functions in Section 3.16.

4.36.2 See also

4.37 SPLINE6_MOTION: Dynamic Position Fitting Position, Velocity, and Acceleration

4.37.1 Description

If the position, velocity, and acceleration of an object is known at two different points in time, SPLINE6_MOTION is an excellent choice of a dynamic position class. This equation of motion will compute the polynomial that exactly fits two points and velocities. This produces a fifth order polynomial and, if the data is smooth, results in a smooth curve connecting the points. If the data source is not smooth, this can result in a very erratic polynomial.

The API for this equation is simple with only one initialization method and three evaluator methods:

- **Initializer method.**
  ```
  void init(double t0,
            double x0[3],
            double v0[3],
            double a0[3],
            double t1,
            double x1[3],
            double v1[3],
            double a1[3])
  ```
  Initializes the equation with the time, t0, position, x0 at t0, velocity, v0 at t0, acceleration, a0 at t0, time, t1, position, x1 at t1, velocity, v1 at t1, and acceleration, a1 at t1. This call generates the coefficients for the fifth order polynomial satisfying the constraints. The coordinate system is initialized with an undefined state so the parent classes, SetEARTH, SetECI, or SetECR, must be called to set the appropriate coordinate system.

- **Evaluation methods.**
  ```
  void operator () (double t, int pt, double x[3])
  ```
  Sets the values of x to the position of the cubic at time, t - StartTime, for coordinate system, pt, where pt is one of ECR, ECI, or EARTH. The conversion is performed using the functions in Section 3.16.

  ```
  void operator () (double t, int pt, double x[3], double v[3])
  ```
  Sets the values of x to the position, the values of v to the velocity of the cubic at time, t - StartTime, for coordinate system, pt, where pt is one of ECR, ECI, or EARTH. The conversion is performed using the functions in Section 3.16.

  ```
  void operator () (double t, int pt, double x[3], double v[3], double a[3])
  ```
  Sets the values of x to the position, the values of v to the velocity, and the values of a to the acceleration of the cubic at time, t - StartTime, for coordinate system, pt, where pt is one of ECR, ECI, or EARTH. The conversion is performed using the functions in Section 3.16.

4.37.2 See also

4.38 SpProxyDefines.H: Proxy and General Simulation Constants

4.38.1 Description

A great many standard constants are needed through the development of simulations. Many of these provided in SpProxyDefines.H are listed here:

- **static const double Days_per_Second**
  Number of days per second.

- **#define EARTH**
  Latitude, Longitude, Altitude coordinate system. This is passed into many conversion functions described in Section 3.16.

- **#define ECI**
  Earth Centered Inertial coordinate system. This is passed into many conversion functions described in Section 3.16.

- **#define ECR**
  Earth Centered Rotating coordinate system. This is passed into many conversion functions described in Section 3.16.

- **static const double Hours_per_Second**
  Number of hours per second.

- **#define INFINITY 1e20**
  Value used throughout SPEEDES to represent $\infty$.

- **static const double Kilometers_per_Mile**
  Number of kilometers in a mile.

- **static const double Kilometers_per_NauticalMile**
  Number of kilometers in a nautical mile.

- **static const double Miles_per_Kilometer**
  Number of miles in a kilometer.

- **static const double Minutes_per_Second**
  Number of minutes per second.

- **static const double NauticalMiles_per_Kilometer**
  Number of nautical miles in a kilometer.

- **static const double NauticalMiles_per_Mile**
  Number of nautical miles in a mile.

- **static const double RE**
  Radius of the earth in kilometers.

- **static const double Seconds_per_Day**
  Number of seconds in a day.

- **static const double Seconds_per_Hour**
  Number of seconds per hour.
- static const double Seconds_per.Minute
  Number of seconds per minute.

4.38.2 Used in

SpConvert.H (see Section 3.16), as well as most of the object proxy code and the dynamic position (equation of motion) classes.
4.39 STRING_ATTRIBUTE: String Attribute

4.39.1 Description

STRING_ATTRIBUTE behaves just like an RB_SpString, with the additional feature that any changes are reflected to all objects subscribing to this object. To this end, a STRING_ATTRIBUTE can be treated just like a normal character string.

- **operator const char*() const**
  Returns the char* (i.e. string) value of the STRING_ATTRIBUTE.

- **const char* operator =(char* rhs)**
  Assigns the value, rhs, to the STRING_ATTRIBUTE.

- **int operator ==(char* a)**
  Returns 1 if the STRING_ATTRIBUTE and a are equal and 0 otherwise.

- **int operator !=(char* a)**
  Returns 0 if the STRING_ATTRIBUTE and a are equal and non-zero otherwise.

4.39.2 See also

BASE_ATTRIBUTE 4.2 and RB_SpString 3.11.
Chapter 5

Data Distribution Management (DDM)
5.1 SpDDMRegion Definition

5.1.1 Description

DDM in SPEEDES is performed by choosing a space and publishing or subscribing within that space (See Sections 5.2 and 5.3 for more information). Regions are created within a space to filter based on various dimensions of that space.

SpDDMRegion is the base class for all region based subscription and publication. SpPublishRegion and SpSubscribeRegion inherit from this class and extend the methods of SpDDMRegion. The constructors for these classes are generally not called by the user, but rather, a SpPublishRegion or SpSubscribeRegion is obtained by the CreateRegion method of the SpPublishSpace or SpSubscribeSpace classes.

- **SpDDMRegion**
  
  - **void CreateDimension(const char* dimName, const char* enum name, ...)**
    Creates a publication or subscription dimension within the Dimension, dimName. The string, dimName, is the name of a space as defined in InterestSpaces.par. The parameter, enum_names, must correspond to valid enumerated values within that dimension. This method takes a variable number of arguments and must be NULL-terminated.
  
  - **void CreateDimension(const char* dimName, double lo, double hi, ...)**
    Creates a dimension within the space with the given dimName. The value for dimName is the name of a space as defined in InterestSpaces.par and the values lo and hi must span a subset of the values within that space. This method takes a variable number of arguments and must be NULL-terminated.
  
  - **void DeleteDimension(const char* dimName)**
    Deletes the dimension with the name, dimName. Deletion of a dimension corresponds to unsubscription or unpublication to that dimension. Returns silently if the region specified does not exist.
  
  - **SpDDMDim* FindDimension(const char* dimName)**
    Returns the dimension with the name, dimName. If it cannot be found, returns NULL.
  
  - **void ModifyDimension(const char* dimName, const char* enum name, ...)**
    Modifies a publication or subscription dimension within the dimension, dimName. The previously created space specified must be of type, EnumType, and the NULL-terminated list of enum_name’s must correspond to valid enumerated values within that dimension.
  
  - **void ModifyDimension(const char* dimName, double lo, double hi, ...)**
    Changes the range of values for publication or subscription within a previously created dimension. This method takes a variable number of arguments and must be NULL-terminated.

- **SpPublishRegion**
  Does not add any methods that should be accessed by the general user. Only the methods in SpDDMRegion may be used.

- **SpSubscribeRegion**
  In addition to the methods of SpDDMRegion, SpSubscribeRegion adds methods for class based subscription and unsubscription.

  - **void AddClass(const char* className, ...)**
    Adds the classes passed as arguments to the list of classes to which this object is subscribed. This needs to be a NULL-terminated list.
5.1 SpDDMRegion Definition

- **void RemoveClass(const char* className, ...)**
  Removes the classes passed as arguments to the list of classes to which this object is subscribed. This needs to be a NULL-terminated list.

### 5.1.2 Examples

In general, the process for creating or modifying a dimension behaves as follows:

```c
// First create a publish space using the S_SpHLA
// method PublishSpace
SpPublishSpace* publishSpace = PublishSpace(SpGetTime(), "Missile");
SpPublishRegion* publishRegion = publishSpace->CreateRegion("Boost State");
publishRegion->CreateDimension("BoostingEnumDim", "Boosting", NULL);

// Or create a subscribe space using the S_SpHLA
// method SubscribeSpace
double subscribeTime = 1294.0;
SpSubscribeSpace* subscribeSpace = SubscribeSpace(subscribeTime, "CommChannel");
SpSubscribeRegion* subscribeRegion = subscribeSpace->CreateRegion("Region 1");
subscribeRegion->CreateDimension("CommRanges", 10.0, 1000.0, NULL);
subscribeRegion->AddClass("Threat", "Interceptor", NULL);
subscribeRegion->CreateDimension("BoostingEnumDim", "Boosting", NULL);

// Inform the framework of the changes to subscription/publication
subscribeSpace->Update();
publishSpace->Update();

// ... another method
// Find a space and unpublish by deleting it.
SpPublishSpace* publishSpace = FindPublishSpace("Missile");
publishSpace->DeleteRegion("Boost State");

// Or find a space and modify the subscriptions within that space
SpSubscribeSpace* subscribeSpace = FindSubscribeSpace("CommChannel");
SpSubscribeRegion* subscribeRegion = subscribeSpace->FindRegion("Region 1");
subscribeRegion->ModifyDimension("CommRanges", low, high, NULL);

// ...
subscribeSpace->Update();
publishSpace->Update();
```

### 5.1.3 See also

S_SpHLA 4.33, SpPublishSpace 5.2, and SpSubscribeSpace 5.3.
5.2 SpPublishSpace: DDM Publication Space Modification

5.2.1 Description

SpPublishSpace, along with SpSubscribeSpace in Section 5.3, forms the basis for all publication and subscription services in DDM. The general pattern is to look up a region within a SpPublishSpace or SpSubscribeSpace, modify that region, and then call Update() on the space. A SpPublishSpace is generally never created by the user, but rather the SPEEDES framework provides a SpPublishSpace through S_SpHLA methods, such as S_SpHLA::PublishSpace.

- **SpPublishRegion* CreateRegion(const char* name)**
  Creates a publication region with the given name. If CreateRegion is called twice with the same name, the previously created region is returned without any error or warning message. Region names can be free form strings using any characters for describing the given region. Passing NULL will result in undefined behavior.

- **int DeleteRegion(const char* name)**
  Deletes the previously created region. If the region does not exist, 0 is returned. Otherwise, 1 is returned. Passing NULL will result in undefined behavior.

- **SpPublishRegion* FindRegion(const char* name)**
  Looks up the previously created region with the given name. If the region is not found, NULL is returned. Passing NULL will result in undefined behavior.

- **void Update()**
  Signals the SPEEDES framework that all changes to the SpPublishSpace since the last Update call should be committed. If Update is not called, no changes will be made to the publication space.

5.2.2 Examples

Referencing the sample InterestSpaces.par file in Section 5.3, the following code demonstrates a small DDM publication example:

```cpp
void S_Tank::Init() {
    // Start publishing at 19.0 seconds
    SpPublishSpace* pubSpace = PublishSpace(19.0, "NorthAmericanTheater");

    // Create a region
    SpPublishRegion* myRegion = pubSpace->CreateRegion("MyInterests");

    // Here one would then take myRegion and set up the correct
    // values such as what radio frequencies are published by this
    // object as well as to what Allegence this object belongs

    pubSpace->Update(); // Tell SPEEDES to update this filter
}

void S_Ship::DisableSubscriptions() {
```
SpPublishSpace* pubSpace =
    FindPublishSpace(10.0, "NorthAmericanTheater");
if (pubSpace == NULL) {
    RB_cout << "Could not find the publish space for "
    << "NorthAmericanTheater" << endl;
    return;
}
SpPublishRegion* myRegion = DeleteRegion("MyInterests");
pubSpace->Update();
}

5.2.3 See also

S_SpHLA 4.33, SpPublishRegion 5.1, and SpSubscribeSpace 5.3. Section 5.3 provides a good description of how to generate an InterestSpaces.par file, as well as advice on its contents.
5.3 SpSubscribeSpace: DDM Subscription Space Modification

5.3.1 Description

SpSubscribeSpace, along with SpPublishSpace in Section 5.2, forms the basis for all publication and subscription services in DDM. The general pattern is to look up a region within a SpPublishSpace or SpSubscribeSpace, modify that region, and then update the space. A SpSubscribeSpace is generally never created by the user, but rather the SPEEDES framework provides a SpSubscribeSpace through S_SpHLA methods such as S_SpHLA::SubscribeSpace.

- **SpSubscribeRegion* CreateRegion(const char* name)**  
  Creates a subscription region with the given name. If CreateRegion is called twice with the same name, the previously created region is returned without any error or warning message. Region names can be free form strings using any characters for describing the given region. Passing NULL will result in undefined behavior.

- **int DeleteRegion(const char* name)**  
  Deletes the previously created region. If the region does not exist, 0 is returned. Otherwise, 1 is returned. Passing NULL will result in undefined behavior.

- **SpSubscribeRegion* FindRegion(const char* name)**  
  Looks up the previously created region with the given name. If the region is not found, NULL is returned. Passing NULL will result in undefined behavior.

- **void Update()**  
  Signals the SPEEDES framework that all changes to the SpSubscribeSpace since the last Update call should be committed. If this is not called, no changes will be made to the subscription space.

5.3.2 Examples

A file called InterestSpaces.par must be created which defines the spaces in which objects can express their interest for publication, subscription, or both. InterestSpaces.par generally has the form:

```
InterestSpaces {
  reference Space NorthAmericanTheater
}

NorthAmericanTheater {
  reference EnumType Allegiance
  reference Dimension RadioFrequency
  reference Theater ContinentalUnitedStates
}

EnumType Allegiance {
  enum Red
  enum Blue
  enum White
  logical Distribute T
}

Dimension RadioFrequency {
```
This example demonstrates all of the basic components of an InterestSpaces.par file. It must always contain the section, InterestSpaces, which must define at least one space. In this example, the NorthAmericanTheater is defined with three different dimensions.

1. The first dimension is an enumerated dimension called Allegiance. Allegiance has three possibilities, the Red team, the Blue team, or the White team. A required line in each dimension is the logical Distribute.

Distribute indicates whether separate grid managers will be generated to manage the dimension. For example, if this were the only dimension and Distribute was set to “F”, only one grid manager would be created to handle the values of Red, Blue, and White. On the other hand, if Distribute was set to “T”, then three grid managers will be created to support each of the dimensions Red, Blue, and White. Having separate grid managers will result in fewer rollbacks for the simulation when a large number of changes are made in subscriptions or publications at the expense of computer memory use.

Caution should be used when setting Distribute to “T” for too many dimensions, as the affects are multiplicative. If we had four different EnumType dimensions, the first with 5 elements, the second with 30, the third with 8, and the fourth with 16, each with Distribute set to “T”, the framework will generate 5*30*8*16=19,200 grid managers which could be excessive.
2. The second dimension is RadioFrequency, which represents the range of radio frequencies which will be monitored or broadcast throughout the simulation. Ranges of doubles like this dimension always have a floating point Lo and Hi value. These are floating point values and Lo must always be strictly less than Hi.

With every range of doubles it is always required to have at least two resolutions. The first of the resolutions must be smaller than Hi-Lo and subsequent resolutions must be progressively smaller. The first resolution is used for determining how many grid manager objects are created if the dimension is distributed. For example, if the dimension went from 110 to 150 and the first resolution was 12, (150-110)/12 rounded up gives 4 grid manager objects which would be created if this dimension were distributed.

SPEEDES dynamically builds a tree within each grid manager that manages a given dimension with finer and finer resolutions. If most publications or subscriptions will have a large subset of the regions, then a great deal of memory will be expended if the resolutions are very fine. On the other hand, if the publications or subscriptions will occupy only a small part of the region, using finer resolutions will improve filtering efficiency.

3. The final dimension is the Theater, ContinentalUnitedStates, which is defined below. A Theater always has a LatLng dimension, as well as an Altitude dimension. The LatLng dimension defines the latitude and longitude limits of the theater in degrees, while the Altitude dimension gives the range of altitudes for the dimension in kilometers above sea level.

Each of these subdimensions is similar to normal dimensions in that at least two resolutions must be provided. Also, as with all dimensions, these dimensions can either be distributed or not.

If a space contains a Theater dimension, range-based filtering will be automatically enabled. This implies that the user will have to implement all the SSpHLA methods for range-based filtering, as described in Section 4.33

Once an InterestSpaces.par file has been written, code can be written to utilize the spaces for publication or subscription. A small example follows:

```cpp
void S_Ship::Init() {
    // Start subscribing at 10.0 seconds
    SpSubscribeSpace* subSpace =
        SubscribeSpace(10.0, "NorthAmericanTheater");

    // Create a region
    SpSubscribeRegion* myRegion =
        subSpace->CreateRegion("MyInterests");

    // Here one would then take myRegion and set up the correct
    // values such as what radio frequencies are monitored by this
    // object as well as what Allegiance objects this object
    // wants to see.

    subSpace->Update(); // Tell SPEEDES to update this filter
}

void S_Ship::DisableSubscriptions() {
    SpSubscribeSpace* subSpace =
        FindSubscribeSpace(10.0, "NorthAmericanTheater");
    if (subSpace == NULL) {
        RB_cout << "Could not find the subscribe space for 
```
5.3 SpSubscribeSpace: DDM Subscription Space Modification

    << "NorthAmericanTheater" << endl;
    return;
}
SpSubscribeRegion* myRegion = DeleteRegion("MyInterests");
subSpace->Update();
}

5.3.3 See also

SpHLA 4.33, SpSubscribeRegion 5.1, and SpPublishSpace 5.2.
Chapter 6

External Modules
6.1 SpEmHostUser: External Module Interface

6.1.1 Description

The SpEmHostUser class is the external module interface to the Host Router (and, therefore, into SPEEDES).

- Methods for constructing the SpEmHostUser.
  - SpEmHostUser()
    Default constructor. This constructor will attempt to open a speedes.par file and read the simulation group number, the port number, and machine name for the SpeedesServer for which a SpEmHostUser connection will be made.
  - SpEmHostUser(char* fileName)
    Constructor for creating a SpEmHostUser from a file called fileName for a pre-recorded simulation run. The pre-recorded files are recorded by an external connection created for simulation with method, RecordInputMessages, enabled.
  - SpEmHostUser(char* machineName, int portNum, int groupId)
    Constructor for creating a SpEmHostUser from a specific machineName, portNum, and the SPEEDES simulation groupId.
  - SpEmHostUser(SpDataParser* parser)
    Constructor for creating a SpEmHostUser from an already opened parameter file.

- DisconnectBarrier()
  Removes any barriers that this SpEmHostUser has established.

- void RecordInputMessages(char* fileName)
  Records all received messages to a file called fileName. This file can then be played back by using the SpEmHostUser(char* fileName) constructor.

- char* ReceiveData(SpSimTime& time, int& bytes, int& simObjGlobalId)
  char* ReceiveData(double& time, int& bytes, int& simObjGlobalId)
  Returns data sent by a SPEEDES simulation using the SpHostUser::SendData call. Parameters time, bytes, and simObjGlobalId represent the time at which the data was sent, the size of the data in bytes, and the simulation global id from which the data was sent, respectively.

- int SpeedesExecuting()
  Returns 1 if the simulation that the SpStateMgr connected to is still running. Otherwise, it returns 0.

- The following methods provide options for querying a SPEEDES simulation about object’s name and id.
  - int NameLookup(char* objName)
    Returns the global id for the name of the object instance specified by objName.
  - SpNameValueList* ObjNames(char* type)
    SpNameValueList* ObjNames(int type = -1)
    Returns a list of the names of all instances of the objects in a SPEEDES simulation. If type is specified, then all object instance names of that type are returned.
6.1 SpEmHostUser: External Module Interface

- int SimObjMgrType(char* type)
  Returns the id of an object manager type for the requested “type” in a SPEEDES simulation.

- SpNameValueList* SimObjMgrTypes()
  Returns a list of the types of object managers in a SPEEDES simulation.

- int GetNumNodes()
  Returns the number of SPEEDES nodes.

- void KillSPEEDES()
  Stops the SPEEDES simulation from running and allows SPEEDES to exit gracefully. Also sends a SPEEDES exiting message to all connected external modules.

- void Pause()
  Pauses the SPEEDES simulation.

- void Pause(char* pauseName, SpSimTime time = 0)
  Creates a named pause called pauseName and pauses the SPEEDES simulation. Parameter, time, specifies when the named pause will take affect. If the time is not specified or it is earlier than GVT, then the simulation will pause at GVT.

- SpNameValueList* Query(char* objName, char* queryName = "")
- SpNameValueList* Query(int simObjGlobalId, char* queryName = "")
  These methods are used to make queries on specific simulation objects in the simulation. When queryName is NULL, then virtual method, SpSimObj::Query, is called. When queryName is not NULL, then virtual method, SpSimObj::NamedQuery, is called. If the virtual methods are not defined, then the Query returns a SpNameValueList, which contains the queried objects global id, query time, and the number of events the query object has on its queue.

- void Resume()
  Resumes the SPEEDES simulation which was halted by pause.

- void Resume(char* resumeName)
  Removes any named pauses whose name is resumeName.

- The following methods provide options for sending data into a SPEEDES simulation.
  - int SendCommand(char* commandName,
    int simObjGlobalId,
    char* messageData,
    int messageLength,
    int returnStatus = 0)
  Sends a command called commandName to an object specified by simObjGlobalId. The receiving simulation object must have an event called by the same name as specified by commandName in order for this method to succeed. The parameter specifies any input data that needs to be sent into SPEEDES with parameter, messageLength, specifying the number of bytes to be sent. Input data can be NULL. When parameter, returnStatus, is equal to 0 (default), the return value is always SpHostRouterMsgs::COMMAND_STATUS_GOOD. When parameter, returnStatus, is non-zero, then the return status could also be SpHostRouterMsgs::COMMAND_STATUS_BAD_OBJECT or SpHostRouterMsgs::COMMAND_STATUS_BAD_EVENT.
- int SendCommand(char* commandName,
  char* objName,
  char* messageData,
  int messageLength,
  int returnStatus = 0)

This command is identical to the previous command, except that the object that the command is sent to is specified by objName instead of simObjGlobalId.

- SCHEDULE_EVENT_REPLY_MESSAGE*
  ScheduleEvent(SpSimTime time,
    int globalId,
    char* eventName,
    char* msg,
    int msgBytes,
    char* data,
    int dataBytes,
    TimeMode tmMode = IF_IN_PAST_IGNORE,
    int cancelHandleRequested = 1)

SCHEDULE_EVENT_REPLY_MESSAGE*
  ScheduleEvent(SpSimTime time,
    char* objName,
    char* eventName,
    char* msg,
    int msgBytes,
    char* data,
    int dataBytes,
    TimeMode tmMode = IF_IN_PAST_IGNORE,
    int cancelHandleRequested = 1)

These methods are similar to method, SendCommand, except that this method schedules events in SPEEDES input parameter time, rather than at GVT. Events can be scheduled on any object by either global id or object name. Parameters, msg and msgBytes, are part of the internal functionality in SPEEDES and, with the implementation of the unified API, SPEEDES users no longer have access to these fields. In order to use this method, the user must create a temporary SpMsg for parameter msg. Data can be sent with this method by using parameters, data and dataBytes. Parameter, tmMode, can be set to IF_IN_PAST_IGNORE or IF_IN_PAST_SCHEDULE_AT_GVT. The former indicates that events scheduled in the past of GVT should be ignored. The latter will schedule events in the past of GVT at GVT. When parameter, cancelHandleRequested, is set to 1, then an event cancel handle is returned. Otherwise, NULL is returned.

- The following methods provide options for controlling and accessing simulation time.
  
  - SpSimTime& GetMinMsgTime()
    Queries SPEEDES for the minimum time of an outgoing message.
  
  - SpSimTime GetTime()
    Queries SPEEDES for the current GVT.
  
  - SpSimTime GetTimeGranted()
    Returns the time granted by SPEEDES.
6.1 SpEmHostUser: External Module Interface

- **int TimeAdvanceGranted()**
  Returns 1 if the simulation has reached the time specified in `TimeAdvanceRequest`, 0 otherwise.

- **void TimeAdvanceRequest(double time)**
  Advances GVT by specified input time (in seconds). Sets a SPEEDES barrier at GVT + time.

- The following methods provide options for subscribing to SPEEDES objects.

  - **int Subscribe(char* namedData)**
    Subscribes to any internal SPEEDES data called namedData. Any event on any internal SPEEDES object that sends data called namedData via `SpHostUser::SendSubscribedData` will be sent to any external interface which subscribed to the data. This method always returns `SpHostRouterMsgs::SUBSCRIBE_STATUS_GOOD`.

  - **int SubscribeType(char* namedData, char* type)**
    This method is exactly the same as the previous method, except that this method subscribes to any namedData for objects of the same type, instead of on all objects. The named data, ObjProxy, is an internal SPEEDES reserved word and cannot be used.

  - **int SubscribeObject(char* namedData, char* objName)**
    This method is the same as the previous method, except that this method subscribes to a specific object instance, objName.

  - **UnSubscribe(char* name)**
    UnSubscribes from a named subscription type for all objects. Returns `SpHostRouterMsgs::UNSUBSCRIBE_STATUS_GOOD`.

  - **UnSubscribe(char* name, char* objName)**
    UnSubscribes from a named subscription type for a specific object instance called objName. Returns `SpHostRouterMsgs::UNSUBSCRIBE_STATUS_GOOD`.

  - **UnSubscribeType(char* name, char* type)**
    UnSubscribes from a named subscription type for the specified object type. Returns `SpHostRouterMsgs::UNSUBSCRIBE_STATUS_GOOD`.

- The following methods provide options for receiving subscribing data.

  - **char* ReceiveSubscribedData(char name, double& time, int& bytes, int& simObjGlobalId)**
  
  - **char* ReceiveSubscribedData(char* name, SpSimTime& time, int& bytes, int& simObjGlobalId)**

  Returns the named data specified by name, which was sent in SPEEDES via `SpHostUser::SendSubscribedData`, for which a subscription was made. Parameters time, bytes, and simObjGlobalId represent the time that the message was sent, the number of bytes in the message and the sending object’s global id, respectively. The user is responsible for deleting the returned data after use (i.e. delete []).
Returns any named data which was sent from SPEEDES via SpHostUser::SendSubscribedData, for which a subscription was made. Parameters name, time, bytes, and simObjGlobalId represent the name of the named data, time that the message was sent, number of bytes in the message, and the sending objects global id, respectively. The user is responsible for deleting the returned data after use (i.e. delete []).

- **Checkpoint(char* name = NULL)**
  Requests that the simulation perform a checkpoint.

### 6.1.2 See also

SpHostUser 3.21, SpStateMgr 6.2
6.2 SpStateMgr: External Module SPEEDES interface

6.2.1 Description

Class SpStateMgr is used to communicate with a SPEEDES simulation. The external module creates an instance of SpStateMgr, which then allows it access to all objects in the simulation that have a proxy. SpStateMgr allows the external module to receive committed or released internal simulation data and allows data to be sent into the SPEEDES simulation. The advancing of the simulation can also be controlled via the StateMgr.

- There are several methods to construct a SpStateMgr.
  
  - `SpStateMgr(char* fileName, int usingProxies = 1)`
    This constructor is used to play back a previous execution of an external module. Parameter, fileName, specifies the name of the file which contains the data to be played back. The playback files are created during external module executions by calling the method, SpStateMgr::RecordInputMessages or SpEmHostUser::RecordInputMessages.

  - `SpStateMgr(char* machineName, int port, int groupId, double timeLag, int usingProxies)`
    Creates a SpStateMgr that connects to the SpeedesServer. Parameters, machineName, port, and groupId, uniquely specify the SpeedesServer and SPEEDES simulation to which this SpStateMgr should connect. Parameter, timeLag, specifies the length of time in seconds that the external module is allowed to fall behind the SPEEDES simulation. This must be a non-zero value.

  - `SpStateMgr(SpDataParser* parser, double timeLag, int usingProxies = 1)`
    Creates a SpStateMgr that connects to the SpeedesServer. The parser passed in must have been initialized with speedes.par. The timeLag value specifies the length of time in seconds that the external module is allowed to fall behind the SPEEDES simulation. This must be a non-zero value. The speedes.par file is required so that the parameters, Group, MachineName, and Port can be extracted from the file so that the external module can connect to the correct SPEEDES simulation through the SpeedesServer.

- The following methods provide several options for subscribing to internal SPEEDES simulation data. Both proxy and data are sent via SpHostUser::SendSubscribedData (see Section 3.21).

  - `int Subscribe(char* subName, ...)`
    Subscribes to any message sent via any simulation object whose name is subName or a list of NULL-terminated names. These messages are sent from the simulation via SpHostUser::SendSubscribedData (see Section 3.21). Currently, this method always returns SpHostRouterMsgs::SUBSCRIBE_STATUS_GOOD.

  - `void SubscribeAll()`
    Subscribes to all object types. This could be a significant amount of data if the simulation contains many objects, so care should be taken when using this command

  - `int SubscribeData(char* subName, char* subType)`
    Subscribes to messages sent via the specific simulation object whose type is subType and
whose name is subName. These messages are sent from the simulation by method SpHost-
User::SendSubscribedData (see Section 3.21). This method returns SpHostRouterMsgs-
::SUBSCRIBE_STATUS_GOOD if subType is valid. Otherwise, it returns SpHostRouter-
Msgs::SUBSCRIBE_STATUS_BAD

- **void SubscribeObject(char* objectName, ...)**
  Subscribes to individual object instances within the simulation.

- **void SubscribeType(char* type, ...)**
  Subscribes to object types (i.e. proxies) where type matches the names of plugged in simu-
lation objects. List of arguments must be NULL-terminated.

- **int UnSubscribe(char* unSubName)**
  Unsubscribes to any object whose name is unSubName. When unsubscribing to objects, the
  name, unSubName, must be the same name used during the subscription process. In other w ords,
  you can not subscribe to all car object types and unsubscribe to object instance car2. Currently ,
  this method always returns SpHostRouterMsgs::UNSUBSCRIBE_STATUS_GOOD.

- **The following methods are used to define and schedule state manager local events.**

  - **void DefineNumberOfEvents(int numberOfEvents)**
    Defines the number of events in the SpStateMgr.

  - **void DefineEvent(char* nameOfEvent,
    int idOfEvent,
    void* (*ConstructFunc)(int),
    int sizeOfEvent,
    int numberOfEventsToCreate)**
    Any event processed in the state manager must of be defined and registered using this
    method with the state manger prior to use of that event. Parameter, nameOfEvent, is the
    string name by which the event will be known. There are two internal SPEEDES event
    names which can be used here. They are “DISCOVER_OBJECT” and “REFLECT_ATTRIB-
    UTES”. These two events allow the state manager to receive updates for any object that
    has been subscribed to at creation time and any proxy attribute updates, respectively. Param-
    eter, idOfEvent, is a unique integer value for each registered event. Parameter, (*Construct-
    Func)(int), points to a function which will create (i.e. new) the event to be processed. This
    must be a child of SpStateMgrEvent (see Section 6.3). Parameter, sizeOfEvent, specifies
    the size of the event to be processed and parameter, numberOfEventsToCreate, is the initial
    amount of events to be created on the free list.

  - **void ScheduleEvent(SpStateMgrEvent* event, SpSimTime time)**
    This method is used to schedule local events on the state manager.

- **The following methods provide several options for controlling and accessing simulation and local
time.**

  - **SpSimTime AdvanceTime(SpSimTime requestedTime = -1.0e20)**
    Advances time as far as possible without blocking. This call allows the local time to catch
    up with simulation time. It returns the time that it was able to advance to.

  - **SpSimTime GetCurrentTime()**
    Returns the local time for the state manager.

  - **SpSimTime GetGrantedTime()**
    Returns the time that the simulation granted us (i.e. this is simulation GVT).
6.2 SpStateMgr: External Module SPEEDES interface

- double GetTimeLag()
  Returns the current value for time lag.

- void GoToTime(SpSimTime inTime)
  Moves external module local time to inTime, processing all local events that occur during this time. It also moves the simulation barrier to the inTime + timeLag, thus letting simulation time advance if GVT was halted at the barrier.

- void SetTimeLag(double timeLag)
  Sets the amount of time the external module is allowed to fall behind the simulation. This number must be positive. The larger the value, the less likely the external module will affect the SPEEDES simulation. Small values for time lag could cause the SPEEDES simulation to hit the barrier set by the external module, causing GVT to quit advancing.

- The following methods provide several options for controlling and accessing object proxies that have been received.

  - SpObjProxy* GetFirstProxy()
    Returns the first object proxy on the list of received proxies, regardless of type.

  - SpObjProxy* GetFirstProxy(char* objectType)
    Returns the first object proxy whose type is objectType.

  - SpObjProxy* GetNextProxy()
    Returns the next proxy on the proxy list after GetFirstProxy was used. If method GetFirstProxy was used, then the proxy returned is not based on object type. If, on the other hand, method, GetFirstProxy(char* objectType), was used, then the next proxy returned will be of the same type specified previously.

  - int GetNproxies()
    Returns the number of proxies at the current state manager time.

  - int GetNproxies(char* objectType)
    Returns the number of object proxies of a given type at the current state manager time.

  - SpObjProxy* GetProxy(char* objectName)
    Returns the proxy requested by the instance name, objectName, at the current state manager time.

- SpEmHostUser* GetEmHostUser()
  Returns the EmHostUser (see Section 6.1).

- void RecordInputMessages(char* fileName)
  Records all received messages to a file called fileName. This file can then be used by the SpStateMgr in a playback mode.

- int SpeedesExecuting()
  Returns 1 if the simulation that the SpStateMgr connected to is still running. Otherwise, it returns 0.

- The following methods provide options for sending data into a SPEEDES simulation.
- int SendCommand(char* commandName,  
  int simObjGlobalId,  
  char* messageData,  
  int messageLength,  
  int returnStatus = 0)

  Sends a command called commandName to an object specified by simObjGlobalId. The  
  receiving simulation object must have an event called by the same name as specified by  
  commandName in order for this command to work. The parameter specifies any input data  
  that needs to be sent into SPEEDES with parameter, messageLength, specifying the number  
  of bytes to be sent. Input data can be NULL. When parameter, returnStatus, is equal to  
  0 (default), the return value is always SpHostRouterMsgs::COMMAND_STATUS_GOOD.  
  When parameter, returnStatus, is non-zero, then the return status could also be SpHost-  
  RouterMsgs::COMMAND_STATUS_BAD_OBJECT or SpHostRouterMsgs::COMMAND-  
  _STATUS_BAD_EVENT.

- int SendCommand(char* commandName,  
  char* objName,  
  char* messageData,  
  int messageLength,  
  int returnStatus = 0)

  This command is identical to the previous command, except that the object that the command  
  is sent to is specified by objName, instead of simObjGlobalId.

6.2.2 See also

SpHostUser 3.21, SpEmHostUser 6.1, and SpStateMgrEvent 6.3.
Class, SpStateMgrEvent, is used to create user-defined classes to process data that is received from a SPEEDES simulation. These classes typically process object proxy data (see Section 4.35) and well known data that the external module subscribed to (see Section 6.2) and received from a SPEEDES simulation.

- **SpStateMgrEvent()**
  Default constructor used when new events are defined.

- **virtual void Process()**
  This method is implemented by the user. Any work to be done by the local event needs to go in this method (e.g. computations on object proxy data).

- The following methods provide options for sending data into a SPEEDES simulation.
  - **int SendCommand(char* commandName,**
    int simObjGlobalId,**
    char* messageData,**
    int messageLength,**
    int returnStatus = 0)**
    Sends a command called commandName to an object specified by simObjGlobalId. The receiving simulation object must have an event called by the same name as specified by commandName in order for this command to work. The parameter messageData specifies any input data that needs to be sent into SPEEDES with parameter, messageLength, specifying the number of bytes to be sent. Input data can be NULL. When parameter, returnStatus, is equal to 0 (default), the return value is always SpHostRouterMsgs::COMMAND_STATUS_GOOD. When parameter, returnStatus, is non-zero, then the return status could also be SpHostRouterMsgs::COMMAND_STATUS_BAD_OBJECT or SpHostRouterMsgs::COMMAND_STATUS_BAD_EVENT.
  
  - **int SendCommand(char* commandName,**
    char* objName,**
    char* messageData,**
    int messageLength,**
    int returnStatus = 0)**
    This command is identical to the previous command, except that the object that the command is sent to is specified by objName instead of simObjGlobalId.

- **SpSimTime& GetTimeTag()**
  Returns event time.

- **void ScheduleEvent(SpSimTime inTime)**
  This method is used to schedule a local event on the state manager at time, inTime.

- **char* GetData()**
  Returns the message data sent from the SPEEDES simulation via SpHostUser::SendSubscribedData (see Section 3.21).
234  

- **int GetDataBytes()**  
  Returns the data bytes for the received message.

- **char* GetEventName()**  
  Returns the name of the event.

- **int GetSimObjGlobalId()**  
  Returns the global id for which object initiated this event.

### 6.3.2 See also

SpHostUser 3.21, SpEmHostUser 6.1, and SpStateMgr 6.2.
Chapter 7

Acronyms and Abbreviations

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFB</td>
<td>Air Force Base</td>
</tr>
<tr>
<td>ANSI</td>
<td>American National Standards Institute</td>
</tr>
<tr>
<td>API</td>
<td>Application Program Interface</td>
</tr>
<tr>
<td>AWACS</td>
<td>Airborne Warning and Control System</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DDM</td>
<td>Data Distribution Management</td>
</tr>
<tr>
<td>DIS</td>
<td>Distributed Interactive Simulation</td>
</tr>
<tr>
<td>DM</td>
<td>Distribution Management</td>
</tr>
<tr>
<td>ECI</td>
<td>Earth Centered Inertial</td>
</tr>
<tr>
<td>ECR</td>
<td>Earth Centered Rotating</td>
</tr>
<tr>
<td>GVT</td>
<td>Global Virtual Time</td>
</tr>
<tr>
<td>HLA</td>
<td>High Level Architecture</td>
</tr>
<tr>
<td>id</td>
<td>identifier</td>
</tr>
<tr>
<td>IEEE</td>
<td>Institute of Electrical and Electronics Engineers</td>
</tr>
<tr>
<td>OPSM</td>
<td>Object Proxy State Manager</td>
</tr>
<tr>
<td>PDES</td>
<td>Parallel Discrete-Event Simulation</td>
</tr>
<tr>
<td>SDRL</td>
<td>Subcontractor Data Requirement List</td>
</tr>
<tr>
<td>SOW</td>
<td>Statement Of Work</td>
</tr>
<tr>
<td>SPEEDES</td>
<td>Synchronous Parallel Environment for Emulation and Discrete-Event Simulation</td>
</tr>
<tr>
<td>TCP/IP</td>
<td>Transmission Control Protocol/Internet Protocol</td>
</tr>
<tr>
<td>XDR</td>
<td>External Data Representation</td>
</tr>
</tbody>
</table>
Index

ALT_CHUNK_SIZE, 14, 15
ASK_ARG, 41
BASE_ATTRIBUTE, 143–144, 199
BASE_DYNAMIC_ATTRIBUTE, 145–146, 166
BINARY_BUFFER_ATTRIBUTE, 147–148
C_DBQ_QUE, 79, 138
C_FREE_LIST, 11, 99–101
C_HASH, 68
C_ITEM, 11, 68, 99, 100, 138
C_QUEUE, 68, 99, 138
C_SQ_ITEM, 75, 79, 138–139, 142
CIRCULAR_ORT, 149
constant lookup time, 98, 102
CONSTANT_MOTION, 150
counter semaphore, 42
CrossProduct, 92
DEFINE_ASKEVENT, 20
DEFINE_ASKEVENT_INTERFACE_N_ARG, 20
DEFINE_ATTRIBUTE, 185–187, 196
DEFINE_AUTONOMOUS_EVENT, 20, 21
DEFINE_AUTONOMOUS_EVENT_0_ARG, 20
DEFINE_AUTONOMOUS_EVENT_1_ARG, 21
DEFINE_AUTONOMOUS_EVENT_2_ARG, 21
DEFINE_AUTONOMOUS_EVENT_8_ARG, 21
DEFINE_AUTONOMOUS_EVENT_N_ARG, 20
DEFINE_EVENT, 34, 52
DEFINE_EVENT_INTERFACE_0_ARG, 21
DEFINE_EVENT_INTERFACE_1_ARG, 21
DEFINE_EVENT_INTERFACE_2_ARG, 21
DEFINE_EVENT_INTERFACE_8_ARG, 21
DEFINE_EVENT_INTERFACE_N_ARG, 20–24
DEFINE_HANDLER, 25, 26
DEFINE_HANDLER_INTERFACE_0_ARG, 27
DEFINE_HANDLER_INTERFACE_1_ARG, 27
DEFINE_HANDLER_INTERFACE_2_ARG, 27
DEFINE_HANDLER_INTERFACE_N_ARG, 26, 28
DEFINE_INTERACTION_HANDLER, 25, 26
DEFINE_INTERFACE_HANDLER, 27, 28
DEFINE_LOCAL_EVENT_0_ARG, 22
DEFINE_LOCAL_EVENT_1_ARG, 22
DEFINE_LOCAL_EVENT_2_ARG, 22
DEFINE_LOCAL_EVENT_8_ARG, 23
DEFINE_LOCAL_EVENT_N_ARG, 22
DEFINE_MAN_3, 15, 66–67
DEFINE_MAN_31, 32, 39
DEFINE_SIMOBJ_EVENT, 21, 23, 24
DEFINE_SIMOBJ_EVENT_0_ARG, 24
DEFINE_SIMOBJ_EVENT_1_ARG, 24
DEFINE_SIMOBJ_EVENT_2_ARG, 24
DEFINE_SIMOBJ_EVENT_8_ARG, 24
DEFINE_SIMOBJ_EVENT_N_ARG, 22, 23
DEFINE_SIMOBJ_HANDLER, 26
DEFINE_SIMOBJ_INTERACTION_HANDLER, 26
DEFINE_SIMOBJ_INTERFACE_HANDLER, 27, 28
DEFINE_SIMOBJ_MGR_ID, 31
Difference, 92
DotProduct, 92
DOUBLE_ATTRIBUTE, 151
DYNAMIC_ATTRIBUTE, 143
DYNAMIC_COMPLEX_EXPONENTIAL, 152–153
DYNAMIC_DOUBLE_ATTRIBUTE, 152, 154–156, 157, 170
DYNAMIC_DOUBLE_CONSTANT, 157
DYNAMIC_EXPONENTIAL, 158
DYNAMIC_EXTRAPOLATE, 159
DYNAMIC_INT_ATTRIBUTE, 160, 161
DYNAMIC_INT_CONSTANT, 161
DYNAMIC_LOGICAL_ATTRIBUTE, 162, 163
DYNAMIC_LOGICAL_CONSTANT, 163
DYNAMIC_POLY_N, 164–165, 169, 170
  DYNAMIC_POLY_1, 164
  DYNAMIC_POLY_10, 164
  DYNAMIC_POLY_2, 164
  DYNAMIC_POLY_3, 164
  DYNAMIC_POLY_4, 164
  DYNAMIC_POLY_5, 164
  DYNAMIC_POLY_6, 164
  DYNAMIC_POLY_7, 164
  DYNAMIC_POLY_8, 164
  DYNAMIC_POLY_9, 164
DYNAMIC_POSITION ATTRIBUTE, 166–168, 171, 183, 188
DYNAMIC_POSITION_ITEM, 150
DYNAMIC_SPLINE_3, 169
DYNAMIC_SPLINE_6, 170
EARTH, 190
EARTH_to_ECI, 92
INDEX

EARTH_to_ECR, 92
ECI, 190
ECLI_to_EARTH, 92
ECI_to_ECR, 92
ECR, 190
ECR_to_EARTH, 92
ECR_to_ECI, 92, 93
Elliptical, 171–172
ExecuteSpeeds, 39, 106
EXTRAPOLATE_MOTION, 168, 173
F_SpProxyItem, 142, 195
free list, see C_FREE_LIST, see DEFINE_MEMPOOL
get_gsam, 93
GREAT_CIRCLE, 168, 177–178
Infinite loop, see Loop, Infinite
INT_ATTRIBUITE, 143, 179–180
InterestSpaces.par, 214, 216–220
Kilometers_to_Miles, 93
Kilometers_to_NauticalMiles, 93
LIST_ATTRIBUITE, 145, 181, 204
logical semaphore, 42
LOGICAL_ATTRIBUITE, 182
LOITER_MOTION, 183–184
Loop, Infinite, see Infinite loop
Magnitude, 92
Miles_to_Kilometers, 93
NauticalMiles_to_Kilometers, 93
NET_DOUBLE, 69, 113
NET_INT, 69, 113
Normalize, 92
OBJECT_ATTRIBUITE, 181, 185–187
ostream, 100, 115
ostrstream, 136
P_BEGIN, 40–45, 47, 51
P_END, 40–45, 47, 51
P_LV, 40–42, 44, 45, 47
P_VAR, 40–45, 47, 51
PLUGIN_EVENT, 39
PLUGIN_SIMOBJ, 32, 39
PLUGIN_SIMOBJ_MGR, 39
PLUGIN_MEMPOOL, 14, 66
point-to-point events, 40
POLY_N_MOTION, 188–189
   POLY_1_MOTION, 188
   POLY_2_MOTION, 188
   POLY_3_MOTION, 188
   POLY_4_MOTION, 188
   POLY_5_MOTION, 188
   POLY_6_MOTION, 188
   POLY_7_MOTION, 188
   POLY_8_MOTION, 188
   POLY_9_MOTION, 188
POSITION_ATTRIBUITE, 190
process model, 40–45
Query, 116
RBcerr, 77
RBcout, 77
RB_DEFINE_CLASS, 6–10, 31
RB_DEFINE_CLASS_PRIMITIVE, 8
RB_DELETE, 6
RB_DELETE_ARRAY, 6
RB_DOUBLE, 70, 151
RB_exostream, 71–72
RB_FREE_DELETE, 11, 197
RB_FREE_NEW, 11, 197
RB_hash, 75–76, 81
RB_INSERT, 81
RB_int, 13, 15, 73–74, 179
RB_memcpy, 11
RB_NEW, 6
RB_NEW_ARRAY, 6, 11
RB ostream, 77–78
RB_POP_BOT, 81
RB_POP_TOP, 81
RB_PUSH_BOT, 81
RB_PUSH_TOP, 81
RB_queue, 55, 75, 79–81, 138, 139, 142, 195
RB_REMOVE, 81
RB_SaveState, 11
RB_SpBinaryTree, 13, 79, 86–89
RB_SpBool, 71, 90
RB_SpCancelHandle, 18–19
RB_SpContainerFuncs.H, 75
RB_SpDefineClass.H, 6–10
RB_SpDynPtrArray, 85, 98
RB_SpFastRandom, 127
RB_SpFrameworkFuncs.H, 11–12, 197
RB_SpHashTree, 102–105
RB_SpList, 79, 109–110
RB_SpPriorityTree, 124–125
RB_SpRandom, 55, 126–127
RB_SpString, 82–83, 211
RB_voidPtr, 84–85
RBSendData, 11
RBSendSubscribedData, 11
RetractionHandle, 124
RHUMB_LINE, 191–192
S_SpHLA, 168, 187, 193–196, 199, 206, 215, 216, 218, 220
ScalarMultiply, 92
SpTag, 136–137
STRING_ATTRIBUTE, 143, 211
Sum, 92

VectorCopy, 92

WAIT, 40, 42
WAIT_FOR, 40, 42, 43, 47, 51
WAIT_FOR_COUNTER, 42, 43
WAIT_FOR_RESOURCE, 43, 44
WAIT_UNTIL, 44, 45