Connection Machine Run-Time System (CMRTS)
Architectural Specification
Version 7.2
Thinking Machines Confidential

William R. Swanson
March 29, 1993

CMRTS Architecture Designed by
Ken Crouch

©1993 Thinking Machines Corporation of Cambridge, Massachusetts. All rights reserved.
Connection Machine Run-Time System (CMRTS)
Architectural Specification
Version 7.2
Thinking Machines Confidential

William R. Swanson
March 29, 1993

CMRTS Architecture Designed by
Ken Crouch

©1993 Thinking Machines Corporation of Cambridge, Massachusetts. All rights reserved.
1 The CMRT Interface

1.1 Common CMRT Function Arguments .................................. 26
1.2 CMRT Initialization ..................................................... 26
   1.2.1 CMRT_init, CMRT_initialized .................................. 26
   1.2.2 CMRT_[set, get]_log2_subgrid_quantum ..................... 27
   1.2.3 CMRT_pe_[length/limit] ....................................... 27
   1.2.4 CMRT_version .................................................. 28
   1.2.5 CMRT_sizeof .................................................. 28
1.3 CMRT Environment Functions ......................................... 29
   1.3.1 CMRT_[enable/disable]_memory_fill, CMRT_memory_fill_enabled .... 29
   1.3.2 CMRT_[enable/disable]_send_verification ................... 29
1.4 CM Array Allocation/Deallocation .................................. 30
   1.4.1 CMRT_allocate_[heap/stack]_array .......................... 30
   1.4.2 CMRT_deallocate_[heap/stack]_array ....................... 30
1.5 Geometry Definition Functions ..................................... 31
   1.5.1 CMRT_intern{_[detailed_specific]}_array_geometry ........... 31
   1.5.2 CMRT_geometry_get_garbage_mask .......................... 32
   1.5.3 CMRT_print_array.geometry .................................. 32
1.6 CM Array Descriptors ................................................. 32
   1.6.1 CMRT_[allocate/deallocate]_descriptor ..................... 32
   1.6.2 CMRT_desc_get_[array_field, axis_field] .................... 33
<table>
<thead>
<tr>
<th>Section</th>
<th>Function Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.6.3</td>
<td>CMRT_desc_set[field]</td>
<td>35</td>
</tr>
<tr>
<td>1.6.4</td>
<td>CMRT_desc_get_array_increment</td>
<td>35</td>
</tr>
<tr>
<td>1.6.5</td>
<td>CMRT_desc_get_subgrid_axis_order</td>
<td>35</td>
</tr>
<tr>
<td>1.7</td>
<td>CM Memory Allocation/Deallocation</td>
<td>36</td>
</tr>
<tr>
<td>1.7.1</td>
<td>CMRT_allocate{physical_[heap/stack]_field</td>
<td>36</td>
</tr>
<tr>
<td>1.7.2</td>
<td>CMRT_deallocate{physical_[heap/stack]_field_through</td>
<td>36</td>
</tr>
<tr>
<td>1.7.3</td>
<td>CMRT_[get/set]_stack_pointer</td>
<td>37</td>
</tr>
<tr>
<td>1.7.4</td>
<td>CMRT_available_memory</td>
<td>37</td>
</tr>
<tr>
<td>1.7.5</td>
<td>CMRT_absolute_address_from_cm_location</td>
<td>37</td>
</tr>
<tr>
<td>1.7.6</td>
<td>CMRT_add_offset_to_cm_location</td>
<td>38</td>
</tr>
<tr>
<td>1.8</td>
<td>Send and Grid Addresses</td>
<td>38</td>
</tr>
<tr>
<td>1.8.1</td>
<td>CMRT_make_send_address</td>
<td>38</td>
</tr>
<tr>
<td>1.8.2</td>
<td>CMRT_my_send_address</td>
<td>38</td>
</tr>
<tr>
<td>1.8.3</td>
<td>CMRT_my_grid_coordinate</td>
<td>39</td>
</tr>
<tr>
<td>1.8.4</td>
<td>CMRT_deposit_grid_coordinate[_zero]</td>
<td>39</td>
</tr>
<tr>
<td>1.8.5</td>
<td>CMRT_extract_grid_coordinate</td>
<td>40</td>
</tr>
<tr>
<td>1.8.6</td>
<td>CMRT_scalar_make_send_address</td>
<td>41</td>
</tr>
<tr>
<td>1.8.7</td>
<td>CMRT_scalar_deposit_grid_coordinate[_zero]</td>
<td>41</td>
</tr>
<tr>
<td>1.8.8</td>
<td>CMRT_scalar_extract_grid_coordinate</td>
<td>42</td>
</tr>
<tr>
<td>1.8.9</td>
<td>CMRT_compress_send_address</td>
<td>42</td>
</tr>
<tr>
<td>1.8.10</td>
<td>CMRT_expand_send_address</td>
<td>42</td>
</tr>
<tr>
<td>1.9</td>
<td>Broadcast Functions</td>
<td>43</td>
</tr>
<tr>
<td>1.9.1</td>
<td>CMRT_broadcast_[type]</td>
<td>43</td>
</tr>
<tr>
<td>1.10</td>
<td>Array Read/Write Functions</td>
<td>43</td>
</tr>
<tr>
<td>1.10.1</td>
<td>CMRT_[read/write]<em>array_element</em>[type]</td>
<td>43</td>
</tr>
<tr>
<td>1.10.2</td>
<td>CMRT_[read/write]_array_element[into/from]_buffer</td>
<td>44</td>
</tr>
<tr>
<td>1.10.3</td>
<td>CMRT_[read/write]_array</td>
<td>44</td>
</tr>
</tbody>
</table>
1.11 General Communications Functions (Sends and Gets) ......................... 45
  1.11.1 CMRT_send, CMRT_send_[combiner] ................................. 45
  1.11.2 CMRT_get ...................................................... 45
1.12 Cross-Geometry Move ..................................................... 46
  1.12.1 CMRT_cross_geometry_move ....................................... 46
1.13 NEWS (Axis Shift) Communication ........................................ 47
  1.13.1 CMRT_news ...................................................... 47
  1.13.2 CMRT_eoshift{.vector} ........................................ 48
  1.13.3 CMRT_cshift{.vector} .......................................... 49
1.14 Scans/Reductions/Spreads ................................................ 50
  1.14.1 CMRT_scan_[combiner] ......................................... 50
  1.14.2 CMRT_reduce_[combiner] ....................................... 53
  1.14.3 CMRT_spread_copy ............................................. 53
  1.14.4 CMRT_multispread_copy ....................................... 54
1.15 Array Reduction (Global) Functions ...................................... 54
  1.15.1 CMRT_global_[combiner] ....................................... 55
  1.15.2 CMRT_global_project_[type] .................................. 56
1.16 Enumeration, Ranking, and Sorting ....................................... 56
  1.16.1 CMRTEnumerate ................................................. 56
  1.16.2 CMTRTRank, CMTSort ........................................... 57
1.17 Array Manipulation Utilities ............................................ 58
  1.17.1 CMRT_[firstloc/lastloc]{.axis} ................................ 58
  1.17.2 CMRT_[maxloc/minloc] ....................................... 59
  1.17.3 CMRT_[pack,unpack] ........................................... 59
  1.17.4 CMRT_project .................................................. 60
  1.17.5 CMRT_reshape ................................................ 60
1.18 Vector and Matrix Multiply Functions ................................... 61
1.18.1 CMRT_vector_matrix_multiply, CMRT_matrix_vector_multiply .......................... 61
1.18.2 CMRT_matrix_multiply .................................................................................. 61

1.19 Random Number Generation Functions ............................................................ 62
1.19.1 The RTS Random Number Generator ............................................................ 62
1.19.2 CMRT_random ................................................................................................ 63
1.19.3 CMRT_physical_random ............................................................................... 64
1.19.4 CMRT_initialize_random, CMRT_initialize_physical_random .................... 64
1.19.5 CMRT_deallocate_rng, CMRT_deallocate_physical_rng ............................... 65
1.19.6 CMRT_random_seed_location ........................................................................ 65

1.20 CM-2 Low-level Interface Functions ................................................................. 66
1.20.1 CMRT_copy_[to/from]_paris_field ................................................................ 66
1.20.2 CMRT_load_code ......................................................................................... 66
1.20.3 CMRT_push_address ..................................................................................... 67
1.20.4 CMRT_push_[type] ....................................................................................... 67

1.21 Indirect Addressing Functions .......................................................................... 68
1.21.1 CMRT_aref_ld .............................................................................................. 68
1.21.2 CMRT_aset_ld ............................................................................................. 68
1.21.3 CMRT_lookup_in_table ................................................................................. 69

1.22 Miscellaneous CMRT Functions ....................................................................... 70
1.22.1 CMRT_fp_check_safety_[pre/post] ................................................................. 70
1.22.2 CMRT_make_triplet_mask ............................................................................. 70

2 The CMCOM Interface ......................................................................................... 71
2.1 CMCOM Function Naming Conventions ............................................................. 71
2.2 CMCOM Function Arguments .......................................................................... 72
2.3 CMCOM Initialization ...................................................................................... 74
2.3.1 CMCOM_init ............................................................................................... 74
<table>
<thead>
<tr>
<th>Section</th>
<th>Function/Procedure Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.3.2</td>
<td>CMCOM_pe_[length,limit]</td>
<td>74</td>
</tr>
<tr>
<td>2.3.3</td>
<td>CMCOM_[attached, initialization_check, safe_cm_init]</td>
<td>74</td>
</tr>
<tr>
<td>2.4</td>
<td>Machine Geometries and Garbage Masks</td>
<td>75</td>
</tr>
<tr>
<td>2.4.1</td>
<td>CMCOM_intern{_specific}_geometry</td>
<td>75</td>
</tr>
<tr>
<td>2.4.2</td>
<td>CMCOM_print_geometry</td>
<td>76</td>
</tr>
<tr>
<td>2.4.3</td>
<td>CMCOM_create_garbage_mask</td>
<td>76</td>
</tr>
<tr>
<td>2.4.4</td>
<td>CMCOM_calculate_garbage_mask_size</td>
<td>76</td>
</tr>
<tr>
<td>2.5</td>
<td>CM Memory Management</td>
<td>77</td>
</tr>
<tr>
<td>2.5.1</td>
<td>CMCOM_allocate_stack_space, CMCOM_allocate_stack_words</td>
<td>77</td>
</tr>
<tr>
<td>2.5.2</td>
<td>CMCOM_available_memory</td>
<td>77</td>
</tr>
<tr>
<td>2.5.3</td>
<td>CMCOM_get_stack_pointer</td>
<td>78</td>
</tr>
<tr>
<td>2.5.4</td>
<td>CMCOM_get_heap_base</td>
<td>78</td>
</tr>
<tr>
<td>2.5.5</td>
<td>CMCOM_expand_heap</td>
<td>78</td>
</tr>
<tr>
<td>2.6</td>
<td>Address Arithmetic Functions</td>
<td>78</td>
</tr>
<tr>
<td>2.6.1</td>
<td>CMCOM_add_offset_to_cm_location[ ]</td>
<td>79</td>
</tr>
<tr>
<td>2.6.2</td>
<td>CMCOM_df_[lsw,msw].addr</td>
<td>79</td>
</tr>
<tr>
<td>2.6.3</td>
<td>CMCOM_[c,dc]_[realpart,imagpart].addr</td>
<td>79</td>
</tr>
<tr>
<td>2.7</td>
<td>Send and Grid Addresses</td>
<td>80</td>
</tr>
<tr>
<td>2.7.1</td>
<td>CMCOM_make_send_address</td>
<td>80</td>
</tr>
<tr>
<td>2.7.2</td>
<td>CMCOM_my_send_address, CMCOM_my_grid_coordinate</td>
<td>80</td>
</tr>
<tr>
<td>2.7.3</td>
<td>CMCOM_physical_pe_address</td>
<td>81</td>
</tr>
<tr>
<td>2.7.4</td>
<td>CMCOM_my_physical_grid_coordinate</td>
<td>81</td>
</tr>
<tr>
<td>2.7.5</td>
<td>CMCOM_deposit_grid_coordinate{_zero}</td>
<td>82</td>
</tr>
<tr>
<td>2.7.6</td>
<td>CMCOM_extract_grid_coordinate</td>
<td>83</td>
</tr>
<tr>
<td>2.7.7</td>
<td>CMCOM_scalar_make_send_address</td>
<td>84</td>
</tr>
<tr>
<td>2.7.8</td>
<td>CMCOM_scalar_deposit_[physical,subgrid].address</td>
<td>84</td>
</tr>
<tr>
<td>2.7.9</td>
<td>CMCOM_scalar_extract_[physical,subgrid].address</td>
<td>84</td>
</tr>
</tbody>
</table>
2.7.10 CMCOM.scalar_deposit_grid.coordinate{zero} .......................... 85
2.7.11 CMCOM.scalar_extract_grid.coordinate ................................. 85

2.8 Broadcast Functions ...................................................... 86
2.8.1 CMCOM.broadcast, CMCOM.physical.broadcast .......................... 86

2.9 Processor Read/Write Functions ......................................... 87
2.9.1 CMCOM.read_from_processor, CMCOM.physical.read_from_processor ... 87
2.9.2 CMCOM.write_to_processor, CMCOM.physical.write_to_processor ....... 87
2.9.3 CMCOM.physical.[read,write].byte.[from,to].processor ............... 88

2.10 Array Read/Write Functions ............................................ 89
2.10.1 CMCOM.transfer_array.[from,to].cm .................................. 89
2.10.2 CMCOM.transfer_vector.[from,to].cm .................................. 90
2.10.3 CMCOM.transfer_array.section.[from,to].cm .......................... 91

2.11 General Communication Function (Send and Get) ....................... 92
2.11.1 CMCOM.send.[combiner, log.combiner] ................................ 92
2.11.2 CMCOM.send.to.queue .................................................. 94
2.11.3 CMCOM.get ............................................................. 95

2.12 Cross-Geometry Move Function .......................................... 96
2.12.1 CMCOM.cross.geometry.move .......................................... 96

2.13 Rotate/Shift Communication Functions .................................. 97
2.13.1 CMCOM.shift{vector}, CMCOM.rotate{vector} .......................... 97
2.13.2 CMCOM.physical.rotate ............................................... 98

2.14 Scans/Reductions/Spreads .............................................. 99
2.14.1 CMCOM.scan.[combiner],CMCOM.scan.[log.combiner] ................ 99
2.14.2 CMCOM.scan.[c/dc].add (CM-2 version) ............................... 103
2.14.3 CMCOM.on.chip.pre.scan.[combiner] ................................ 104
2.14.4 CMCOM.on.chip.pre.scan.copy (non-CM-2 only) ...................... 105
2.14.5 CMCOM.on.chip.pre.scan.[log.combiner] .............................. 106
<table>
<thead>
<tr>
<th>Section</th>
<th>Function Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.14.6</td>
<td>CMCOM_onchip_pre_scan.[c/dc].add (CM-2 only)</td>
<td>106</td>
</tr>
<tr>
<td>2.14.7</td>
<td>CMCOM_on_chip_post_scan.[combiner]</td>
<td>107</td>
</tr>
<tr>
<td>2.14.8</td>
<td>CMCOM_on_chip_post_scan_copy (non-CM-2 only)</td>
<td>108</td>
</tr>
<tr>
<td>2.14.9</td>
<td>CMCOM_on_chip_post_scan.[log.combiner]</td>
<td>109</td>
</tr>
<tr>
<td>2.14.10</td>
<td>CMCOM_onchip_post_scan1c/dcLadd (CM-2 only)</td>
<td>109</td>
</tr>
<tr>
<td>2.14.11</td>
<td>CMCOM_physical_scan.[combiner]</td>
<td>110</td>
</tr>
<tr>
<td>2.14.12</td>
<td>CMCOM_physical_scan.[log.combiner]</td>
<td>111</td>
</tr>
<tr>
<td>2.14.13</td>
<td>CMCOM_physical_scan_copy</td>
<td>111</td>
</tr>
<tr>
<td>2.14.14</td>
<td>CMCOM_reduce.[combiner,log.combiner]</td>
<td>113</td>
</tr>
<tr>
<td>2.14.15</td>
<td>CMCOM_on_chip_reduce.[combiner,log.combiner]</td>
<td>114</td>
</tr>
<tr>
<td>2.14.16</td>
<td>CMCOM_on_chip_reduce.delivery</td>
<td>114</td>
</tr>
<tr>
<td>2.14.17</td>
<td>CMCOM_physical_reduce.[combiner]</td>
<td>115</td>
</tr>
<tr>
<td>2.14.18</td>
<td>CMCOM_physical_reduce.[log.combiner]</td>
<td>116</td>
</tr>
<tr>
<td>2.14.19</td>
<td>CMCOM_spread.[combiner,log.combiner]</td>
<td>117</td>
</tr>
<tr>
<td>2.14.20</td>
<td>CMCOM_spread_copy</td>
<td>118</td>
</tr>
<tr>
<td>2.14.21</td>
<td>CMCOM_multispread_copy</td>
<td>118</td>
</tr>
<tr>
<td>2.14.22</td>
<td>CMCOM_on_chip_spread_copy.gather</td>
<td>119</td>
</tr>
<tr>
<td>2.14.23</td>
<td>CMCOM_on_chip_spread_delivery</td>
<td>119</td>
</tr>
<tr>
<td>2.14.24</td>
<td>CMCOM_physical_spread.[combiner]</td>
<td>119</td>
</tr>
<tr>
<td>2.14.25</td>
<td>CMCOM_physical_spread.[log.combiner]</td>
<td>120</td>
</tr>
<tr>
<td>2.14.26</td>
<td>CMCOM_physical_spread_copy</td>
<td>120</td>
</tr>
<tr>
<td>2.14.27</td>
<td>CMCOM_physical_spread_copy_from_leader_always</td>
<td>120</td>
</tr>
<tr>
<td>2.15</td>
<td>Array Reduction (Global) Functions</td>
<td>121</td>
</tr>
<tr>
<td>2.15.1</td>
<td>CMCOM_global.[combiner]</td>
<td>121</td>
</tr>
<tr>
<td>2.15.2</td>
<td>CMCOM_on_chip_global.[combiner]</td>
<td>122</td>
</tr>
<tr>
<td>2.15.3</td>
<td>CMCOM_physical_global.[combiner]</td>
<td>123</td>
</tr>
<tr>
<td>2.16</td>
<td>Rank (Sorting) Functions</td>
<td>124</td>
</tr>
</tbody>
</table>
2.1.6.1 CMCOM_{u,s,f,df,ds}.rank, CMCOM_{u,s,f,df,ds}.sort ........................................... 124
2.17 CM-2 RTS Interface to Paris ...................................................... 126
2.17.1 CMCOM_copy_{to/from}.paris_field ........................................... 126

3 The CMIP Interface ................................................................. 127
3.1 CMIP Function Arguments ......................................................... 128
3.2 Array Copying Functions .......................................................... 129
  3.2.1 CMIP.move ................................................................. 129
  3.2.2 CMIP.{u,du}.merge_with_constant ...................................... 130
3.3 Logical Arithmetic Functions ..................................................... 131
  3.3.1 CMIP.logop, CMIP.lognot ................................................ 131
3.4 Arithmetic Functions ............................................................. 132
  3.4.1 CMIP.{type}.[add,subtract,product,div,mod,truncate] ..................... 132
  3.4.2 CMIP.{type}.[max,min] .................................................. 133
  3.4.3 CMIP.{s,f,df}.abs ..................................................... 133
  3.4.4 CMIP.{f,df}.sqrt ...................................................... 134
  3.4.5 CMIP.u.shift.constant .................................................. 134
3.5 Comparison Instructions ......................................................... 135
  3.5.1 CMIP.{u,f,df,du}.eq, CMIP.{u,s}.lt ..................................... 135
3.6 Comparison-Preserving Conversion Functions .................................. 135
  3.6.1 CMIP.convert.{s,f,df}.[to,from].unsigned_format .......................... 135
3.7 Random Number Generation Functions ......................................... 136
  3.7.1 CMIP.random.{df.to.s,df.to.ss} ......................................... 136
3.8 Indirect Addressing Functions .................................................. 137
  3.8.1 CMIP.aref.ld ............................................................. 137
  3.8.2 CMIP.aref.ld.any_order ................................................ 139
  3.8.3 CMIP.aset.ld ............................................................. 139
3.8.4 CMIP_aset ld any_order
3.8.5 CMIP_lookup_in_table
3.9 Mask Construction Functions
3.9.1 CMIP_pack_mask
3.9.2 CMIP_unpack_mask{ conditional}
3.9.3 CMIP_compress_mask
3.9.4 CMIP_pack_and_logand_mask
3.9.5 CMIP_make_triplet_mask
3.10 Scatter/Gather Functions
3.10.1 CMIP [scatter.gather]
3.11 Send Address Operations
3.11.1 CMIP [compress,expand] send_address

4 CMRTS Geometries

4.1 Array Geometries — Essential Concepts
4.1.1 Array Subgrids and Physical Grids
4.1.2 Efficient Subgrid Layout
4.1.3 Not All Arrays Are So Cooperative
4.1.4 Processing Element Numbers and Axis Sequences
4.1.5 Array Element Addressing
4.1.6 Off-Chip Bits and Axis Sequences
4.1.7 Send-Order and NEWS-Order Axes
4.1.8 Subgrid Structure
4.1.9 Serial Axes — Guaranteeing PE Locality
4.2 Handling Arrays of Arbitrary Size — Garbage Masks
4.3 Geometries with Non-contiguous Off-chip Bits
4.4 Vector Unit Array Layout
5.1.2 IBCLR (Integer) ..................................................... 191
5.1.3 IBITS (Integer) .................................................... 191
5.1.4 IBSET (Integer) ................................................... 192
5.1.5 ACOS (Float) ...................................................... 192
5.1.6 CCOS (Complex) ................................................ 192
5.1.7 CDCOS (Dcomplex) ............................................. 193
5.1.8 COS (Float) ....................................................... 193
5.1.9 COSH (Float) .................................................... 193
5.1.10 DACOS (Double) .............................................. 194
5.1.11 DCOS (Double) ............................................... 194
5.1.12 DCOSH (Double) ............................................ 194
5.1.13 CDEXP (Dcomplex) .......................................... 195
5.1.14 CEXP (Complex) ............................................. 195
5.1.15 DEXP (Double) ................................................. 195
5.1.16 EXP (Float) .................................................. 196
5.1.17 AINT (Float) ................................................ 196
5.1.18 ANINT (Float) .............................................. 196
5.1.19 DINT (Double) ............................................... 197
5.1.20 DNINT (Double) ............................................ 197
5.1.21 CDLOG (Dcomplex) ........................................ 197
5.1.22 CLOG (Complex) ........................................... 198
5.1.23 DLOG (Double) .............................................. 198
5.1.24 LOG (Float) ................................................ 198
5.1.25 DLOG10 (Double) ........................................ 199
5.1.26 LOG10 (Float) ............................................. 199
5.1.27 C.C_POWER (Complex) ................................... 199
5.1.28 C.F_POWER (Complex) .................................... 200
5.1.29 C_IC_POWER (Complex) .................. 200
5.1.30 C_LPOWER (Complex) .................. 201
5.1.31 D_DPOWER (Double) .................. 201
5.1.32 D_ICPOWER (Double) .................. 202
5.1.33 D_LPOWER (Double) .................. 202
5.1.34 F_FPOWER (Float) .................. 202
5.1.35 F_ICPOWER (Float) .................. 203
5.1.36 F_LPOWER (Float) .................. 203
5.1.37 I_LPOWER (Integer) .................. 203
5.1.38 Z_DPOWER (Dcomplex) .................. 204
5.1.39 Z_ICPOWER (Dcomplex) .................. 204
5.1.40 Z_LPOWER (Dcomplex) .................. 204
5.1.41 Z_ZPOWER (Dcomplex) .................. 205
5.1.42 ISHFT (Integer) .................. 205
5.1.43 ISHFTC (Integer) .................. 205
5.1.44 ASIN (Float) .................. 206
5.1.45 CDSIN (Dcomplex) .................. 206
5.1.46 CSIN (Complex) .................. 206
5.1.47 DASIN (Double) .................. 207
5.1.48 DSIN (Double) .................. 207
5.1.49 DSINH (Double) .................. 207
5.1.50 SIN (Float) .................. 208
5.1.51 SINH (Float) .................. 208
5.1.52 CDSQRT (Dcomplex) .................. 208
5.1.53 CSQRT (Complex) .................. 209
5.1.54 ATAN (Float) .................. 209
5.1.55 ATAN2 (Float) .................. 209
5.1.56 DATAN (Double) .................................................. 210
5.1.57 DATAN2 (Double) ............................................. 210
5.1.58 DTAN (Double) ................................................... 210
5.1.59 DTANH (Double) ................................................. 211
5.1.60 TAN (Float) .................................................... 211
5.1.61 TANH (Float) ................................................... 211

5.2 DPEAC routines ...................................................... 212
5.2.1 BTEST (single precision) ..................................... 212
5.2.2 KBTEST (double precision) .................................. 212
5.2.3 IBCLR (single precision) ..................................... 213
5.2.4 KBCLR (double precision) .................................... 213
5.2.5 IBITS (single precision) ...................................... 213
5.2.6 KBITS (double precision) ..................................... 214
5.2.7 KDIV (double precision) ....................................... 214
5.2.8 IBSET (single precision) ..................................... 214
5.2.9 KBSET (double precision) ..................................... 215
5.2.10 ACOS (single precision) ..................................... 215
5.2.11 CCOS (single precision) ..................................... 215
5.2.12 CDCOS (double precision) ................................. 216
5.2.13 COS (single precision) ....................................... 216
5.2.14 COSH (single precision) .................................... 216
5.2.15 DACOS (double precision) ................................. 217
5.2.16 DCOS (double precision) ................................. 217
5.2.17 DCOSH (double precision) ............................... 217
5.2.18 CDEXP (double precision) ............................... 218
5.2.19 CEXP (single precision) ................................. 218
5.2.20 DEXP (double precision) ................................. 218
5.2.21 EXP (single precision) ........................................ 219
5.2.22 AINT (single precision) ....................................... 219
5.2.23 ANINT (single precision) ...................................... 219
5.2.24 DINT (double precision) ....................................... 220
5.2.25 DNINT (double precision) .................................... 220
5.2.26 CDLOG (double precision) ................................... 220
5.2.27 CLOG (single precision) ...................................... 221
5.2.28 DLOG (double precision) .................................... 221
5.2.29 LOG (single precision) ........................................ 221
5.2.30 DLOG10 (double precision) ................................ 222
5.2.31 LOG10 (single precision) .................................... 222
5.2.32 C.C_Power (single precision) ............................... 222
5.2.33 C.IC_Power (single precision) .............................. 223
5.2.34 C.I_Power (single precision) ............................... 223
5.2.35 C.K_Power (single precision) ............................... 223
5.2.36 D.D_Power (double precision) .............................. 224
5.2.37 D.IC_Power (double precision) ............................. 224
5.2.38 D.I_Power (double precision) .............................. 224
5.2.39 D.K_Power (double precision) ............................. 225
5.2.40 F.F_Power (single precision) .............................. 225
5.2.41 F.IC_Power (single precision) ............................. 225
5.2.42 F.I_Power (single precision) .............................. 226
5.2.43 F.K_Power (single precision) .............................. 226
5.2.44 L.I_Power (single precision) .............................. 226
5.2.45 Z.IC_Power (double precision) ............................ 227
5.2.46 Z.I_Power (double precision) ............................. 227
5.2.47 Z.Z_Power (double precision) ............................. 227
5.2.48 Z_K_POWER (double precision) ........................................ 228
5.2.49 K_K_POWER (double precision) ........................................ 228
5.2.50 ISHFT (single precision) .............................................. 228
5.2.51 KSHFT (double precision) ............................................ 229
5.2.52 ISHFTC (single precision) ............................................ 229
5.2.53 KSHFTC (double precision) ........................................... 229
5.2.54 ASIN (single precision) ................................................ 230
5.2.55 CDSIN (double precision) ............................................. 230
5.2.56 CSIN (single precision) ................................................ 230
5.2.57 DASIN (double precision) ............................................. 231
5.2.58 DSIN (double precision) ................................................ 231
5.2.59 DSINH (double precision) ............................................. 231
5.2.60 SIN (single precision) ................................................ 232
5.2.61 SINH (single precision) ................................................ 232
5.2.62 CDSQRT (double precision) ........................................... 232
5.2.63 CSQRT (single precision) .............................................. 233
5.2.64 ATAN (single precision) .............................................. 233
5.2.65 ATAN2 (single precision) ............................................. 233
5.2.66 DATAN (double precision) ............................................ 234
5.2.67 DATAN2 (double precision) ........................................... 234
5.2.68 DTAN (double precision) .............................................. 234
5.2.69 DTANH (double precision) ............................................ 235
5.2.70 TAN (single precision) ................................................ 235
5.2.71 TANH (single precision) ................................................ 235
0 The CM Run-Time System (CMRTS)

PLEASE NOTE: The CM Run-Time System is *internal* software — it is typically referenced only by CM compilers and libraries. The direct use of CMRTS routines by CM programmers is not recommended. In short, if you’re not a CM compiler, you probably should not be writing code that directly uses this level of the CM software.

0.1 Introduction

This document describes the Connection Machine Run-Time System (commonly abbreviated as CMRTS, or just RTS). The CMRTS is a set of low-level CM code libraries that are used for defining and manipulating parallel data structures in CM memory.

The functions in these libraries are designed to allocate CM memory and manipulate its contents “at run-time”, that is, during the execution of CM programs.

This document is divided into six major sections:

- This Introduction, which presents essential concepts and programming issues
- The CMRT layer — functions and macros
- The CMCOM layer — functions and macros
- The CMIP layer — functions and macros
- CMRTS Geometries — CMRT and CMCOM geometries
- The CMTRA layer — low-level (CMIS/DPEAC) arithmetic routines

Each subsection that describes one of the software layers includes an introduction presenting function and data type information specific to that software layer.

Acknowledgements: The CMRTS software is based on the CMRTS architecture design defined by Ken Crouch, and this document incorporates some material from earlier RTS spec documents written by Ken Crouch, David Gingold, and Cliff Lasser.

Functional descriptions and many other details have been provided by Bob Lordi, Jerry Callen, Mike McKenna, Marco Zagha, Richard Shapiro, and Alan Mainwaring. Many thanks to these people and to everyone who has reviewed this document in whole or in part and sent back detailed commentary!
0.2 The CMRTS Software

The CMRTS is divided into three main libraries of functions:

CMRT — The CM Run-Time Library.
CMCOM — The CM Communications Library.
CMIP — The CM In-Processor Library.

For the CM-2/200 and the CM-5 with vector units, there is an additional library:

CMTRA — The CM Transcendentals Library.

CMRTS Software Libraries

<table>
<thead>
<tr>
<th>CMRT</th>
<th>CMCOM</th>
<th>CMIP</th>
<th>CMTRA</th>
</tr>
</thead>
</table>

The CMRT layer is the topmost layer of the RTS software, and represents an "external," machine-independent interface for the RTS. CMRT functions provide access to all CM operations defined in the RTS.

The CMCOM and CMIP layers are "internal" support software for the CMRT layer. CMCOM functions provide access to CM communication operations (sends, scans, etc.). CMIP functions provide access to in-processor operations (arithmetic and logical computations, etc.).

The CMTRA library provides low-level transcendental operations (sin, cos, log, etc.) implemented directly in DPEAC assembly code. The CMTRA library exists primarily to support CM language compilers.

Usage Note: The CMRT layer is the "preferred" entrypoint for the RTS. CMRT functions hide most of the machine-dependent nature of the lower software layers, and generally provide a "cleaner" interface. You should only call CMCOM and CMIP functions directly if you have a good reason to do so — for example, if you wish to bypass the small overhead of the data structures used by CMRT for describing and manipulating arrays.
0.2.1 CMRTS Software Versions

The CMRTS software exists in distinct versions for the following CM configurations:

- **CM-2/200** — The CM-2 or CM-200 with floating-point hardware
- **CM-5/SPARC** — The CM-5 without vector units
- **CM-5/VU** — The CM-5 with vector units
- **CMSIM** — The "CM Simulator" library

The CMRT software interface is hardware-independent, and contains the same functions in all RTS versions. The CMCOM and CMIP layers are hardware-dependent — the functions they provide differ between the different versions, and in a few cases a single function has different argument lists for different software versions.

**Note:** The CM-5/SPARC version of the RTS software can also be used on CM-5 systems that have vector units. In this case, the vector units are ignored; all RTS functions execute as they would on a CM-5 without vector units (with a corresponding loss of performance).

0.2.2 The CM Simulator

The CMSIM library includes nearly all of the functions of the other RTS versions. Specifically, it most closely matches the CM-2 and CM-5/SPARC versions of the RTS in functionality. However, the CMSIM RTS functions are defined in such a way that they execute entirely on the partition manager (front end) or on a compatible workstation. Thus, compiling code using the CMSIM is effectively like compiling it on a single-node CM. The CM Simulator can be used to prototype and test code when CM hardware is not available.

0.2.3 Getting at the RTS

CM programs that call CMRTS functions are typically written in high-level languages (such as CM Fortran, C*, and *Lisp programs). In these cases, the CM compiler for the language generates all necessary calls to the RTS. It is also possible for knowledgeable programmers to include direct CMRTS calls in CM programs. To have this kind of direct access to the CMRTS libraries in a program, you must include the header file `cmrt.h`:

```c
#include <cm/cmrt.h>
```

**Note:** The CMRTS libraries are, with one exception, implemented in the C programming language. Thus all function descriptions and code examples in this document, such as the above `#include` line, are presented in C code format.)
0.3 CM Hardware Terminology

While the descriptions in this document are usually machine-independent, describing the RTS often requires references to specific CM hardware components ("partition manager," "processing nodes," etc.) This is complicated by the use, in the CM-2 and CM-5 machine models, of different names for components that serve roughly similar functions in each CM model:

**CM-2 Hardware:** (both CM-2 and CM-200)

- **Front End** — The front-end computer that executes CM-2 programs.
- **Processors** — The bit-serial/floating-point processing units of the CM-2.

**CM-5 Hardware:** (with and without vector units)

- **Partition Manager** (PM) — A control processor within the CM-5 that manages a partition of processing nodes.
- **Processing Nodes** (or just nodes) — The group of processing units within the CM-5 that is associated with a given PM.
- **Vector Unit** (VU) — One of the four vector processing chips installed on each processing node in a CM-5 that has vector units.

### CM Hardware Layout

(CM-2/200 and CM-5 without VUs)

![CM Hardware Layout (CM-2/200 and CM-5 without VUs)](image)

(CM-5 with VUs)

![CM Hardware Layout (CM-5 with VUs)](image)

This document refers to CM components generically using CM-5 terminology — for example, "partition manager" (PM) always refers to the front-end or control processor, and "node" always refers to one of the associated processing units. Unless otherwise noted, these generic descriptions apply equally to the CM-2/200 and the CM-5.
0.3.1 The Vector Unit Hardware

The CM-5 vector units (VUs) are custom vector arithmetic processors added to each node of the CM-5. Currently, there are four VUs per node, each with its own separate bank of memory.

These four VUs are actually implemented by only two VU chips, each of which contains the hardware necessary to simulate two "virtual" VUs.

Because of this feature of the hardware design, data communication between VUs is most efficient for the two VUs that share a single chip, next most efficient for VUs on separate chips in the same node, and somewhat less efficient for data transfers between nodes.

0.3.2 Processing Element (PE)

In this document, the term "processing element" (or PE) is used generically to refer to the minimal hardware component of the CM used for parallel computation. This is defined differently for each of the CM configurations:

- **CM-2**: A single Weitek FP chip and 32 bit-serial processors.
- **CM-5/SPARC**: A processing node, that is, one SPARC/NI chip unit.
- **CM-5/VU**: One of the four vector units associated with each SPARC/NI unit.
- **CMSIM**: The single processor of the workstation running your program.

It is important for you to keep this distinction in mind, especially in the two CM-5 cases — since there are 4 VUs on each processing node, programs compiled with the CM-5/VU version of the RTS software effectively have four times as many "processing elements" to work with.
0.3.3 CM Memory Management

There are three kinds of memory in a CM:

- PM memory: The memory of the PM
- node memory: All memory accessible on a single processing node.
- VU memory: The portion of node memory that is assigned to the VUs.

The term "parallel memory" refers to the region of node memory used to store parallel data. This is either VU memory (if VUs are present), or a specially allocated region of node memory.

PE Memory Layout

<table>
<thead>
<tr>
<th>Stack</th>
<th>Gap</th>
<th>Heap</th>
</tr>
</thead>
</table>

Top of processing element memory

Bottom of processing element memory

Bottom of stack

Top of heap

In either case, each processing element of the CM is assigned its own region of parallel memory. The RTS divides this PE memory region into stack and heap areas, as shown in the diagram above. There is typically a "gap" of free space between the stack and heap, into which the two storage areas expand.

Note: Depending on the version of the CMRTS software in use, the locations of the stack and heap areas in PE memory may be reversed.

0.3.4 Array Allocation Vs. Memory Allocation

The RTS provides a set of functions (and macros, for performance) that allocate chunks of memory in either the stack or the heap. These functions allow allocation of memory either directly in terms of words, or indirectly, in terms of the amount of memory needed to store an array of a given shape and data type.

The CMRT layer provides array allocation functions. These functions handle all the details of allocating an array of a given size and data type, including reserving CM memory, constructing appropriate array descriptors, etc. They require a predefined array geometry object (described below) and a data type as arguments, and return an array descriptor of type CMRT_desc_t that refers to the newly allocated array.
The CMCOM and CMIP layers provide memory allocation functions. These functions allocate memory in terms of a given number of words or bytes. They require a number of words as an argument, and return an address object of type CMCOM_cm_address_t that indirectly refers to the memory region's address. (Indirectly, because address objects must be further decoded to get the actual memory address — this layer of indirection is included to allow the use of address translation mechanisms in the CM-5/VU version of the RTS.)

Usage Note: The CMRT array allocation functions are the preferred method for allocating and freeing parallel memory. The CMRT layer includes functions that allocate parallel memory in terms of words, but these functions are provided mainly for symmetry with the corresponding CMCOM operators.

0.4 The Run-Time System — Essential Concepts

The RTS has two main purposes:

- To provide a standard set of CM functions for allocating and deallocating parallel memory on the in the form of multidimensional arrays.
- To provide a machine-independent interface to the features of the CM hardware, allowing parallel computation and communication to be performed on these multidimensional arrays.

This section presents an overview of the RTS software layers, and defines a number of terms and concepts that are essential for understanding and using the RTS.

0.4.1 Arrays, Subgrids, and Geometries

Internally, the RTS represents parallel array data in three ways:

- **CM arrays**, the actual regions of parallel data stored on the CM
- **machine arrays**, which are defined and used at the CMCOM/CMIP level
- **CMRT arrays**, which are defined and used at the CMRT level

Note for CMF and C* Users: High-level languages use CMRT arrays to store parallel data on the CM.

A **CM array** is a region of memory reserved in the memory of each PE for the purpose of storing array elements. This memory region has the same position and size for each PE, so a CM array can be viewed as a “stripe” across the combined memory of the CM processing elements.
On the CM-2/200 and CM-5 without vector units, arrays are laid out as follows:

**CM Memory Layout**  
(CM-2/200 or CM-5 without VUs)

- **Nodes**: P P P P P P P P P P P P
- **Node Memory**: 
- **CM Array**: [Diagram showing CM Array Data]

On the CM-5 with vector units, arrays are laid out as follows:

**CM Memory Layout**  
(CM-5 with VUs)

- **Nodes**: P P P P P P P P P
- **Vector Units**: V U U V U U V U U V U U V U U V U U V U U
- **VU Memory**: 
- **CM Array**: [Diagram showing CM Array Data]

Each PE's memory holds a small segment, or *subgrid* of the array. The subgrids of an array all have the same shape and size, and each subgrid holds a fixed number of array elements. The array elements themselves all have the same *element size*, or length in bytes (usually one or two words, depending on the data type).

The layout of an array's elements in PE memory (in other words, the correspondence between a given array element and its location in the memory of a particular PE) is referred to as the array's *geometry*.

**Terminology Note:** This distribution of data across the processors of the CM is referred to in parallel computing literature as the *decomposition* of the data. The CM supports *blocked decomposition* (one contiguous subgrid per PE) as opposed to *cyclic decomposition* (axes "striped" across PE memories).
The geometry of an array is specified by a geometry object — a data structure that describes both the size and shape of the array, and the size and shape of its subgrids. (Part of the purpose of the RTS is to internally manage the data structures defining arrays and geometries, making these details as transparent as possible to high-level languages.)

There are two kinds of geometries used in the RTS:

- *machine geometries*, used at the CMCOM/CMIP level
- *array geometries*, used at the CMRT level

Each of these geometry types corresponds to an RTS array type, as described below.

**Note:** This introduction presents only a brief overview of geometries and their use. For more detail, please refer to Section 4, which describes the geometry management features of the RTS.

### 0.4.2 Machine Arrays and Machine Geometries

To describe a CM array, only four pieces of information are needed:

1. The *address* of the array (its starting position in the memory of each PE)
2. The *element size*, or number of bytes in each array element
3. The *increment* in bytes between neighboring elements in PE memory
4. The *geometry* of the array

The combination of a CM memory address, an element size, and an increment is referred to as a *machine array*. Thus, a machine array consists of the following pieces of data:

- A CM memory address, of type `CMCOM_cm_address_t.`
- A element size, of type `int4`
- A increment, of type `int4`

**For the Curious:** The element size and increment of a CM array are typically the same. However, some RTS functions allow you to specify a different element size and/or increment for an array; this causes the function to "stride" through the array data in useful patterns.
For any machine array, there is a corresponding *machine geometry* that defines the layout of data in the array. The machine geometry specifies the machine array's rank, dimension lengths, and the shape and size of its subgrids. The machine geometry of an array is specified by a *machine geometry object*. The components of machine geometry objects are defined in Section 4. (The reason the machine geometry is a separate data structure is that many machine arrays can share the same geometry.)

CMCOM and CMIP functions take machine arrays and machine geometries as arguments, and thus manipulate CM arrays at the lowest possible level. However, this has two drawbacks:

- Using three separate pieces of information to refer to a machine array is somewhat unwieldy. (CMCOM and CMIP functions have large numbers of arguments for just this reason.)
- Machine arrays are too closely tied to the structure of the CM hardware (the number of processing nodes, etc.) to be able to represent all possible shapes and sizes of arrays.

For these reasons, the CMRT layer is built on top of CMCOM and CMIP, and defines its own arrays using a method that provides a layer of abstraction, hiding the details of machine arrays and geometries.

### 0.4.3 CMRT Arrays and Array Descriptors

At the CMRT level, an *array* consists of a number of components:

- A machine array that contains the actual array data
- An *array geometry* that specifies the layout of data in the array (See Section 0.4.4 below.)
- Miscellaneous information about the array's rank, size, type, etc.

The components of a CMRT array are kept in an *array descriptor* data structure, which has slots for each of the important components of the array. Specifically, an array descriptor is a structure of type `CMRT_desc_t` that contains the following components:

- A *CM memory location*, of type `CMRT_cm_location_t`
- An element size, and an increment, both of type `int4`

  **Note:** The above three components are the CMRT equivalent of a CMCOM/CMIP machine geometry. The function `CMRT_absolute_address_from_cm_location` converts a CM memory location into a `CMCOM_cm_address_t` value.
- An element type, of type `CMRT_element_type_t`

  This is an enumerated type that specifies the array's element type by name. The currently defined values are listed in Section 0.7 below.
- An array geometry, of type `CMRT_array_geometry_t`
  The array geometry provides most of the information about the array's layout, including its rank and axis lengths. The components of an array geometry are described in Section 4.
- An `is_modified` slot, of type `log4`, which is used by high-level languages to indicate whether an array's contents have been modified.
- A `home` slot, of type `CMRT_home_t`, which indicates where the array is stored: in node memory or in the memory of the partition manager. (This slot is used by higher levels of software to define array descriptors referring to arrays stored on the partition manager.)
- A number of other structure slots that are used for internal information.

CMRT functions use array descriptors to pass all information about a single array as a single argument. CMRT functions thus have fewer arguments and can be more "generic," leaving details of element types and array layouts to the machine-oriented CMCOM and CMIP functions.

### 0.4.4 Array Geometries and Garbage Masks

As described in Section 4, array geometries are distinct from machine geometries. An array geometry includes a machine geometry as one of its components, and adds to it a set of axis lengths and a "garbage mask" that together indicate which elements of the machine array actually contain "real" array data.

This distinction allows arrays with arbitrary axis lengths to be stored and accessed efficiently, and also allows several different arrays with the same overall shape to share a machine geometry, even though their actual sizes may be different.

**Note:** CMRT functions get an array's garbage mask from the array's geometry object. CMCOM functions typically require that the garbage mask be provided as a separate argument.

### 0.4.5 Mask Arrays

Functions that operate on an entire array typically have a `mask` argument, which is used to mask out specific elements of an array during the operation. The actual effect of the mask — that is, whether it selects source elements to be operated on or selects destination elements to be modified — depends on the operation being performed.

Any function with a `mask` argument may be passed the value `NULL` instead of a mask array, to indicate that no masking is required. In most cases this allows a more efficient non-masking algorithm to be used. (CMCOM operations also require a mask increment argument — this can be specified as 0.)
The data type of the mask array depends on the RTS version:

- **CM-2:** The mask argument is an array of \( \log_4 \) values. A LOG_TRUE value (all 1s) indicates an active array element, and a LOG_FALSE value (all 0s) indicates an inactive array element. Only active array elements participate in a masked operation.

- **CM-5/SPARC:** The mask argument is an array of unsigned word-length integers, but only the least significant byte of each word is used. A non-zero byte value indicates an active element, and a zero value indicates an inactive element. Only active array elements participate in a masked operation.

- **CM-5/VU:** The mask argument is either a packed or unpacked array. See the discussion of these array cases below.

Be careful not to confuse these user mask array arguments with the garbage mask arrays used in RTS geometry objects. Users supply user mask arrays as arguments to functions — garbage masks are generated and managed internally by the RTS. (Incidentally, garbage masks have the same packed format as packed user masks in the CM-5/VU RTS.)

**For CM-2 Programmers:** The RTS uses the mask argument to implement processor context. In earlier programming environments on the CM-2, a context flag was set in each processor to determine whether or not that processor would store the result of an operation in its memory. In the RTS a mask array is used for this purpose, allowing far more flexible use of context.

### 0.4.6 CM-5 Packed Mask Arrays

The CM-5/VU version of the RTS supports both packed and unpacked forms of mask arrays. In both cases, the mask array is an array of unsigned4 values. The subgrid increment of the array determines whether the array is packed or not.

If the subgrid increment is a non-negative integer, the array is in unpacked form. That is, it is an array of unsigned word-length integers, in which a non-zero word value indicates an active element, and a zero value indicates an inactive element.

If the array increment is the constant CMCOM_PACKED_MASK, the array is in packed format. The mask values are stored as packed bits, 32 bits per word, in ascending order from LSB to MSB. The LSB corresponds to the lowest-numbered array element in the subgrid or subgrid bank. A “1” bit indicates an active element, and a “0” an inactive element. Only active array elements participate in a masked operation.

For CMRT arrays, the subgrid increment is taken from the array descriptor. In CMCOM functions, the mask_inc argument specifies the mask’s subgrid increment — specifying the constant CMCOM_PACKED_MASK as the mask_inc argument indicates that the mask is packed.
0.4.7 Send and Grid Addresses

Among other things, the geometry of an array assigns an address to each element in the array. These element addresses are used in a number of RTS functions to specify the destinations of CM communication operations and to define the order in which array elements are copied or combined.

An array element can be referred to by either of two kinds of address:

- **send address** — A unique value for each element, representing its location within the CM hardware. (See the discussion of send address data types in Section 0.7 below.)

- **grid address** — A series of integers, representing an element’s coordinates within a given geometry. (This is analogous to the array indices used by high-level languages.)

**Terminology Note:** The grid address of an element is also known as its set of grid coordinates. Thus, a single CMRT_send_address array corresponds to a number of CMRT_grid_coordinate arrays, one coordinate array for each array dimension.

An array element’s grid address can be different for different geometries, but its send address is geometry-independent. Thus, send addresses can be used to “translate” grid coordinates from one geometry to another. The RTS includes a number of functions that are used to manipulate send and grid addresses in this fashion.

0.4.8 Host and Node Functions

In the current version of the RTS, function execution takes place in one or more parts, depending on the software layer.

At the CMRT level, all functions execute on the partition manager, making calls to the CMCOM and CMIP layers as needed.

At the CMCOM and CMIP level, there is a distinction between code that executes on the partition manager and code that executes on the CM nodes. Most CMCOM and CMIP functions are implemented in pairs:

- A **host function** that executes on the partition manager and controls the sequencing of CM operations.

- A **node function** that executes on the PEs and handles the actual computation involved.

When a CMCOM or CMIP function is called, the host function executes first. It makes a call to the corresponding node function, providing whatever arguments the node function requires. The host function then either waits for the node function to complete (if a returned value is required), or simply returns and lets the node function complete its execution at its own pace.
The host functions of the CMCOM and CMIP layer are always coded in C. The node functions are coded differently for each CM hardware version:

- CM-2: Node functions are coded in CMIS or PEAC
- CM-5: Node functions are coded in C/CMNA
- CM-5/VU: Node functions are coded in C/CMNA (or C/DPEAC for the CM-5/VU)

0.4.9 Virtual, On-Chip, and Off-Chip Functions

In addition, many CMCOM functions such as scans, spreads, and global reductions, are performed by a trio of functions:

- An on-chip function handles the part of the operation that involves the subgrid stored on each PE. (For example, the on-chip part of a global operator combines all the values of each subgrid.)
- A physical function handles the part of the operation that involves transferring data between the PEs, or combining data taken from each PE. (For example, the physical part of a global operator combines the on-chip value computed on each PE into a single global value.)
- A virtual function that makes calls to both the on-chip and physical functions, and combines the results appropriately. This “virtual” function hides the distinction between the on-chip and physical functions, and is the normal function-call entrypoint.

<table>
<thead>
<tr>
<th>CM Function Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host Function</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Node Function</td>
</tr>
</tbody>
</table>

This distinction is made (and interfaces for the on-chip and physical routines are provided) so that a high-level language compiler (Fortran, C*, etc.) can generate calls to the on-chip and physical functions directly for special purposes.

Note: The on-chip routines exist as separate functions only in the CM-2 version of the RTS.
0.4.10 CMRTS "On A Node"

Currently, the CMRTS executes in a global data-parallel manner, with function execution divided between the partition manager and the nodes — the PM executes the main program, and the nodes handle any parallel data manipulations in tandem (that is, all array operations are performed globally, across the entire machine.)

A future version of the RTS will allow code to execute in a "local" manner (sometimes called "CMRTS On A Node"). In this execution model, you can execute a program in multiple copies directly on the CM processing nodes, in effect treating each node as a single CM supercomputer. The nodes communicate with each other via message-passing operations (such as those in the CMMD library), and the partition manager’s role is reduced to that of an I/O server or a task monitor for the nodes.

On a CM-5 with vector units, for example, each node acts as a 4 PE "partition", with the processing node acting as partition manager and executing the main program, while the VUs act as "nodes" and perform parallel operations on the data stored in the node’s memory.

CMRTS "On A Node" will be used to implement the "on a node" and "global/local" versions of high-level parallel programming languages, such as CM Fortran and C*.
0.5 RTS Programming Issues

0.5.1 Initializing the RTS

Before the RTS software can be used, its internal data structures must be initialized. The CMRT function \texttt{CMRT}_init performs this initialization step, and must be called before any other RTS operation. (For more information, see the description of \texttt{CMRT}_init later in this document.)

0.5.2 General Function Conventions

The RTS has a few general array argument conventions that can be assumed to hold for all cases unless otherwise noted:

- All array axes are zero-based.
- All CM array arguments to a function must have the same geometry (and typically the same element type).
- Array arguments may not overlap in CM memory, except identically. (See below.)
- Array elements must be properly aligned to their element size. (See below.)

Overlapping: Array arguments can overlap either exactly or partially. Exact overlapping occurs only when the two arrays have the same starting location in memory, the same total length, and the same element size (for example, when the destination array argument to a function is the same as one of its source array arguments and the two have the same array increment). Exact overlapping of arguments is allowed only where explicitly stated.

Partial overlap of array arguments is never allowed. Partial overlapping occurs when two arrays share some of their elements, \textit{but not all} elements in both arrays. For example, partial overlapping occurs when the starting memory location for one array is within the allocated space of another array. (This can only happen if the array addresses have been manipulated by CMCOM/CMIP operators. If only CMRT array allocation and deallocation operators are used, partial overlap cannot occur.)

\textbf{C* Implementation Note:} Interleaved arrays do not violate the partial overlapping rule. Such arrays occur when, for example, the arrays in question are different slots in an array of structures. Elements of these arrays are interleaved in memory, so that operations that read or write either array affect only the elements that belong to that array (by appropriate striding through memory, etc.), and do not affect elements of the other array. In this case, even though the starting position of one array is "within" the memory region occupied by the other, no overlap actually occurs.
Alignment: All array elements moved or computed on must be strided by a multiple of 4 bytes, and, on the CM5, must be aligned to the element size, (i.e. double-words must be double-aligned). However, this alignment requirement does not apply to functions that take an “element-size” argument — this argument overrides the actual element-size of the array.

0.5.3 Function Formats and Conventions

The section below is an example of the format used in this document to describe functions in the CMRT, CMCOM, and CMIP layers.

Note: A single description may refer to a group of functions. In this case, square brackets are used to denote variations in function names, and symbolic names are used to represent large groups of options, such as type names. Optional portions of names are indicated by curly braces.

For example, the names of the six broadcast functions for different data types are specified by:

```
broadcast_{[\em type]}
type = int4, log4, real4, real8, cmpx8, cmpx16
```

And the functions `CMRT_cshift` and `CMRT_cshift_vector` are specified by:

```
CMRT_cshift{\_vector}
```

0.5.4 CMLIB_function_name

```
returned_type CMLIB_function_name( arg1, arg2, etc.);
data_type_1 arg1;
data_type_2 arg2;
etc.
```

Description: Description of function, its purpose, and its effects.

Restrictions: Restrictions on use or availability of functions. (For example, whether it is unavailable for a particular CM hardware version.)

Performance Hints: Optional — provides useful or helpful hints on getting the best performance from the operation.
0.5.5 Using the CMRTS

This section describes a number of basic things you might want to do with the RTS, and lists the functions you would need to use to do them. See the descriptions of these functions in later sections for argument lists and important usage information. See Section 4 in particular for detailed information about defining and using geometry objects.

To allocate a CMRT array, do the following:

- Define the array's geometry, using one of the following functions:
  - CMRT_intern_array_geometry — defines a "generic" geometry.
  - CMRT_intern_detailed_array_geometry — allows detailed control over array shape and axis layout.
  - CMRT_intern_specific_array_geometry — similar, uses masks to determine axis layout with increased flexibility.

- Using the geometry, allocate the array with one of:
  - CMRT_allocate_heap_array — defines a heap memory array
  - CMRT_allocate_stack_array — defines a stack memory array

  Note: the array's data type is specified by a CMRT_element_type argument.

- When you no longer need the array, deallocate it with one of:
  - CMRT_deallocate_heap_array
  - CMRT_deallocate_stack_array

To store values into an array, use one of these functions:

- CMRT广播[type] — stores the same value in all elements.
- CMRT写入array_element[type] — modifies one element.
- CMRT写入array — copies in an entire array of values from PM.
- CMRT随机 — stores random values into array elements.

To read values from an array, you can use the corresponding functions:

- CMRT读取array_element[type] — gets value of one element.
- CMRT读取array — copies CM array to array on PM.
Once an array is initialized, you can use RTS parallel data operations to modify its values in parallel. Examples of these operations are:

- **CMRT_send** — performs an arbitrary shuffling of values.
- **CMRT_news** — performs a grid-wise “shift” of values.
- **CMRT_scan_[combiner]** — does a parallel-prefix operation.
- **CMRT_sort** — sorts the values of an array.

If you need to get at the actual CM array, machine geometry, or other components of a CMRT array, there are a number of array descriptor accessor functions that you can use. For example:

- **CMRT_desc_get_cm_location** — address of CM array.
- **CMRT_desc_get_geometry** — machine geometry of array.
- **CMRT_desc_get_subgrid_size** — number of array elements per PE.
- **CMRT_desc_get_element_size** — number of bytes per element.

Although it is not recommended, you can create a machine array on the CM stack using CMCOM operators as follows:

- Define the array’s geometry, using either of the functions CMCOM_intern_geometry or CMCOM_intern_specific_geometry.
- Determine the amount of memory needed on each PE for the array (typically, this will be the product of the number of bytes per element times the number of elements per PE — this “subgrid size” will in turn be determined by the geometry you’ve defined.)
- Use CMCOM_get_stack_pointer to get the current stack pointer.
- Allocate the required stack memory, using:
  - CMCOM_allocate_stack_space to allocate a number of bytes
  - CMCOM_allocate_stack_words to allocate a number of words
- When you no longer need the array, deallocate it by setting the stack pointer back to its original value with CMCOM_set_stack_pointer.
Once you have obtained the location, subgrid size, and element size of an array (via the accessor functions described above, or by defining the array yourself using CMCOM operators), you can call CMIP operators to perform “in-processor” operations on the array. Some examples are:

- **CMIP_move** — copies contents of one array to another.
- **CMIP_add, CMIP_subtract, CMIP_product** — arithmetic operations.
- **CMIP_[type].eq, CMIP_[type].lt** — arithmetic comparisons.

### 0.6 The CMU Utility Functions

The following utility functions are provided by the RTS:

```c
void CMU_set_error_handler(function)
    void (*function)();
```

**CMU_set_error_handler** specifies an error function that is to be called whenever an error is signalled. (The default error function simply prints the error message and aborts.)

```c
void CMU_set_warning_handler(function)
    void (*function)();
```

**CMU_set_warning_handler** specifies a warning function that is to be called whenever a warning is signalled. (The default warning function simply prints the warning message.)

```c
void CMU_error(format, args...)
    char *format;
```

**CMU_error** can be called by user code to signal an error. It takes a variable number of arguments (similar to printf), and calls the error handler function. (By default, this simply prints the error and aborts the program.)

```c
void CMU_warning(format, args...)
    char *format;
```

**CMU_warning** can be called by user code to display a warning. It takes a variable number of arguments (similar to printf), and calls the warning handler function. (By default, this simply prints the warning.)
0.7 RTS Data Types

A number of specialized C data types are defined and used in the RTS.

Of these, a small subset can be used either on the partition manager, or as the element-type of a CM array stored on the PEs:

- **unsigned4** — an unsigned single-precision (32-bit) integer
- **int4** — a signed single-precision (32-bit) integer
- **unsigned8** — an unsigned double-precision (64-bit) integer
- **int8** — a signed double-precision (64-bit) integer

**Implementation Note:** The **unsigned8** and **int8** types do not currently exist, but will be needed to support larger axis coordinates than are currently supported on the CM-5. For example, to access a word from a one-dimensional array on a CM with 64K processors (each with 4 Mbytes of memory), sixteen bits are required to address a specific CM processor, and twenty-two bits are needed to address a memory word, making 38 bits total.

Until the long integer types are supported by a C compiler, the **int8** type is **typedefed** to **int4** and the **unsigned8** type is **typedefed** to **unsigned4**. This limits the size of a single axis of an array to the ranges of those types. Furthermore, the total number of elements in the array is limited to be smaller than $(2^{31}) - 1$.

- **duint** — a structure type representing an unsigned8 value
- **dint** — a structure type representing an int8 value

**Implementation Note:** The **dint** and **duint** are intended as interim data types, to be used until **unsigned8** and **int8** can be implemented. Currently, only the CM-5/VU version of the RTS has operations that support them.

- **real4** — a single-precision (32-bit) floating-point value
- **real8** — a double-precision (64-bit) floating-point value
- **cmpx8** — a single-precision (64-bit) complex value
- **cmpx16** — a double-precision (128-bit) complex value
- **log4** — logical value

**Implementation Note:** The **log4** type is actually **typedefed** to **int4**, with valid values of **LOG_TRUE** (all 1s) or **LOG_FALSE** (all 0s). These values differ from the usual **TRUE** (1) and **FALSE** (0).
• CMCOM_send_address – send address value

Implementation Note: The way send addresses are stored depends on the CM hardware version:

– CM-2: Send addresses are stored as a 32-bit integer — 12 bits for the physical component, and 20 bits for the subgrid component.

– CM-5: Send addresses are stored as two 32-bit integers, one for the physical component, one for the subgrid component.

Implementation Note: The element type CMRT_grid_coordinate is currently declared to be of type int8. To define a “grid coordinate” array, you can declare it to be of element type int8. Currently CMRT_grid Coordinate is a 4-byte type, but when int8 is expanded to an 8-byte type in the future, use of other 4-byte element types (CMRT_u_integer for example) may not be portable to future versions of the RTS. Also, if the grid coordinate argument passed to a RTS function is of an element type larger than 4 bytes, only the first 4 bytes of each element are used. If/when the type int8 is expanded to 8 bytes, the first 8 bytes of each element will be used.

The following data types are used only on the partition manger:

• CMRT_desc_t — a CMRT array descriptor

• CMRT_cm_location_t — a CM memory location (used at CMRT level)

• CMCOM_cm_address_t — a CM memory address (used at CMCOM/CMIP level)

• CMRT_element_type_t — an array element type specifier, one of:

   CMRT_logical — Logical (LOG_TRUE, LOG_FALSE) values, of type log4.
   CMRT_s_integer — Signed single-precision (32-bit) integers, type int4.
   CMRT_double_u_integer — Unsigned double-precision (64-bit) integers, type unsigned8.
   CMRT_double_s_integer — Signed double-precision (64-bit) integers, of type int8.
   CMRT_double_float — Double-precision (64-bit) floating-point numbers, of type real8.
   CMRT_complex — Single-precision (64-bit) complex numbers, of type cmpx8.
   CMRT_double_complex — Double-precision (128-bit) complex numbers, of type cmpx16.
   CMRT_character — 8-bit characters, of type char.
   CMRT_send_address — PE send addresses, of type CMCOM_send_address_t.
   CMRT_grid_coordinate — PE grid coordinates, of type int8.
• CMRT\_array\_geometry\_t — CMRT array geometry data structure

• CMCOM\_machine\_geometry\_t — CMCOM machine geometry data structure

• CMRT\_home\_t — an array "home" value, one of:
  CMRT\_home\_fe — array typically stored on the PM
  CMRT\_home\_cm — array typically stored on the nodes
  CMRT\_home\_fe\_only — array stored only on the PM
  CMRT\_home\_cm\_only — array stored only on the nodes

• CMCOM\_order\_t — physical send address order, either of:
  CMCOM\_send\_order — send-address ordering
  CMCOM\_news\_order — grid-address ordering

  Implementation Note: The CM-5 uses only send-address ordering, and therefore arguments with the CMCOM\_order\_t type are ignored by the CM-5 versions of RTS functions. For a description of the distinction between send and grid address ordering, see the Geometries section of this document.

The following data types are used exclusively by the scanning operations in the CMRT and CMCOM layers:

• CMCOM\_direction\_t — scan direction, one of:
  - CMCOM\_upward
  - CMCOM\_downward

• CMCOM\_segment\_mode\_t — scan segment mode, one of:
  - CMCOM\_none — no segmentation
  - CMCOM\_segment\_bit — direction-independent segments
  - CMCOM\_start\_bit — direction-dependent segments

• CMCOM\_inclusion\_t — scan inclusion mode, one of:
  - CMCOM\_inclusive — include element in scan result
  - CMCOM\_exclusive — exclude element from scan result
1 The CMRT Interface

This section lists and describes the functions available in the CMRT software layer. The functions are listed by category, in the following order:

- CMRT initialization
- CMRT environment functions
- CMRT array allocation/deallocation
- Geometry definition functions
- CMRT array descriptors
- CM memory allocation/deallocation
- Send and grid addresses
- Broadcast functions
- Array read/write functions
- General communications functions (sends and gets)
- Cross-geometry move
- NEWS (axis shift) communication
- Scans/reductions/spreads
- Array reduction (global) functions
- Enumeration, ranking, and sorting
- Array manipulation utilities
- Vector and matrix multiply functions
- Random number generation functions
- CM-2 low-level interface functions
- Indirect Addressing Functions
- Miscellaneous CMRT functions
1.1 Common CMRT Function Arguments

There are a number of "standard" arguments that most CMRT functions have, such as the source, destination, and mask array arguments. These arguments can be identified by their names:

- **source** (source1, source2, etc.) — The source array(s) from which values are read.
- **dest** (dest_array, etc.) — The destination array into which the result is written.
- **mask** — The user mask argument, as described in the Concepts section above.
- **geometry** — An array geometry object, as returned by one of these functions:
  - CMRT_intern_array_geometry
  - CMRT_intern_detailed_array_geometry
  - CMRT_intern_specific_array_geometry
  (Note that a geometry argument is only required when geometry information cannot be derived from the source or destination array descriptors.)
- **axis** — An integer (zero-based), selecting an axis of a geometry (typically the geometry specified by the geometry argument); axis must be an integer in the range 0 to one less than the rank of the geometry.
- **coordinate** — An integer, giving an absolute (zero-based) coordinate along an axis (in other words, an integer from 0 up to 1 less than the length of the axis).

These "standard" arguments are not described separately for each function; rather, only arguments unique to each function or unique meanings of the above arguments are described.

1.2 CMRT Initialization

Before the RTS software can be used, its internal data structures must be initialized. This section describes functions that are used to initialize the CMRT software layer and any attached CM, and to obtain information about the current software and hardware.

1.2.1 CMRT_init, CMRT_initialized

```c
void CMRT_init()
int4 CMRT_initialized()
```

**Description:** CMRT_init initializes the internal tables and data structures of the CMRT software layer. It also initializes the attached CM hardware. (The CM-2 version attempts to attach to CM hardware, if necessary.)

CMRT_initialized returns TRUE if the CMRT layer is initialized, FALSE otherwise.
1.2.2 CMRT_set_log2_subgrid_quantum

    void CMRT_set_log2_subgrid_quantum(nbits)
    int nbits;

    int CMRT_get_log2_subgrid_quantum()

Description: These functions regulate the default minimum subgrid size for arrays defined by the CMRTS. This is expressed in terms of the minimum number of bits needed to represent the offset of an array element in the subgrid. Thus, if the minimum subgrid quantum is set to \( n \), then array subgrids will have at least \( 2^n \) elements.

**CMRT_set_log2_subgrid_quantum** sets the default subgrid quantum to a specified number of bits. **CMRT_get_log2_subgrid_quantum** returns the current setting.

**Note:** This setting only controls the default minimum subgrid size — regardless of the current subgrid quantum setting, you can still use geometry operators to define arrays with subgrids that are a multiple of this value, and changing the quantum value does not affect the structure of any arrays you have already defined.

**Implementation Note:** The subgrid quantum setting exists primarily to provide easier portability between different versions of the RTS, where different values for the default minimum subgrid size are required.

Currently, the initial setting of the subgrid quantum depends on the RTS version in use:

- **CM-2** — The initial value is 2 (that is, a minimum of 4 elements per subgrid)
- **CM-5/SPARC** — The initial value is 0 (a minimum of 1 element per subgrid)
- **CM-5/VU** — The initial value is 3 (a minimum of 8 elements per subgrid)

1.2.3 CMRT_pe_[length/limit]

    int CMRT_pe_length()
    int CMRT_pe_limit()

Description: These functions both return information about the number of available processors: **CMRT_pe_limit** returns the number of nodes in the partition. **CMRT_pe_length** returns the number of bits required to represent the limit (in other words, \( \log_2(\text{CMRT_pe_limit}) \)).
1.2.4 CMRT_version

char *CMRT_version()

Description: This function returns a string describing the CMRT software version.

1.2.5 CMRT_sizeof

int4 CMRT_sizeof (type)
   CMRT_element_type_t type;

Description: This function returns the size of the specified type in bytes. Currently, this function returns the following:

For the array types:

- CMRT_logical, CMRT_u_integer, CMRT_s_integer, CMRT_float
- CMRT_send_address, CMRT_grid_coordinate (on the CM-2)

CMRT_sizeof returns 4.

For the array types:

- CMRT_send_address, CMRT_grid_coordinate (on the CM-5)
- CMRT_double_float, CMRT_complex

CMRT_sizeof returns 8.

For the CMRT_double_complex array type, CMRT_sizeof returns 16.

For the CMRT_character array type, CMRT_sizeof returns 1.

And if the type argument is unknown, CMRT_sizeof returns -1.
1.3 CMRT Environment Functions

The functions described in this section enable and disable CMRT software features.

1.3.1 CMRT_{enable/disable}_memory_fill, CMRT_memory_fill_enabled

\begin{verbatim}
void CMRT_{enable/disable}_memory_fill()
int4 CMRT_memory_fill_enabled()
\end{verbatim}

Description: These three functions control the automatic clearing of newly allocated arrays.
CMRT_{enable}_memory_fill turns array clearing on.
CMRT_{disable}_memory_fill turns array clearing off.
CMRT_memory_fill_enabled returns TRUE if clearing is enabled, and FALSE otherwise.

1.3.2 CMRT_{enable/disable}_send_verification

\begin{verbatim}
void CMRT_enable_send_verification()
void CMRT_disable_send_verification()
\end{verbatim}

Description: These two functions control the verification of CMRT_send and CMRT_get operations by simulation on the partition manager. (Needless to say, this is very slow, and is recommended only for debugging purposes.)
CMRT_enable_send_verification turns on send simulation.
CMRT_disable_send_verification turns off send simulation.
1.4 CM Array Allocation/Deallocation

The functions described in this section allocate and deallocate CM arrays in the heap or the stack.

Usage Note: Prior to allocating a CM array, you must define a geometry object using one of the geometry-definition functions. This geometry object, along with the element type, determines the size and shape of the newly allocated array.

1.4.1 CMRT_allocate_[heap/stack]_array

```c
CMRT_desc_t CMRT_allocate_[heap/stack]_array(geometry, element_type)
CMRT_array_geometry_t geometry;
CMRT_element_type_t element_type;
```

Description: These two functions allocate and return a CM array with the specified array geometry and element type, in either the heap or the stack.

Arguments:

- `geometry` — CMRT geometry that defines shape of array to be allocated.
- `element_type` — Data type of the elements of the array.

1.4.2 CMRT_deallocate_[heap/stack]_array

```c
void CMRT_deallocate_[heap/stack]_array(array)
CMRT_desc_t array;
```

Description: These functions deallocate the supplied CM array, which must have previously been allocated by either of the functions

CMRT_allocate_heap_array or
CMRT_allocate_stack_array.

Note: Deallocating a stack array implicitly deallocates all stack arrays that were allocated after it. All such deallocated arrays become invalid, and should not be used. (The CM memory space can be reallocated, causing errors when a new array overwrites one that has been invalidated.)
1.5 Geometry Definition Functions

The functions described in this section are used to define geometry objects, which can be used to specify the shape and size of newly allocated arrays. For more information about geometry objects, and about the functions in this section, see Section 4.

1.5.1 CMRT_intern{[detailed,specific]}_array_geometry

CMRT_array_geometry_t CMRT_intern_array_geometry
(rank, lower_bounds, upper_bounds, orders, weights, is_serial,
highest_axis_varies_fastest)

CMRT_array_geometry_t CMRT_intern_detailed_array_geometry
(rank, lower_bounds, upper_bounds, orders, off_chip_positions,
off_chip_lengths, subgrid_lengths, is_serial,
highest_axis_varies_fastest)

CMRT_array_geometry_t CMRT_intern_specific_array_geometry
(rank, lower_bounds, upper_bounds, orders, axis_order,
physical_masks, subgrid_extents)

int4 rank,*weights,*off_chip_positions,*off_chip_lengths,*subgrid_lengths;
int8 *lower_bounds, *upper_bounds;
CMCOM_order_t *orders;
log4 *is_serial, highest_axis_varies_fastest;
int4 *axis_order, *physical_masks, *subgrid_extents;

Description: These three functions each define and return a CM array geometry object.

CMRT_intern_array_geometry defines a geometry with an automatically-calculated layout that is efficient for a given pattern of usage (the pattern specified by the axis weight values passed in the weights argument).

CMRT_intern_detailed_array_geometry defines a geometry with a specific layout (determined by off-chip position/length arguments). CMRT_intern_specific_array_geometry is similar, but allows the use of masks to specify the axis layout.

Implementation Note: Newly created geometry objects are internally stored ("interned") so that they can be re-used. When the arguments to the geometry-creating functions below specify a geometry that has previously been defined, an existing geometry object is returned, rather than a new geometry object with the same properties. This reduces overhead and redundancy by allowing several arrays to share the same geometry object.
1.5.2 CMRT.geometry_get_garbage_mask

CMRT_cm_location_t CMRT_geometry_get_garbage_mask(array_geometry)
CMRT_array_geometry_t array_geometry;

Description: This function takes a CM array geometry object and returns the CM memory location of the geometry's garbage mask.

1.5.3 CMRT.print_array_geometry

void CMRT_print_array_geometry(geometry)
CMRT_array_geometry_t geometry;

Description: This function prints to the standard output a nicely-formatted description of the supplied CMRT array geometry.

1.6 CM Array Descriptors

The functions described in this section are used to allocate and deallocate CM array descriptors, and access the information fields of descriptor objects.

Usage Note: It is not necessary to allocate an array descriptor prior to allocating an array. The CMRT_allocate_[heap/stack]_array functions automatically allocate and initialize an appropriate descriptor. The only reason for explicitly allocating an array descriptor is to construct unusual forms of arrays, such as two arrays that share the same memory region.

1.6.1 CMRT.[allocate/deallocate].descriptor

CMRT_desc_t CMRT_allocate_descriptor()

void CMRT_deallocate_descriptor(descriptor)
CMRT_desc_t descriptor;

Description: CMRT.allocate_descriptor allocates and returns a CM array descriptor. CMRT.deallocate_descriptor deallocates the supplied descriptor, which must have been allocated by CMRT.allocate_descriptor.
1.6.2 CMRT_desc_get_\[array_field, axis_field\]

\[\text{type} \] CMRT_desc_get_\[array_field\](descriptor)  
CMRT_desc_t descriptor;

\[\text{type} \] CMRT_desc_get_\[axis_field\](descriptor, axis)  
CMRT_desc_t descriptor;  
int4 axis;

type = (defined in Description section below)

array_field = cm_location, element_size, element_type, geometry, home,  
increment, is_modified, number_of_elements, rank, subgrid_size
axis_field = is_serial, lower_bound, offchip_length, offchip_mask,  
offchip_position, order, subgrid_axis_increment,  
subgrid_dimension, subgrid_orthogonal_length,  
subgrid_outer_count, subgrid_outer_increment, upper_bound

Description: These functions retrieve and return the value of a field from the supplied CM array descriptor. For fields that have a different value for each array axis, a second axis argument is used to select the axis value that is returned.

The accessible fields (and their corresponding types) are:

Array Fields — fields that apply to the entire array:

- \text{cm_location} (CMRT_cm_location_t) — The CM memory location of the array data.
- \text{element_size} (int4) — The number of words of memory per element.
- \text{element_type} (CMRT_element_type_t) — The data type of the elements of the array.
- \text{geometry} (CMRT_array_geometry_t) — The array’s geometry descriptor.
- \text{home} (CMRT_home_t) — Indicates where the array is stored (in partition manager or node memory).
- \text{increment} (int4) — The increment (number of memory words) between successive array elements in memory.
- \text{is_modified} (log4) — LOG_TRUE if the array has been modified by higher-level (CMF) routines, LOG_FALSE otherwise.
- \text{number_of_elements} (int8) — Total number of array elements.
- \text{rank} (int4) — The array rank (number of dimensions).
- \text{subgrid_size} (int4) — The number of elements in the per-node subgrid.
Axis Fields — fields that apply to each axis individually:

- **is_serial** (log4) — Whether the axis is stored in serial order.
- **lower_bound** (int8) — Lower bound of axis length of array.
- **offchip_length** (int4) — The length of the bit field for the axis in the offchip mask.
- **offchip_mask** (unsigned4) — The offchip bit mask for the specified axis of the array. The mask contains one bit for each dimension of the off-chip communications network, and indicates with 1 bits which dimensions (if any) are used for the specified axis.
- **offchip_position** (int4) — Starting position of the bit field for the axis in the off-chip mask.
- **order** (CMCOM_order_t) — Returns the order (send or NEWS) in which the coordinates of the specified axis are arranged. Note: The CM-2 uses both types of axis ordering. The CM-5 uses only send ordering.
- **subgrid_axis_increment** (log4) — Byte increment between elements along subgrid axis.
- **subgrid_dimension** (int4) — The size of the specified axis of the per-node subgrid.
- **subgrid_orthogonal_length** (log4) — The product of the subgrid lengths of all the other axes (essentially, the size of the multidimensional “slices” along this subgrid axis).
- **subgrid_outer_count** (log4) — Product of the subgrid lengths of all axes with larger subgrid axis increments than this axis.
- **subgrid_outer_increment** (log4) — Product of the subgrid axis increment and subgrid length for this axis.
- **upper_bound** (int8) — Upper bound of axis length of array.

**Note:** Prior to Version 7.2 of the CM-5 run-time system and in or before Version 6.1.2 of the CM-2 slicewise run-time system, the **is_serial** operator was named **CMRT_desc_axis_is_serial**.

**Restrictions:** The **desc_get_offchip_position** function cannot be called if the offchip bits of the axis are not contiguous. (In other words, if there is no single starting position for the offchip bits of the axis.)
1.6.3 CMRT_desc_set_[field]

    void CMRT_desc_set_[field](descriptor, location)
    field = cm_location, element_type, increment, geometry,
           is_modified, element_size

CMRT_desc_t descriptor;
CMRT_cm_location_t location;
CMRT_element_type_t type;
CMRT_array_geometry_t geometry;
log4 is_modified;
int4 element_size;

Description: These functions each set the value of a field of the supplied CM array descriptor. The fields are as described under CMRT_desc_get_[field], with the exception of the array_increment and subgrid_axis_order functions, which have no corresponding desc_set form.

1.6.4 CMRT_desc_get_array_increment

    int4 CMRT_desc_get_array_increment(descriptor)
CMRT_desc_t descriptor;

Description: This function returns the value of the increment field, if it is non-zero, and returns the value of the element_size field otherwise.

This is useful when you need to get an increment of some kind for an array, and want to automatically disallow the zero-increment case.

1.6.5 CMRT_desc_get_subgrid_axis_order

    int4 CMRT_desc_get_subgrid_axis_order(descriptor, axis_order)
CMRT_desc_t descriptor;
int4 *axis_order;

Description: This function determines the subgrid axis sequence implied by the information in the given array descriptor, and stores this sequence as a series of integers in the user-supplied axis_order array.

See Section 4.1.8 for more information about subgrid axis sequences.
1.7 CM Memory Allocation/Deallocation

The functions described in this section are used to allocate and deallocate regions of CM memory.

Usage Note: You do not need to explicitly allocate CM memory to allocate a CM array. The 
\texttt{CMRT\_allocate\_heap/stack\_array} functions automatically allocate an appropriate region of 
CM memory. The only reason for explicitly allocating CM memory is to reserve memory for unusual 
operations (such as constructing arrays with arbitrary shapes or sizes).

1.7.1 \texttt{CMRT\_allocate\_\{physical\[_\{heap/stack\]\_field

\begin{verbatim}
CMRT_cm_location_t CMRT_allocate_\{stack,heap\}_field(geometry, num_bytes)
CMRT_array_geometry_t geometry;
int4 num_bytes;

CMRT_cm_location_t CMRTallocate\_physical\_\{stack,heap\}_field(num_bytes)
int4 num_bytes;
\end{verbatim}

Description: These functions allocate and return a field in CM memory, either heap or stack, 
with the specified length in bytes. The \texttt{physical} functions allocate memory strictly in terms of 
bytes. The \texttt{non-physical} functions determine the number of elements stored in each node from the 
geometry, and allocate a field with the specified number of bytes for each element.

1.7.2 \texttt{CMRT\_deallocate\_\{physical\[_\{heap/stack\]\_field\_through

\begin{verbatim}
void CMRT\_deallocate\_\{stack,heap\}\_field\_through(field, geometry, num_bytes)
CMRT_cm_location_t field;
CMRT_array_geometry_t geometry;
int4 num_bytes;

void CMRT\_deallocate\_physical\_\{stack,heap\}\_through(field, num_bytes)
CMRT_cm_location_t field;
int4 num_bytes;
\end{verbatim}

Description: These functions deallocate the supplied CM field, which must have previously been 
allocated by the corresponding field allocation function.

Note: Dealloacting a given stack field also deallocates all stack fields that were allocated after it.
1.7.3 CMRT_[get/set]_stack_pointer

    CMRT_cm_location_t CMRT_get_stack_pointer()

    int4 CMRT_set_stack_pointer(new_sp)
    CMRT_cm_location_t new_sp;

Description: These functions control the position of the CM stack pointer.

CMRT_get_stack_pointer gets the current stack position.

CMRT_set_stack_pointer sets the stack pointer to the specified position.

1.7.4 CMRT_available_memory

    int4 CMRT_available_memory()

Description: This function returns the amount of remaining CM memory (in bytes) available
for use in either the heap or stack.

Performance Hints: This count includes any free memory within the heap itself. However, this
free heap memory may not be usable because of fragmentation in the heap.

1.7.5 CMRT_absolute_address_from_cm_location

    CMCOM_cm_address_t CMRT_absolute_address_from_cm_location(field)
    CMRT_cm_location_t field;

Description: This function takes a CMRT memory location and converts it into a CMCOM
memory address, which can be passed to CMCOM-level operators.

Performance Hints: Currently, the two data types CMRT_cm_location_t and
CMCOM_cm_address_t are identical, so this operator is basically a cast. However, this corre-
spondence may change in the future, so code should not depend on it.
1.7.6 CMRT_add_offset_to_cm_location

CMRT_cm_location_t CMRT_add_offset_to_cm_location(field, num_bytes)
CMRT_cm_location_t field;
int4 num_bytes;

Description: This function adds an offset in bytes to a CMRT memory location, and returns the new location. This can be used, for example, to obtain an offset into a previously defined array.

1.8 Send and Grid Addresses

The functions described in this section are used to manipulate send and grid addresses.

Terminology Note: The "scalar" functions operate on send address objects (that is, values of the CMCOM_send_address_t type) stored in the memory of the partition manager.

1.8.1 CMRT_make_send_address

void CMRT_make_send_address(send_address)
CMRT_desc_t send_address;

Description: This function initializes a previously allocated CM array so that it contains send addresses. (In other words, it clears all elements of the array to zero.) The initialized array can then be used to construct send addresses (i.e., by CMRT_deposit_grid_coordinate_zero).

Restrictions: The array must have an element type of CMCOM_send_address_t, or any other type with a length of 8 bytes.

1.8.2 CMRT_my_send_address

void CMRT_my_send_address(dest, mask)
CMRT_desc_t dest;
CMRT_desc_t mask;

Description: The destination array dest, assumed to be created with element type CMCOM_send_address_t, is filled with "identity" send addresses. (That is, each element is filled with its own send address within the array.) The mask argument controls which elements of dest are modified.
1.8.3 **CMRT_my_grid_coordinate**

```c
void CMRT_my_grid_coordinate(dest, axis, mask)
CMRT_desc_t dest;
int4 axis;
CMRT_desc_t mask;
```

**Description:** Each element in the destination array `dest` is filled with the coordinate of that element along the specified axis `axis`, as defined by the specified `geometry`. The `dest` array is assumed to have an element type of `CMRT_grid_coordinate`.

**Note:** Element type `CMRT_grid_coordinate` is declared to be of type `int8`. Currently this is a 4-byte type, but may be expanded to an 8-byte type in the future. Use of other 4-byte element typed arrays (`CMRT_u_integer` for example) may not be portable to future versions of the RTS. If `dest` is of an element type larger that 4 bytes, the first 4 bytes of each element are overwritten with the coordinate value. If and when the type `int8` is expanded to 8 bytes, the first 8 bytes of each element will be written.

The `mask` argument controls which elements of `dest` are modified.

1.8.4 **CMRT_deposit_grid_coordinate{.zero}**

```c
void CMRT_deposit_grid_coordinate(grid_geometry, dest_send_address,
source_send_address, axis,
coordinate, mask)

void CMRT_deposit_grid_coordinate_zero(grid_geometry, dest_send_address,
source_send_address, axis,
coordinate, mask)
```

```c
CMRT_array_geometry_t grid_geometry;
CMRT_desc_t dest_send_address;
CMRT_desc_t source_send_address;
int4 axis;
CMRT_desc_t coordinate;
CMRT_desc_t mask;
```

**Description:** These functions “deposit” grid coordinate information into the send addresses in the `source_send_address` argument, and store the resulting modified send addresses in corresponding elements of the `dest_send_address` argument.
Each element in the destination array `dest_send_address` is modified to refer to position `coordinate` along the specified `axis`, as defined by the supplied `grid.geometry`. `coordinate` is an array of values of type `CMRT_grid.coordinate`, each between 0 and the extent of that axis minus 1.

The `dest_send_address` array must be of element type `CMRT_send_address`. The `mask` argument controls which elements of the `dest` are modified. The `grid.geometry` argument gives the geometry in which the supplied coordinate and axis arguments are to be interpreted.

The `_zero` version of this function assumes that the portion of the source send addresses corresponding to the specified axis is already zero. This allows it to execute substantially faster.

**NOTE:** If this assumption is violated, the results are unpredictable, and the function may produce invalid send addresses.

For example, assume that an element within the `source_send_address` array contains the send address corresponding to grid coordinates `[3,5,6]` in the specified `grid.geometry` (the coordinates are in incrementing axis order, i.e. 3 is the axis 0 coordinate, 5 the axis 1 coordinate and so forth). If the corresponding element of the `coordinate` array contains the value 20, and the `axis` argument is 1, then the send address corresponding to grid coordinates `[3,20,6]` will be stored into `dest_send_address`.

If the `cmrt` safety feature is enabled (by previously called `CMRT_enable_safety`) all attempts to deposit coordinates which beyond the extent of the specified axis will result in a warning message to the user terminal at run time.

### 1.8.5 CMRT_extract_grid_coordinate

```c
void CMRT_extract_grid_coordinate(grid_geometry, dest, send_address, axis, mask)
CMRT_array_geometry_t grid_geometry;
CMRT_desc_t dest, send_address, mask;
int4 axis;
```

**Description:** This function "extracts" grid coordinates from send addresses, by determining the coordinate along the specified `axis` that each `send_address` array element refers to, and storing it in the `dest` array.

The `mask` argument controls which elements of the `dest` are modified. The `grid.geometry` argument gives the geometry in which the supplied coordinate and axis arguments are to be interpreted.
1.8.6 CMRT_scalar_make_send_address

CMCOM_send_address_t CMRT_scalar_make_send_address()

Description: This function returns a value of type CMCOM_send_address_t, a send address, which addresses an array element having a grid coordinate of zero along every axis (in any geometry). Currently, such a send address is two words in length, both equal to zero. This send address object can be used to construct new send addresses (i.e., by using CMRT_scalar_deposit_grid_coordinate_zero).

1.8.7 CMRT_scalar_deposit_grid_coordinate{_zero}

void CMRT_scalar_deposit_grid_coordinate(grammar, send_address, axis, coordinate)
void CMRT_scalar_deposit_grid_coordinate_zero(grammar, send_address, axis, coordinate)
CMRT_array_geometry_t grammar;
CMCOM_send_address_t *send_address;
int4 axis;
int8 coordinate;

Description: These functions both "deposit" grid coordinate information into the send_address argument, which must be a pointer to a send-address object. The send_address argument is modified so that it refers to the specified grid coordinate along the given axis, relative to the supplied CM array geometry.

The _zero version of the function assumes that the portion of the source send address corresponding to the specified axis is already zero. This allows it to execute substantially faster.

NOTE: If this assumption is violated, the results are unpredictable, and the function may produce invalid send addresses.

The send_address argument, which is of type CMCOM_send_address_t, is modified to reference position coordinate along the specified axis, as defined by the geometry grammar. coordinate is an integer between 0 and the extent of that axis minus 1. The resulting send address is returned as the function value.

For example, assume send_address contains the send address corresponding to the grid coordinates [3,5,6] within the specified grid geometry (the coordinates are in incrementing axis order, i.e. 3 is the axis 0 coordinate, 5 is the axis 1 coordinate and so forth). If the coordinate argument is 20 and the axis argument is 1, the returned send address will correspond to grid coordinates [3,20,6].

Also: The CMRT safety feature does not check for a valid coordinate argument.
1.8.8 CMRT_scalar_extract_grid_coordinate

```c
int8 CMRT_scalar_extract_grid_coordinate(geometry, send_address, axis)
CMT_ARRAY_GEOMETRY_T geometry;
CMCOM_SEND_ADDRESS_T send_address;
int4 axis;
```

**Description:** The argument `send_address`, of type `CMCOM_SEND_ADDRESS_T`, is converted to the grid coordinates of the element it addresses (as per the `geometry`), and the coordinate along the specified `axis` is extracted and returned as the function value.

This function determines and returns the coordinate along the specified axis in the supplied CM array geometry to which the `send_address` argument refers.

This function “extracts” grid coordinates from the given `send address` argument, by determining the coordinate along the specified `axis` that the `send_address` refers to.

The `mask` argument controls which elements of the `dest` are modified. The `grid geometry` argument gives the geometry in which the supplied coordinate and axis arguments are to be interpreted.

1.8.9 CMRT_compress_send_address

```c
void CMRT_compress_send_address(dest, source, mask, array_geometry)
CMRT_DESC_T dest, source, mask;
CMT_ARRAY_GEOMETRY_T array_geometry;
```

**Description:** This function compresses 64-bit CM-5 send addresses (`source`) to 32-bit CM-2 send addresses (`dest`), if possible. The `mask` argument controls which `dest` elements are actually modified.

1.8.10 CMRT_expand_send_address

```c
void CMRT_expand_send_address(dest, source, mask, array_geometry)
CMRT_DESC_T dest, source, mask;
CMT_ARRAY_GEOMETRY_T array_geometry;
```

**Description:** This function expands 32-bit CM-2 send addresses (`source`) to 64-bit CM-5 send addresses (`dest`), if possible. The `mask` argument controls which `dest` elements are actually modified.
1.9 Broadcast Functions

The functions in this section are used to broadcast a single value from the partition manager to all nodes in a partition.

1.9.1 CMRT_broadcast_{type}

    void CMRT_broadcast_{type} (dest, mask, value)
    CMRT_desc_t dest;
    CMRT_desc_t mask;
    [type] value;

    type = log4, int4, int8, real4, real8, cmpx8, cmpx16, dint, duint

Description: This function "broadcasts" a single value to all elements of the dest CM array. The mask determines which values of dest are actually modified.

1.10 Array Read/Write Functions

The functions in this section are used either to read (or write) a single value of a CM array, or to copy an array to (or from) the CM.

1.10.1 CMRT_{read/write}_array_element_{type}

    [type] CMRT_read_array_element_{type} (source, send_address)
    CMRT_desc_t source;
    CMCOM_send_address_t send_address;

    void CMRT_write_array_element_{type} (dest, send_address, value)
    CMRT_desc_t dest;
    CMCOM_send_address_t send_address;
    [type] value;

    type = int4, unsigned4, log4, real4, real8, cmpx16, dint, duint

Description: These functions read or write a single value of the supplied CM array, at the location specified by the send_address argument.
Restriction: The dint and duint versions of this function are defined only in the CM-5/VU version of the RTS.

1.10.2 CMRT_[read/write]_array_element[into/from]_buffer

void CMRT_read_array_element_into_buffer(source, send_address, buffer, element_size)
void CMRT_write_array_element_from_buffer(dest, send_address, buffer, element_size)
CMRT_desc_t source;
CMRT_desc_t dest;
CMCOM_send_address_t send_address;
int *buffer;
int element_size;

Description: These functions read or write a single value in the supplied CM array, of the specified element_size, and use the supplied buffer (region of partition manager memory) to hold the value read or written. (This allows you to read or write arbitrary array values without having to explicitly refer to their data type.)

1.10.3 CMRT_[read/write]_array

void CMRT_read_array(dest, source)
int4 *dest;
CMRT_desc_t source;

void CMRT_write_array(dest, source)
CMRT_desc_t dest;
int4 *source;

Description: These functions copy arrays between the partition manager and the nodes. The arrays must be of the same total length. The geometry and element type are determined from the CM array descriptor (source or dest).
1.11 General Communications Functions (Sends and Gets)

The functions described in this section are used to perform multi-way point-to-point exchanges of data between processors.

1.11.1 CMRT_send, CMRT_send_[combiner]

   void CMRT_send(dest, send_address, source, mask)
   void CMRT_send_[combiner](dest, send_address, source, mask)
   combiner = logior, logxor, logand, add, product, max, min

   CMRT_desc_t dest;
   CMRT_desc_t send_address;
   CMRT_desc_t source;
   CMRT_desc_t mask;

Description: These functions send values from the source CM array to the dest CM array, according to the values of the send_address array. For each element of source, the corresponding value of the send_address array is the address of the dest element to which the source value is to be sent. The mask argument controls which elements of the source array are sent, and hence implicitly controls which dest array elements are modified.

When two or more values are sent to the same address, the values are combined according to the function called. CMRT_send retains a single arbitrary value, which "overwrites" all other values sent to the same address. The CMRT_send_[combiner] functions combine all received values for a single location by the indicated combiner function, and store the combined result in the dest array.

1.11.2 CMRT_get

   void CMRT_get(dest, source, send_address, mask)
   CMRT_desc_t dest;
   CMRT_desc_t source;
   CMRT_desc_t send_address;
   CMRT_desc_t mask;

Description: This functions "gets" values from the source CM array and stores them in the dest CM array, according to the values of the send_address array. For each dest element, the corresponding send_address array value is the address of the source element to "get." The mask argument controls which elements of the dest array are modified.
1.12 Cross-Geometry Move

The function described in this section is used to move array values between CM arrays having different geometries.

1.12.1 CMRT_cross_geometry_move

```c
void CMRT_cross_geometry_move(dest, source, axis_mapping,
    source_axis_coords, dest_axis_coords, mask)
```

- `dest`: CMRT_desc_t
- `source`: CMRT_desc_t
- `axis_mapping`: int4 *
- `source_axis_coords`: int8 *
- `dest_axis_coords`: int8 *
- `mask`: CMRT_desc_t

**Description:** This function copies values from the `source` to the `dest` array, which can have different geometries (including different ranks), but must have the same total size and element type. Additionally, the axes that are selected for copying (see `axis_mapping` below) must have the same length as the axes to which they are mapped.

**Usage Note:** Essentially, you can think of this function as a way to “rotate” a multidimensional array or subset of an array, and store it in a `dest` array with the same basic shape but possibly different dimensionality — a greater number of axes, for example. (You can not use this function to perform arbitrary rearrangements of data — use the CMRT_send and CMRT_get operations for that.)

Values are transferred between the two geometries according to the values of the `axis_mapping`, `source_axis_coords`, and `dest_axis_coords` arguments:

- `axis_mapping` defines the mapping of axes between `source` and `dest` geometries. If the nth element of `axis_mapping` is i, then the nth source array axis is mapped to the ith `dest` axis. A value of CMRT_NO_AXIS indicates an unmapped axis (one that doesn’t exist in the source or is not transferred to the dest)

- `source_axis_coords` and `dest_axis_coords` define, for unmapped axes, the source coordinate along that axis that is read from, and the `dest` coordinate that is written into. (Essentially, these two arguments define which “plane” of the source array is copied to the dest, and which plane of the `dest` array receives it.) To map a source axis to every coordinate along the `dest` axis, `dest_axis_coords` should be CMRT_NO_AXIS. For all axes that have a mapping in `axis_mapping`, the value of `source_axis_coords` should be CMRT_NO_AXIS.

The `mask` argument controls which elements of the source array are sent to the destination.
1.13 NEWS (Axis Shift) Communication

The functions described in this section are used to shift array data along one or more geometry axes. The main features of each of them are:

- **CMRT_news** — Non-wrapping shift, using a mask array to select elements.
- **CMRT_eoshift{.vector}** — Non-wrapping shift, non-masked, with a specified axis extent.
- **CMRT_cshift{.vector}** — Wrapping shift, non-masked, with a specified axis extent.

### 1.13.1 CMRT_news

```c
void CMRT_news(dest, source, distance, axis, mask)
CMRT_desc_t dest;
CMRT_desc_t source;
int8 distance;
int4 axis;
CMRT_desc_t mask;
```

**Description:** This function shifts `source` array elements along the specified `axis` of the `source` geometry (without wrapping), and stores the shifted result in the `dest` array. `source` and `dest` must be in the same geometry, and may be identical, but not in any other way overlapping.

Each element of the `source` array is shifted the same `distance` along the specified `axis`. The `distance` argument is the relative distance along the `axis` from which each `dest` element receives a value. (That is, a positive `distance` shifts values downward toward the low end of the axis, and a negative `distance` shifts values upward toward the high end.) Elements that are shifted out of the array axis altogether are lost, and array elements that do not receive a shifted value are undefined. (No wrapping is performed.) Any element shifted an absolute distance greater than the extent of the axis is guaranteed to be discarded.

Diagramatically:

```
CMRT_news: DEST[...i,...] = SOURCE[...i+distance,...] if it exists
```

The extent (length) of the selected `axis` is determined from the `dest` array geometry, specifically by subtracting 1 from the difference between the lower and upper bounds of the specified axis. (For the other shift functions, the axis extent is set by an argument.)

The `mask` argument controls which elements of the `dest` array are modified. (That is, only `dest` elements selected by the `mask` receive a shifted value.) In all other positions, the previous value is retained.
1.13.2 CMRT_eoshift{.vector}

```c
void CMRT_eoshift{.vector}(dest, source, distance, axis, extent)
    CMRT_desc_t dest;
    CMRT_desc_t source;
    int8 distance;
    CMRT_desc_t distance;
    int4 axis;
    int8 extent;
```

**Description:** These functions shift *source* array elements along the specified *axis* of the *source* geometry (without wrapping), and store the shifted result in the *dest* array. *source* and *dest* must be in the same geometry, and may be identical, but not in any other way overlapping.

Each element of the *source* array is shifted the same *distance* along the specified *axis*. The *distance* argument is the relative distance along the *axis* from which each *dest* element receives a value. (That is, a positive *distance* shifts values downward toward the low end of the axis, and a negative *distance* shifts values upward toward the high end.)

Elements that are shifted out of the array axis altogether are lost, and array elements that do not receive a shifted value are undefined. (No wrapping is performed.) Any distance greater than the *extent* (or less than -*extent* for upward shifts) guarantees the value to be discarded.

Diagramatically:

```
CMRT_eoshift: DEST[...i,...] = SOURCE[...i+distance,...] if it exists
```

The *extent* argument specifies the “length” of the *source* axis. The actual length may be greater than this value — this argument is useful for shifting subsections of an array. For example, the caller can do partial rotations within the news axis by offsetting *source* and *dest* and providing an *extent* less that the full extent of the axis. *extent* can legally have a value of 0, meaning “do no shifting”.

For CMRT_eoshift, the *distance* argument is a constant, and each element of the *source* array is shifted the same distance along the specified *axis*.

For CMRT_eoshift_vector, the *distance* argument is a CM array, and each element of the *source* array can be shifted by a different distance along the specified *axis*. 
1.13.3 CMRT_cshift{_vector}

void CMRT_cshift{_vector}(dest, source, distance, axis, extent)
  CMRT_desc_t dest;
  CMRT_desc_t source;
  int8 distance;
  CMRT_desc_t distance;
  int4 axis;
  int8 extent;

Description: These functions shift source array elements along the specified axis of the source geometry (with wrapping), and store the shifted result in the dest array. source and dest must be in the same geometry, and may be identical, but not in any other way overlapping.

Each element of the source array is shifted the same distance along the specified axis. The distance argument is the relative distance along the axis from which each dest element receives a value. (That is, a positive distance shifts values downward toward the low end of the axis, and a negative distance shifts values upward toward the high end.)

Elements that are shifted out of the array axis at one end are wrapped around to the other end of the axis (as determined by the extent argument). In other words, if distance is greater in magnitude than extent, it is reduced to the modulo of itself and the extent (thus rotation of distance -10 along an axis with extent 3 is equivalent to a rotation of -1).

Diagramatically;

\[
\text{CMRT_cshift: } \text{DEST}[\ldots,i,\ldots] = \text{SOURCE}[\ldots,(i+\text{distance})\%\text{extent},\ldots]
\]

The extent argument specifies the "length" of the source axis. The actual length may be greater than this value – this argument is useful for shifting subsections of an array. For example, the caller can do partial rotations within the news axis by offsetting source and dest and providing an extent less that the full extent of the axis. extent can legally have a value of 0, meaning "do no shifting".

For CMRT_cshift, the distance argument is a constant, and each element of the source array is shifted the same distance along the specified axis.

For CMRT_cshift_vector, the distance argument is a CM array, and each element of the source array can be shifted by a different distance along the specified axis. If two or more elements are rotated such that they land on the same location in the dest array, it is undefined as to which of them is written there, with all others being discarded. Elements in dest to which no source elements are rotated will contain unspecified values.
1.14 Scans/Reductions/Spreads

The functions described in this section are used to perform scanning (parallel prefix), reduction, and spreading operations on CM arrays.

1.14.1 CMRT_scan_[combiner]

CMRT_scan_[combiner](dest, source, segment, axis, direction, inclusion, segment_mode, mask)
combiner = add, product, max, min, copy, logior, logxor, logand

CMRT_desc_t dest, source, segment, mask;
CMCOM_direction_t direction;
CMCOM_inclusion_t inclusion;
CMCOM_segment_mode_t segment_mode;
int4 axis;

Description: These functions perform a scan (parallel prefix) operation along the specified axis of the source array, and store the results in the dest array. The elements along the axis are cumulatively combined with the indicated combiner operation.

The mask argument controls which elements of the source are included in the scan, and similarly which elements of dest are actually modified. (In arithmetic scans, an array element that is excluded by the mask is treated as if it contains the identity value for the scan operation.)

The direction, inclusion mode, and segmentation of the scan are controlled by the other arguments:

• direction is either CMCOM_upward or CMCOM_downward, and controls whether the scan is performed in increasing or decreasing order of axis coordinates.

• inclusion is either CMCOM_exclusive or CMCOM_inclusive, and controls whether the scan result stored in each element of the dest array excludes or includes the corresponding element of the source array. Note: in CMCOM_exclusive scans, the scan result for the first element of each segment is undefined (unless otherwise noted below).

• segment_mode controls the segmentation of the scan, and is one of:
  - CMCOM_none (for no segments)
  - CMCOM_segment_bit (for unidirectional segments)
  - CMCOM_start_bit (for bidirectional segments)
• segment is an array of logical values that specifies the starting elements of segments, as follows:
  - If segment_mode is CMCOM_none, the segment argument is ignored.
  - If segment_mode is CMCOM_segment_bit, then each LOG_TRUE in the segment array defines the starting element of a segment that always runs in increasing axis coordinate order (in other words, the location of the segment is unaffected by the scan direction).
  - If segment_mode is CMCOM_start_bit, then each LOG_TRUE in the segment array defines the starting element of a segment that runs in the current scan direction. (Upwards for upward scans, downward for downward scans.)

Note: When the segment_mode is CMCOM_start_bit, the mask argument also controls which elements of the segment argument can define segments — a segment element excluded by the mask cannot begin a segment even if the segment element’s value is LOG_TRUE. When the segment_mode is CMCOM_segment_bit, the segment argument is unaffected by the mask.

For arithmetic scans only:

When the inclusion is CMCOM_exclusive, the value of the first element in each segment depends on the segment mode. When the segment_mode is CMCOM_start_bit, the first element in a segment (which would otherwise be undefined) receives the scan result from all of the contributing elements in the previous segment, or the scan operation’s identity value if there is no previous segment or if none of the previous segment’s elements were selected. When the segment_mode is CMCOM_start_bit, the first element in a segment receives the scan operation’s identity value.

For copy scans only:

The scan result of a segment is the value of the source at the first element in the segment selected by the mask. If no elements in the segment are selected, the result is zero.

When the inclusion is CMCOM_exclusive, the value of the first element in each segment depends on the segment mode. When the segment_mode is CMCOM_start_bit, the first element in a segment (which would otherwise be undefined) receives the scan result from all of the contributing elements in the previous segment, or zero if there is no previous segment or if none of the previous segment’s elements were selected. When the segment_mode is CMCOM_start_bit, the first element in a segment receives a zero value.
Effects of Scan Arguments:

As an example, here is a table showing the effects of various combinations of the `direction`, `inclusion`, and `segment_mode` arguments on a one-dimensional addition scan.

**Note:** For the mask and segment arrays, a `LOG_TRUE` value is represented by a "1", and a `LOG_FALSE` value by a "0". Unmodified destination array elements are indicated by a ".".

```
CMRT_scan_add(dest, source, segment, axis,
              direction, inclusion, segment_mode, mask)
```

| mask array: | 1 1 1 0 0 0 1 1 0 1 1 1 0 |
| segment array: | 0 0 0 1 0 0 1 0 0 0 0 0 1 0 0 |
| source array: | 1 1 1 1 1 1 1 1 1 1 1 1 1 1 |

<table>
<thead>
<tr>
<th>direction</th>
<th>inclusion</th>
<th>segment_mode</th>
<th>dest array values:</th>
</tr>
</thead>
<tbody>
<tr>
<td>upward</td>
<td>exclusive</td>
<td>none</td>
<td>0 1 2 3 . . 4 5 . . 6 7 8 .</td>
</tr>
<tr>
<td>downward</td>
<td>exclusive</td>
<td>none</td>
<td>8 7 6 5 . . 4 3 . . 2 1 0 .</td>
</tr>
<tr>
<td>upward</td>
<td>inclusive</td>
<td>none</td>
<td>1 2 3 4 . . 5 6 . . 7 8 9 .</td>
</tr>
<tr>
<td>downward</td>
<td>inclusive</td>
<td>none</td>
<td>9 8 7 6 . . 5 4 . . 3 2 1 .</td>
</tr>
<tr>
<td>upward</td>
<td>exclusive</td>
<td>segment_bit</td>
<td>0 1 0 1 . . 0 1 . . 2 0 1 .</td>
</tr>
<tr>
<td>downward</td>
<td>exclusive</td>
<td>segment_bit</td>
<td>1 0 1 0 . . 2 1 . . 0 1 0 .</td>
</tr>
<tr>
<td>upward</td>
<td>inclusive</td>
<td>segment_bit</td>
<td>1 2 1 2 . . 1 2 . . 3 1 2 .</td>
</tr>
<tr>
<td>downward</td>
<td>inclusive</td>
<td>segment_bit</td>
<td>2 1 2 1 . . 3 2 . . 1 2 1 .</td>
</tr>
<tr>
<td>upward</td>
<td>exclusive</td>
<td>start_bit</td>
<td>0 1 2 1 . . 2 3 . . 4 5 1 .</td>
</tr>
<tr>
<td>downward</td>
<td>exclusive</td>
<td>start_bit</td>
<td>2 1 5 4 . . 3 2 . . 1 1 0 .</td>
</tr>
<tr>
<td>upward</td>
<td>inclusive</td>
<td>start_bit</td>
<td>1 2 1 2 . . 3 4 . . 5 1 2 .</td>
</tr>
<tr>
<td>downward</td>
<td>inclusive</td>
<td>start_bit</td>
<td>3 2 1 5 . . 4 3 . . 2 1 1 .</td>
</tr>
</tbody>
</table>
```
1.14.2 CMRT_reduce_[combiner]

    void CMRT_reduce_[combiner](dest, source, axis, coordinate, mask)
    combiner = logior, logand, count, max, min, product, sum

    CMRT_desc_t dest, source;
    int4 axis;
    int8 coordinate;
    CMRT_desc_t mask;

Description: These functions perform a reduction scan along the specified axis of the source array, cumulatively combining all elements along the axis using the indicated combiner, and then store the combined result at the specified coordinate along the axis. The mask argument controls which elements of the source array are used in the reduction operation.

1.14.3 CMRT_spread_copy

    void CMRT_spread_copy(dest, source, axis, coordinate, mask)
    CMRT_desc_t dest, source;
    int4 axis;
    int8 coordinate;
    CMRT_desc_t mask;

Description: This function "spreads" (copies) the array elements of the source array that are at the specified coordinate of the given axis to all coordinates along that axis. The result of the spread is stored in the dest array.

Essentially, this function copies one "column" of the array (at the given coordinate along the given axis) to all other "columns" along that same axis.

The mask argument controls which elements of the dest array are modified:
1.14.4 CMRT.multispread_copy

```c
void CMRT_multispread_copy(dest, source, axes_mask, coordinate, mask)
CMRT_desc_t dest, source;
unsigned4 axes_mask;
int8 *coordinate;
CMRT_desc_t mask;
```

**Description:** This function is much like CMRT.spread.copy, except that it allows multiple spread axes to be specified. The axes_mask is an integer mask, which should contain a 1 in each bit position corresponding to an axis from which values are to be copied. The coordinate argument should be an array of as many values as there are axes in the source array.

For each axis selected by the axes_mask, the corresponding value of coordinate specifies the coordinate along that axis from which a source array value is to be copied to all elements along that axis.

You can think of this operation as a way to specify more than one axis along which to spread values. For example, in a three-dimensional array, you can specify two dimensions as the axes to spread over, so that every two-dimensional "plane" of array elements that lies parallel to those two axes will receive a copy of the single value located at the intersection of the two specified coordinate values. For higher dimensions, the effect is similar but correspondingly more difficult to visualize.

The result of the multispread operation is stored in the dest array. The mask argument controls which elements of the dest array are modified.

1.15 Array Reduction (Global) Functions

The functions described in this section are used to combine many values from a CM array into a single value, using one of several possible combining operations.

**Note:** While the combining operation is deterministic (and therefore repeatable), code should not depend on the order in which values are combined. For example, the combination order can affect whether or not overflow occurs. It can also result in loss of precision when large and small floating point numbers are combined.

The available combiners fall into two major categories: typed and logical. The four typed combiners (max, min, add, and product) have separate routines for each data type. The three logical combiners (logical "or", logical "exclusive or", and logical "and") operate on arbitrary 32-bit values.
1.15.1 CMRT_global_[combiner]

[type] CMRT_global_[combiner](source, mask)
CMRT_desc_t source, mask;

Description: These functions globally combine all the elements of the source CM array, using the indicated combiner operation, and return the combined value. The mask argument controls which elements of the source array are combined.

The defined combiner names (and corresponding type names) are:

- **sum_[type]** — arithmetic sum
  (type = int4, int8, unsigned4, real4, real8, cmpx8, cmpx16, duint, dint)
- **product_[type]** — arithmetic product
  (type = int4, int8, unsigned4, real4, real8, cmpx8, cmpx16, duint, dint)
- **max_[type]** — numeric maximum
  (type = int4, int8, unsigned4, real4, real8, duint, dint)
- **min_[type]** — numeric minimum
  (type = int4, int8, unsigned4, real4, real8, duint, dint)
- **logior** [log4] — 32-bit logical inclusive OR
- **logxor** [log4] — 32-bit logical exclusive OR
- **logand** [log4] — 32-bit logical AND
- **count** [int8] — number of LOG_TRUE values

Restrictions: The unsigned4 versions of the sum, product, max, and min operators are only defined in the CM-5 versions of the RTS. The dint and duint versions of the sum, product, max, and min operators are only defined in the CM-5/VU version of the RTS.
1.15.2 CMRT_global_project_[type]

[type] CMRT_global_project_[type] (source, fill, mask)
CMRT_desc_t source, mask;
[type] fill;

[type] = int4, log4, real4, real8, cmpx8, cmpx16, dint, duint

Description: These functions return the single value of the source array selected by the mask, which can contain at most one LOG_TRUE value. If no source element is selected, the fill value is returned, instead.

Restrictions: The dint and duint functions are only defined in the CM-5/VU version of the RTS.

1.16 Enumeration, Ranking, and Sorting

The functions described in this section are used to perform simple numeric ranking and sorting of CM array elements, as well as to enumerate selected elements of a CM array.

1.16.1 CMRT enumerate

void CMRT enumerate(destination, mask)
CMRT_desc_t destination, mask;

Description: This function enumerates the elements of destination in Fortran (column-major) order of axis elements. (The elements of destination are modified to contain the enumeration values.) The values stored in the destination array are integers ranging from 0 up to 1 less than the total number of elements selected by the mask argument.

The mask controls which elements of the destination are counted in the enumeration — in other words, if an element is not selected by the mask, it is "skipped over" in the enumeration.

Performance: CMRT enumerate is more expensive for N-d geometries (geometry rank \( \geq 1 \)). For 1d geometries the time per element improves as the subgrid size increases, since a higher portion of the time is spent doing on-processor computation (as with scans).
1.16.2 CMRT_rank, CMRT_sort

void CMRT_rank(dest, source, segment, axis, direction, segment_mode, mask)
void CMRT_sort(dest, source, segment, axis, direction, segment_mode, mask)

CMRT_desc_t dest, source, segment, mask;
int4 axis;
CMCOM_direction_t direction;
CMCOM_segment_mode_t segment_mode;

Description: CMRT_rank ranks the elements of the source array along the specified array axis. CMRT_sort sorts the elements of the source array along the specified array axis. Both functions store the ranked or sorted values in the destination array.

For example, if the source array contains the four values:

\[
1.0 \ 7.0 \ 3.0 \ 2.0
\]

then for a CMCOM_upward rank operation, the dest result is:

\[
0 \ 3 \ 2 \ 1
\]

and for a CMCOM_upward sort operation, the dest result is:

\[
1.0 \ 2.0 \ 3.0 \ 7.0
\]

The data type of the source and dest arrays can be either integer or floating-point values. (The CM-5/VU RTS version allows doubleword signed integer array values.) In addition, the dest and source array arguments may exactly overlap. The direction, segment_mode, segment, and mask arguments are as described for the CMRT_scan operations.

Note: CMRT_rank can be used to create a random permutation, by ranking an array of random numbers.
Restrictions: Currently, only the CM-5/VU RTS supports segmented sort or rank. Also, only the CM5/VU RTS supports double-word signed integer arguments to the sort and rank functions.

Performance: CMRT_sort is equivalent to doing a CMRT_rank and a send, but is faster in some cases.

Sorting is generally more efficient for big subgrids. The time per element for sort or rank improves as the subgrid size increases.

Sorting double-word keys (e.g. double floats) is approximately twice as slow as sorting single-word keys (e.g. floats or 4-byte integers). Sorting small integers (ranging from 0 to a value less than $2^{32}$) can be faster than sorting arbitrary integers.

Segmented and n-dimensional sorts and ranks can be significantly slower than 1-d unsegmented sorts and ranks.

1.17 Array Manipulation Utilities

The functions described in this section are used to perform large-scale manipulations and tests on CM arrays.

1.17.1 CMRT_[firstloc/lastloc]_{_axis}

\begin{verbatim}
void CMRT_[firstloc/lastloc](dest, source)
void CMRT_[firstloc/lastloc]_axis(dest, source, axis)
CMRT_desc_t dest, source;
int4 axis;
\end{verbatim}

Description: These functions are used to locate the first or last LOG_TRUE value in an array, in Fortran (column-major) order.

For the CMRT_[firstloc/lastloc] functions, a single LOG_TRUE value is stored in the element of the dest array corresponding to the first LOG_TRUE value in the entire source array.

For the CMRT_[firstloc/lastloc]_axis functions, this operation is performed independently for every "row" of the source array that is parallel to the specified axis.

Note: There is no mask argument in these functions, — by default, all array elements are included in the operation.
1.17.2 CMRT_[maxloc/minloc]

    void CMRT_[maxloc/minloc](dest, source, mask)
    int8 *dest;
    CMRT_desc_t source;
    CMRT_desc_t mask;

Description: These functions locate the maximum (or minimum) element in the source array, and place a LOG_TRUE value in the corresponding element of the dest array.

The mask argument controls which elements of the source are checked for maximum (or minimum) values.

1.17.3 CMRT_[pack,unpack]

    void CMRT_[pack,unpack](dest, source, mask)
    CMRT_desc_t dest, source, mask;

Description: These functions convert between N-D and 1-D arrays by “packing” and “unpacking” array elements. (Essentially, these functions convert an array into a very long vector.) The 1-D array must have at least as many total elements as the N-D array.

CMRT_pack converts an N-D source to a 1-D dest array by storing the source elements into the dest array in Fortran (column-major) order.

CMRT_unpack reverses this process, converting a 1-D source to an N-D dest array by extracting the source elements in Fortran (column-major) order and storing them in dest.
1.17.4 CMRT_project

void CMRT_project(dest, fill, source, axis, coordinate, mask)
CMRT_desc_t fill;
CMRT_desc_t dest;
CMRT_desc_t source;
int4 axis;
int8 coordinate;
CMRT_desc_t mask;

Description: This function “collapses” the source array along the specified axis, thereby reducing it by one dimension, and stores the result in dest. The mask argument is used to select which element along each “row” of the source array is copied into the dest array. If no mask element is LOG_TRUE for a given row, then an element of the fill array is copied into the dest.

The coordinate argument specifies which fill element along the axis to copy, and also specifies which dest element along the axis receives the result. (In other words, the coordinate argument selects the “slice” of the fill and dest arrays that are used in the operation.)

Restrictions: All array arguments must have the same rank and shape. The mask argument can have no more than one LOG_TRUE value for any “row.”

1.17.5 CMRT_reshape

void CMRT_reshape(dest, source, pad, order)
CMRT_desc_t pad, dest, source;
int4 *order;

Description: This function is used to “reshape” an array, by copying it into an array with a completely different geometry. Array elements are copied from source to dest in Fortran (column-major) order. If there are more elements in dest than there are in source, the pad array is copied in Fortran order into the remaining dest elements as many times as necessary.
1.18 Vector and Matrix Multiply Functions

1.18.1 CMRT_vector_matrix_multiply, CMRT_matrix_vector_multiply

```c
void CMRT_vector_matrix_multiply (dest, source_vector, source_matrix)
void CMRT_matrix_vector_multiply (dest, source_matrix, source_vector)
CMRT_desc_t dest, source_matrix, source_vector;
```

Description: These functions perform a matrix multiplication of the supplied vector and matrix arguments, and store the result in the supplied dest array argument.

Restrictions: For CMRT_vector_matrix_multiply, the vector must be 1 by n, the matrix must be n by p, and the dest must be 1 by p. (That is, the dimensions of each array must be appropriate for a vector/matrix multiplication.) Similarly, for CMRT_matrix_vector_multiply, the matrix must be p by n, the vector must be n by 1, and the dest must be p by 1.

1.18.2 CMRT_matrix_multiply

```c
void CMRT_matrix_multiply (dest, source1, source2)
CMRT_desc_t dest, source1, source2;
```

Description: This function performs a matrix multiplication of the supplied matrix arguments (source1 and source2), and stores the result in the supplied dest array argument.

Restrictions: All the array arguments must be two-dimensional. If source1 is m by n, and source2 is n by p, then the dest argument must be m by p. (That is, the shape and size of each array must be appropriate for a matrix multiplication.)
1.19 Random Number Generation Functions

The functions defined in this section are used to generate random numbers in a CM array. The random number generator (RNG) used is Wolfram's Rule 30 Cellular Automaton. The primary reference for this is Steven Wolfram, "Random Sequence Generation by Cellular Automata", Advances in Applied Mathematics 7, pp. 123-169 (1986). This paper may be more readily available as a reprint in Stephen Wolfram, Theory and Application of Cellular Automata (including selected papers 1983-1986), World Scientific (1986).

Wolfram's RNG automaton is a one dimensional cellular automaton of N cells, with each cell represented by one bit \( b[i] \), where \( i=0,...,N-1 \). At each tick of the clock, all cells update as follows:

\[
\begin{align*}
\text{Wolfram noticed that if you pick out one bit (e.g. } i=0), \text{ then when N is sufficiently large the values of that bit at successive time steps comprise an effectively pseudorandom sequence. Note that picking two or more bits from the same time step causes the selected bits to be highly correlated; it is only when the same bit is examined at different time steps that a useful pseudorandom sequence is generated. This means that at most one bit can be generated for each step of the RNG, so 32 steps are required to generate a one-word random integer.}
\end{align*}
\]

1.19.1 The RTS Random Number Generator

The current CM implementation uses a Wolfram CA RNG with \( N=59 \), with a separate space of cells defined for each CM node. Thus, a single random bit can be generated for each node in a single time-step. The RNG is run through as many steps as needed to produce a random value of the requested type (integer or float). This process is then repeated as many times as necessary to fill all the subgrid elements of an array argument, using a different seed value for each subgrid element.

The RNG is initialized when it is first used, and uses space in the CM heap to store the seed value(s) used in its calculations.

On the CM-2, this space is established by the user, through a call to an initialization function. On the CM-5, this space is internally allocated, and the user only needs to provide a seed value used to initialize it. The RNG may be reinitialized by command at any time.

**Note:** If the initialization functions are not called prior to the random-number functions, the initializations are implicitly called the first time the random-number functions are used, to initialize the RNG with an constant internal seed value.

Before generating each element of a subgrid, the RNG state is initialized to a value taken from a
seed array. Depending on the RNG version used (virtual or physical), either an array of seed values is used, one for each element of the subgrid, or a single seed value is used for every subgrid element.

The RNG initialization functions define the seed array, and initialize it in each node to a value determined by combining a user-provided seed value with the physical self-address of each node. This ensures that the initial RNG state will be unique for each node. Before the first use, the automaton is run for some number of cycles (currently 100) to allow it to "thermalize" and become truly pseudorandom.

Integer random numbers are generated by running the RNG for \( n \times s \) iterations, where \( n \) is the number of bits (32) and \( s \) is the subgrid size, picking out a single bit at each iteration.

Floating point numbers are generated in three steps: first, the above process is used to generate the significand (23 bits) of the number; next, the exponent (8 bits) and the sign are set to produce a number uniformly distributed between 1.0 and 2.0; finally, a float subtract by 1.0 is done to yield a number uniformly distributed between 0.0 and 1.0.

1.19.2 CMRT_random

```c
void CMRT_random(dest, mask, limit)
  CMRT_desc_t dest, mask;
  int4 limit;
```

**Description:** This function stores a random value in each element of the supplied `dest` array.

The `mask` argument controls which elements of the `dest` are modified.

If `dest` is an integer array, then the values generated will be integers in the range 0 to `limit-1` (inclusive). For floating-point `dest` arrays, the values produced will be positive and strictly less than `limit`.

If `limit` is 0, the values produced will be random 32-bit patterns. If interpreted as signed numbers, these values may be negative. A limit of 0x80000000 may be used to generate positive signed integers.

An array of seed values is used, one for each subgrid element. The seed array used by the RNG is the array last specified as an argument to `CMRT_initialize_random`, as long as this seed array has the same subgrid size as `dest`. If `CMRT_initialize_random` has not yet been called, or was most recently called with a seed array having a different subgrid size than `dest`, an internal seed array will be automatically allocated and seeded with an internal seed value.

**Restrictions:** This is the only random number function that is defined for both the CM-2 and CM-5. It compiles into the appropriate random-number code based on the RTS library that is used.
1.19.3 CMRT_physical_random

void CMRT_physical_random(dest, mask, limit)
CMRT_desc_t dest, mask;
int4 limit;

Description: This function is identical to CMRT_random, except that only one seed value is
used to generate all values for each subgrid. This "physical" seed is internally generated, or may
be set by the CMRT_initialize_physical_random function.

Restrictions: This operation is defined only in the CM-2 version of the RTS.

1.19.4 CMRT_initialize_random, CMRT_initialize_physical_random

void CMRT_initialize_random(dest, mask, seed64)
CMRT_desc_t dest, mask;
int8 *seed64;

void CMRT_initialize_physical_random(seed64)
int8 *seed64;

Description: These functions initialize the seed array (or value) used by the CM-2 random
number functions.

CMRT_initialize_random initializes the dest array to a set of initial random seed values. The
value seed64 is a pointer to a 2-word value to be used to initialize this array. The resulting array
will be used as the seed for all subsequent random number generations via CMRT_random on
 grids with a subgrid size that matches dest.

CMRT_initialize_physical_random initializes the single per-subgrid seed used by the physical
RNG function CMRT_physical_random. seed64 is a pointer to a 64-bit (two-word) seed value.

Restrictions: These operations are defined only in the CM-2 version of the RTS.
1.19.5 CMRT_deallocate_rng, CMRT_deallocate_physical_rng

void CMRT_deallocate_rng()
void CMRT_deallocate_physical_rng()

Description: These operations free up the heap space occupied by the seed value(s) of the CM-2 random number generator. Note: If these functions are called to deallocate the seed value space, and are not followed by a call to the RNG initialization functions to reallocate this space, then any succeeding calls to the RNG functions will cause the seed value space to be automatically reallocated.

Restrictions: These operations are defined only in the CM-2 version of the RTS.

1.19.6 CMRT_random_seed_location

CMRT_cm_location_t CMRT_random_seed_location()

Description: This routine returns the CM heap memory location of the current seed array in use by the CM-2 RNG functions.

Restrictions: This operation is defined only in the CM-2 version of the RTS.
1.20 CM-2 Low-level Interface Functions

The functions described in this section are used to perform low-level CM-2 operations such as translating between Paris fields and CM arrays, and calling CMIS IMP code routines.

Note: These functions are defined only in the CM-2 version of the RTS.

1.20.1 CMRT_copy_to/from_paris_field

void CMRT_copy_to_paris_field(paris_field, source, paris_axis_mappings)
void CMRT_copy_from_paris_field(dest, paris_field, paris_axis_mappings)
CM_field_id_t paris_field;
CMRT_desc_t dest;
CMRT_desc_t source;
int *paris_axis_mappings;

Description: These functions copy array data from CMRTS array format to Paris field format. The paris_axis_mappings argument must be an array of integers that determines the correspondence of RTS array axes to Paris VP-set axes. (This argument works much like the axis_mapping argument of CMRT_cross_geometry_move, except that all axes must be mapped explicitly — there are no CMRT_NO_AXIS values here.)

Restrictions: These functions are defined in the CM-2 RTS only. They are intended primarily to provide compatibility with useful Paris routines.

1.20.2 CMRT_load_code

void CMRT_load_code(code_ptr)
IMP_imp_descriptor_ptr_t code_ptr;

Description: This function, when given a pointer to an IMP (internal macroinstruction procedure) routine, loads the IMP code into CM microprocessor memory

Restrictions: This function is defined in the CM-2 RTS only.
1.20.3 CMRT_push_address

```c
void CMRT_push_address(addr)
    int4 addr;
```

**Description:** This function pushes an address down the instruction FIFO for use by an IMP routine.

**Restrictions:** This function is defined in the CM-2 RTS only. This function should only be called after an appropriate `CMRT_load_code` call has loaded the IMP code, and should be used only to provide arguments to the IMP.

1.20.4 CMRT_push_[type]

```c
void CMRT_push_[type](ival)
    type = int4, real4, real8, cmpx8, cmpx16
    [type] ival;
```

**Description:** This function pushes a value down the instruction FIFO for use by an IMP routine.

**Restrictions:** This function is defined in the CM-2 RTS only. This function should only be called after an appropriate `CMRT_load_code` call has loaded the IMP code, and should be used only to provide arguments to the IMP.
1.21 Indirect Addressing Functions

The functions in this section implement lookup tables and indirect array referencing. Note: Since these routines are designed to be called from Fortran, array indices are 1-based, not 0-based as with other RTS functions.

1.21.1 CMRT_aref_1d

```c
define CMRT_aref_1d(dest, source, index, mask)
    CMRT_desc_t dest;
    CMRT_desc_t source;
    CMRT_desc_t index;
    CMRT_desc_t mask;
```

Description: This function is a CMRT interface to the function CMIP_aref_1d_any_order. It implements a FORTRAN-style indirect addressing mechanism that treats the subgrid of each array argument as either a vector or a table of vectors.

The `dest` and `index` arguments are treated as vectors of a length equal to their subgrid length, and `source` is treated as a table of `n` vectors of the same length, where `n` is as large as the `source` subgrid size will permit.

The value in each element of `index` determines which element of the corresponding vector in `source` is stored in `dest`. The `mask` argument controls which elements of `dest` are modified.

For more information, see the description of CMIP_aref_1d_any_order.

1.21.2 CMRT_aset_1d

```c
define CMRT_aset_1d(dest, source, index, mask)
    CMRT_desc_t dest;
    CMRT_desc_t source;
    CMRT_desc_t index;
    CMRT_desc_t mask;
```

Description: This function is a CMRT interface to the function CMIP_aset_1d_any_order. It implements a FORTRAN-style indirect addressing mechanism that treats the subgrid of each array argument as either a vector or a table of vectors.

The `source` and `index` arguments are treated as vectors of a length equal to their subgrid length, and `dest` is treated as a table of `n` vectors of the same length, where `n` is as large as the `source` subgrid size will permit.
The value in each element of index determines which element of the corresponding vector in dest receives the corresponding value of source. The mask argument controls which elements of source are copied, and thus which elements of dest are modified.

For more information, see the description of CMIP_nset_1d_any_order.

1.21.3 CMRT_lookup_in_table

void CMRT_lookup_in_table(dest, table_address, index, mask)
    CMRT_desc_t dest;
    CMRT_cm_location_t table_address;
    CMRT_desc_t index;
    CMRT_desc_t mask;

Description: This function is a CMRT interface to the function CMIP_lookup_in_table. It implements a FORTRAN-style indirect addressing mechanism that treats the subgrid of each array argument as either a vector or a table of vectors.

The dest, table_address, and index arguments are all treated as vectors of a length equal to their subgrid length. The table_address argument is assumed to be the starting location of a machine array of the appropriate size.

The value in each element of index determines which element of the table_address array is stored in dest. The mask argument controls which elements of dest are modified.

For more information, see the description of CMRT_lookup_in_table.
1.22 Miscellaneous CMRT Functions

1.22.1 CMRT_fp_check_safety_[pre/post]

```c
void CMRT_fp_check_safety_pre(array, varname, mask)
    CMRT_desc_t array;
    char *varname;
    CMRT_desc_t mask;

CMRT_make_triplet_mask

void CMRT_make_triplet_mask (ctxt, mask, coord, lower, upper, stride)
    CMRT_desc_t ctxt, mask, coord;
    int4 lower, upper, stride;
```

**Description:** This function fills the array `ctxt` with an unpacked mask that selects elements according to the FORTRAN vector triplet subscript (`lower:upper:stride`). In other words, each element is the result will be set to true (LOG_TRUE) if the corresponding value in `coord` falls between `lower` and `upper` inclusive and is an even multiple of `stride` away from `lower`, and if the corresponding element in `mask` is non-zero. Result elements in all other cases will be written with LOG_FALSE (zero). In the SPARC RTS, only the low byte of each element of `mask` is sensed.
2 The CMCOM Interface

This section lists and describes the functions available in the CMCOM software layer. The functions are listed by category, in the following order:

- CMCOM initialization
- CMCOM environment functions
- Machine geometries and garbage masks
- CM memory management
- Address arithmetic functions
- Send and grid addresses
- Broadcast functions
- Processor read/write functions
- Array read/write functions
- General communication function (send and get)
- Cross-geometry move function
- Rotate/shift communication functions
- Scans/reductions/spreads
- Array reduction (global) functions
- Rank (sorting) functions
- CM-2 RTS interface to Paris

2.1 CMCOM Function Naming Conventions

A CMCOM function name has the general form:

```
CMCOM_{[on_chip/physical]}_[function]_[data type]_[combiner]
```

Functions with on_chip in their names are the on-chip part of a CMCOM operator, and functions with physical in their names are the physical part. Functions with neither on_chip nor physical in their names are usually the virtual part of a CMCOM operator.
The function class component of the name indicates the type of operation being performed, such as scan, spread, or reduce.

The data type portion of the name is optional, and when present is an abbreviation that indicates the data type of the operation:

- `u` indicates an unsigned4 operation.
- `s` indicates an int4 operation.
- `du` indicates a duint (unsigned8) operation.
- `ds` indicates a dint (int8) operation.
- `f` indicates an real4 operation.
- `df` indicates an real8 operation.
- `c` indicates an cmpx8 operation.
- `dc` indicates an cmpx16 operation.

Note: Currently ds and du functions exist only in the CM-5/VU version of the RTS.

The combiner portion of the name indicates the method used to combine elements that are brought together by operations such as scans, reductions, and sends. It is typically the name of an operation such as add for addition, max for maximum, etc.

2.2 CMCOM Function Arguments

As described in the Concepts section, CMCOM functions refer to arrays in terms of their actual memory locations. To be specific, a CMCOM function refers to an array by the following three pieces of information:

- a CMCOM_cm_address_t value, which gives the array’s location in CM memory
- a subgrid increment, an integer giving the increment in words between successive subgrid elements
- a machine geometry object (CMCOM_machine.geometry.t), which describes the layout of the array’s elements in CM memory. (Note: The geometry is the only way a CMCOM function can tell how the array’s elements are laid out in memory.)
The arguments to CMCOM functions can generally be divided into four groups:

1. CM array arguments (source, destination, mask, etc.) and their corresponding increments
2. a geometry object and element size, which generally apply to all the array arguments
3. a garbage mask array, if one is required by the function
4. other values, such as an axis number, or coordinate

There are a number of "standard" arguments that most CMCOM functions have, such as the source, destination, and mask array arguments. These arguments can be identified by their names:

- **source** (source1, source2, etc.) — The source array(s) from which values are read.
- **dest** (dest_array, etc.) — The destination array into which the result is written.
- **mask** — The user mask argument (or NULL, to indicate that no masking is needed).
- **argument_inc** — Byte increment between adjacent elements in the argument array.
- **element_size** — The size, in bytes, of each array element. (In other words, the number of data bytes that must be operated on.)
  
  Note: The increment does not have to be the same as element_size! Increments and element sizes can be different, to allow useful striding patterns through array data.

- **geometry** — A geometry object, as returned by CMCOM.intern.Geometry.
- **garbage_mask** — A garbage mask array (or NULL, indicating no garbage mask is needed).
- **axis** — An integer (zero-based), selecting an axis of a geometry.
- **coordinate** — An integer (zero-based), representing a coordinate along an array axis.
- **send_address** — A send-address value.

These "standard" arguments are not described separately for each function; rather, only arguments unique to each function or unique meanings of the above arguments are described.
2.3 CMCOM Initialization

This section describes functions that are used to initialize the CMCOM software layer and to obtain information about the current CM hardware.

2.3.1 CMCOM_init

    void CMCOM_init()

Description: CMCOM_init initializes the CMCOM software layer. This function must be called before any CMCOM operations can be used. Note: CMCOM_init is called by CMRT_init.

2.3.2 CMCOM_pe_[length, limit]

int CMCOM_pe_[length, limit]()

Description: These functions return information about the available nodes. CMRT_pe_limit returns the number of nodes in the partition. CMRT_pe_length returns the number of bits required to represent the limit (in other words, \( \log_2(\text{CMRT.pe.limit}) \)).

2.3.3 CMCOM_[attached, initialization_check, safe_cm_init]

int4 CMCOMAttached()
void CMCOM_initialization_check()
void CMCOM_safe_cm_init()

Description:
CMCOMAttached returns TRUE if CM hardware is attached, FALSE otherwise.
CMCOM_initialization_check tests whether a CM is attached, and whether it is initialized. If either test fails, an error is signaled.
CMCOM_safe_cm_init initializes the CM variables that support the RTS.

Note: CMCOM_safe_cm_init is called by CMCOM_init.

Restrictions: These functions exist only in the CM-2 version of the RTS.
2.4 Machine Geometries and Garbage Masks

The functions described in this section are used to define machine geometry objects, which can be used to specify the shape and size of arrays. For more information on the arguments and use of these functions, see Section 4.

**Implementation Note:** Newly created geometry objects are internally stored ("interned") so that they can be re-used. When the arguments to the geometry-creating functions below specify a geometry that has previously been defined, an existing geometry object is returned, rather than a new geometry object with the same properties. This reduces overhead and redundancy by allowing several arrays to share the same geometry object.

2.4.1 CMCOM_intern{specific}.geometry

```c
CMCOM_machine_geometry_t
CMCOM_intern_geometry(rank, orders, off_chip_positions, off_chip_lengths,
                        subgrid_axis_lengths, is_serial,
                        highest_axis_varies_fastest)

CMCOM_machine_geometry_t
CMCOM_intern_specific_geometry(rank, orders, axis_permutation,
                                 off_chip_masks, subgrid_axis_lengths)

CMCOM_order_t *orders;
int4 rank, *off_chip_positions, *off_chip_lengths, *subgrid_axis_lengths;
log4 *is_serial, highest_axis_varies_fastest;
unsigned4 *off_chip_masks;
int4 *axis_permutation;
```

**Description:** These functions each define and return a machine geometry object.

**CMCOM_intern_geometry** defines a geometry based on a set of off-chip positions and lengths.

**CMCOM_intern_specific_geometry** defines a geometry based on an axis ordering and a set of off-chip masks.
2.4.2 CMCOM_print_geometry

```c
void CMCOM_print_geometry(g)
CMCOM_machine_geometry_t g;
```

**Description:** This function prints a nicely formatted description of the supplied geometry to the standard output.

2.4.3 CMCOM_create_garbage_mask

```c
void CMCOM_create_garbage_mask(extents, geometry, garbage_mask)
int8 *extents;
CMCOM_machine_geometry_t geometry;
CMCOM_cm_address_t garbage_mask;
```

**Description:** This function modifies the `garbage_mask` argument to contain an appropriate garbage mask for the given geometry and array extents. This function is used by the CMRT layer geometry operators to define the garbage masks for CMRT array geometries.

2.4.4 CMCOM_calculate_garbage_mask_size

```c
int4 CMCOM_calculate_garbage_mask_size(geometry)
CMCOM_machine_geometry_t geometry;
```

**Description:** This function calculates the amount of memory (in bytes) needed to store a garbage mask for an array with the specified machine geometry.
2.5 CM Memory Management

The functions described in this section are used to allocate and deallocate regions of CM memory.

2.5.1 CMCOM_allocate_stack_space, CMCOM_allocate_stack_words

CMCOM_cm_address_t CMCOM_allocate_stack_space(nbytes)
CMCOM_cm_address_t CMCOM_allocate_stack_words(nwords)

unsigned4 nbytes, nwords;

Description: These functions allocate the requested number of bytes (or words) of CM stack space, and return the address of the allocated space.

Restrictions: CMCOM_allocate_stack_space is not defined in the CM-2 RTS version.

2.5.2 CMCOM_available_memory

int4 CMCOM_available_memory()

Description: This function returns the amount of remaining CM memory (in bytes) available for use in either the heap or stack.

Performance Hints: This count does not include free space within the heap itself, because the heap space is not managed by the CMCOM layer.
2.5.3 CMCOM_get_stack_pointer

CMCOM_cm_address_t CMCOM_get_stack_pointer()

int4 CMCOM_set_stack_pointer(new_sp)
CMCOM_cm_address_t new_sp;

Description: These functions control the position of the CM stack pointer.
CMCOM_get_stack_pointer gets the current stack position.
CMCOM_set_stack_pointer sets the stack pointer to a new location.

2.5.4 CMCOM_get_heap_base

CMCOM_cm_address_t CMCOM_get_heap_base()
CMCOM_cm_address_t CMCOM_get_heap_limit()

Description: These functions return the CM memory address of the base (bottom) and top (limit) of the heap region.

2.5.5 CMCOM_expand_heap

unsigned4 CMCOM_expand_heap(size)
unsigned4 size;

Description: This function attempts to expand the heap region by the requested size (in bytes). It returns the actual amount of new heap space allocated, which can be anywhere from 0 up to the amount requested (both inclusive).

2.6 Address Arithmetic Functions

The functions described in this section are used to perform simple arithmetic on CM memory addresses (for example, to obtain offsets from an address). Note: This is the only legal interface for such address manipulations in the CMRTS.
2.6.1 CMCOM_add_offset_to_cm_location

CMCOM_cm_address_t CMCOM_add_offset_to_cm_location(address, num_bytes)
CMCOM_cm_address_t address;
int4 num_bytes;

Description: This function adds an offset in bytes to a CMCOM memory address, and returns the offset address.

Restrictions: Currently, this function is not defined in the CMSIM RTS.

2.6.2 CMCOM_df_[lsw,msw]_addr

CMCOM_cm_address_t CMCOM_df_[lsw,msw]_addr(address)
CMCOM_cm_address_t address;

Description: These functions both take the address of a CM array containing double-float values, and return an address corresponding to one of the two words of those values. CMCOM_df_lsw_addr returns the address for the least significant word. CMCOM_df_msw_addr returns the address for the most significant word.

Restrictions: Currently, this function is not defined in the CMSIM RTS.

2.6.3 CMCOM_[c,dc]_[realpart,imagpart]_addr

CMCOM_cm_address_t CMCOM_[c,dc]_[realpart,imagpart]_addr(address)
CMCOM_cm_address_t address;

Description: These functions both take the address of a CM array containing complex or double complex values, and return an address corresponding to one of the two components of those values. CMCOM_[c/dc]_realpart_addr returns the address of the real component. CMCOM_[c/dc]_imagpart_addr returns the address of the imaginary component.

Restrictions: Currently, these functions are not defined in the CMSIM RTS.
2.7 Send and Grid Addresses

The functions described in this section are used to manipulate send and grid addresses.

2.7.1 CMCOM_make_send_address

```c
void CMCOM_make_send_address(send_address, send_address_inc, geometry)
CMCOM_cm_address_t send_address;
int4 send_address_inc;
CMCOM_machine_geometry_t geometry;
```

**Description:** This function initializes the CM memory region `send_address` (with the specified `send_address_inc` and `geometry`) so that it contains zero send addresses. In other words, the first 2 words (8 bytes) of each element are cleared to zero. The `send_address` argument can then be used to construct new send addresses (i.e., by using `CMCOM_deposit_grid_coordinate_zero`).

**Note:** The `geometry` argument is used only to determine the subgrid length.

**Restrictions:** On the CM5, each element must be double-aligned.

2.7.2 CMCOM_my_send_address, CMCOM_my_grid_coordinate

```c
void CMCOM_my_send_address(send_address, send_address_inc, mask, mask_inc, geometry)
CMCOM_cm_address_t send_address, mask;
int4 send_address_inc, mask_inc;
CMCOM_machine_geometry_t geometry;

void CMCOM_my_grid_coordinate(coordinate, coordinate_inc, mask, mask_inc, geometry, axis)
CMCOM_cm_address_t coordinate, mask;
int4 coordinate_inc, mask_inc, axis;
CMCOM_machine_geometry_t geometry;
```

**Description:** These functions calculate the address of each array element (either send address or grid coordinate along an axis) for the given `geometry`, and store the address values into the appropriate destination array (`send_address` or `grid_address`). The `mask` argument controls which elements of `dest` are modified.
For CMCOM\_my\_send\_address, the send\_address array is filled with "identity" send addresses; that is, each element is filled with its own send address within the array. Currently, this means the subgrid on node N is filled with double words, each of which has a high-order word of N and a low-order word that increments from 0 to 1 less than the subgrid size, in order through memory.

For CMCOM\_my\_grid\_coordinate, the elements of the coordinate grid are written with grid coordinates along the specified axis.

Note: The geometry argument is used only to determine the subgrid lengths.

### 2.7.3 CMCOM\_physical\_pe\_address

```c
void CMCOM\_physical\_pe\_address(physical\_pe\_address)
CMCOM\_cm\_address\_t physical\_pe\_address;
```

**Description:** This function calculates and stores into physical\_pe\_address the physical address (in other words, the node number) of each of its elements.

Each element of the physical\_pe\_address argument, which is assumed to have a subgrid length of 1, is written with its own physical subgrid number. For the CM-2 and CM-5/SPARC RTS versions, this is the local PN number. For the CM-5/VU RTS, this is the local PN number shifted left by 2 bits (or the log2 of the number of VU's per PN if different from 4) OR'ed with the relative VU number (0 through the number of VU's per PN). For the CMSIM RTS version, the physical pe address is always 0.

### 2.7.4 CMCOM\_my\_physical\_grid\_coordinate

```c
void CMCOM\_my\_physical\_grid\_coordinate(coordinate, geometry, axis)
CMCOM\_cm\_address\_t coordinate;
CMCOM\_machine\_geometry\_t geometry;
int4 axis;
```

**Description:** This function stores into each element of the coordinate array the physical coordinate of that element along the specified axis of the specified geometry. (Note: the coordinate array is assumed to have a subgrid length of 1, and a type of CMRT\_grid\_coordinate.
For example, consider a 100 by 100 2-dimensional virtual grid spread onto 16 subgrids arranged in a 4 by 4 physical grid. The 225 elements in the grid within the coordinate ranges of \([25:49,25:49]\) will fall on the subgrid in the second row, second column of the physical grid.

If the function \texttt{CMCOM\_my\_physical\_grid\_coordinate} is called on such a geometry, the value placed in \texttt{coordinate} on subgrid 5 (which is the second subgrid in the second row) would be 2.

**Restrictions:** This function is defined only in the CM-2 RTS version.

### 2.7.5 \texttt{CMCOM\_deposit\_grid\_coordinate\{\_zero\}}

```c
void CMCOM_deposit_grid_coordinate(send_address, send_address_inc, 
    coordinate, coordinate_inc, mask, mask_inc, 
    source\_geometry, dest\_geometry, axis)
void CMCOM_deposit_grid_coordinate_zero(send_address, send_address_inc, 
    coordinate, coordinate_inc, mask, mask_inc, 
    source\_geometry, dest\_geometry, axis)
CMCOM\_cm\_address\_t coordinate, send\_address, mask;
int4 send\_address\_inc, coordinate\_inc, mask\_inc, axis;
CMCOM\_machine\_geometry\_t source\_geometry, dest\_geometry;
```

**Description:** These functions both modify the \texttt{send\_address} argument by "depositing" grid coordinate information so that the new \texttt{send\_address} values refer to the specified \texttt{coordinate} along the given \texttt{axis} of the \texttt{dest\_geometry}.

The \texttt{mask} argument controls which \texttt{send\_address} elements are modified.

The \texttt{source\_geometry} argument is the actual machine geometry of \texttt{send\_address, coordinate,} and \texttt{mask}. The \texttt{dest\_geometry} argument is the machine geometry in which the supplied coordinate and axis values are interpreted. (The \texttt{dest\_geometry} can be the same as or different from the \texttt{source\_geometry} argument. If different, the deposited grid information can be used to transfer data between geometries via communications functions.)

The \texttt{\_zero} version of this function assumes that the portion of the \texttt{send\_address} values that corresponds to the specified axis is already zero. This allows the function to execute substantially faster.

**NOTE:** If this assumption is violated, the results are unpredictable, and the function may produce invalid \texttt{send\_address} values.
For example, consider an element somewhere within the `send_address` array. Assume it contains the send address of the element at coordinates [3,5,6] within the specified `source_geometry` (the coordinates are in incrementing axis order, i.e. 3 is the axis 0 coordinate, 5 the axis 1 coordinate and so forth).

Also assume that at the same position in the `coordinate` grid (the position corresponding to the send address in the `send_address` grid), there is the integer value 20. After a call to `CMCOM_deposit_grid_coordinate` with `axis` set to 1, the destination element will contain the send address of the element at coordinates [3,20,6].

**Restrictions:** The elements of `coordinate` must be at least 8 bytes in size.

On the CM-5, each element in `send_address` must be doubleword aligned.

### 2.7.6 CMCOM_extract_grid_coordinate

```c
void CMCOM_extract_grid_coordinate(coordinate, coordinate_inc,
                                    send_address, send_address_inc, mask, mask_inc,
                                    source_geometry, dest_geometry, axis)
```

**Description:** This function determines the coordinate along the specified `axis` that each `send_address` element refers to. The coordinates are stored in the `coordinate` array.

The `mask` argument controls which `coordinate` elements are modified.

The `source_geometry` argument is the actual machine geometry of `send_address`, `coordinate`, and `mask`. The `dest_geometry` argument is the machine geometry in which the supplied `coordinate` and `axis` values are interpreted. (The `dest_geometry` can be the same as or different from the `source_geometry` argument. If different, the extracted grid information can be used to transfer data between geometries via communications functions.)

**Restrictions:** The elements of the `send_address` array, must be at least 8 bytes in size, and must be doubleword aligned on the CM5.
2.7.7 CMCOM_scalar_make_send_address

CMCOM_send_address_t CMCOM_scalar_make_send_address()

Description: This function creates and returns a new scalar send address object. (That is, a front end/partition manager data object that represents a single send address.) The address object initially has the value 0, so it can be used to construct new send addresses (i.e., by using CMCOM_scalar_deposit_grid_coordinate_zero).

2.7.8 CMCOM_scalar_deposit_[physical,subgrid]_address

void CMCOM_scalar_deposit_physical_address(send_address, physical_address)
CMCOM_send_address_t *send_address;
unsigned4 physical_address;

void CMCOM_scalar_deposit_subgrid_address(send_address, subgrid_address)
CMCOM_send_address_t *send_address;
unsigned4 subgrid_address;

Description: These functions overwrite either the physical portion or the subgrid portion of the supplied send_address argument with the supplied physical_address or subgrid_address value.

Restrictions: On the CM5, the send_address argument must be doubleword aligned.

2.7.9 CMCOM_scalar_extract_[physical,subgrid]_address

unsigned4 CMCOM_scalar_extract_subgrid_address(send_address)
CMCOM_send_address_t send_address;

unsigned4 CMCOM_scalar_extract_physical_address(send_address)
CMCOM_send_address_t send_address;

Description: These functions extract a either the physical or subgrid component from the supplied send address object.
2.7.10 CMCOM_scalar_deposit_grid_coordinate{._zero}

```c
void CMCOM_scalar_deposit_grid_coordinate(send_address, axis, coordinate, geometry)
void CMCOM_scalar_deposit_grid_coordinate_zero (send_address, axis, coordinate, geometry)
CMCOM_send_address_t *send_address;
int4 axis;
int8 coordinate;
CMCOM_machine_geometry_t geometry;
```

**Description:** These functions both "deposit" grid coordinate information into the specified send_address object. The send_address argument is modified so that it refers to the specified grid coordinate along the given axis, relative to the supplied machine geometry.

For example, assume send_address contains the send address of the element at coordinates [3,5,6] within the specified grid geometry (the coordinates are in incrementing axis order, i.e. 3 is the axis 0 coordinate, 5 the axis 1 coordinate and so forth). Assume also that the coordinate argument is given as 20. After a call to CMCOM_scalar_deposit_grid_coordinate with axis set to 1, the send address returned will reference the element at coordinates [3,20,6].

The._zero version of the function assumes that the portion of the send address corresponding to the specified axis is already zero. This allows it to execute substantially faster.

**NOTE:** If this assumption is violated, the results are unpredictable, and the function may produce invalid send addresses.

2.7.11 CMCOM_scalar_extract_grid_coordinate

```c
int8 CMCOM_scalar_extract_grid_coordinate(send_address, axis, geometry)
CMCOM_send_address_t send_address;
int4 axis;
CMCOM_machine_geometry_t geometry;
```

**Description:** This function determines and returns the coordinate along the specified axis in the supplied machine geometry to which the send_address argument refers.
2.8 Broadcast Functions

The functions in this section are used to broadcast a single value from the partition manager to all nodes in a partition.

2.8.1 CMCOM_broadcast, CMCOM_physical_broadcast

```c
void CMCOM_broadcast(dest, dest_inc, mask, mask_inc,
                      geometry, source, element_size)
```

```c
CMCOM_cm_address_t dest, mask;
int4 dest_inc, mask_inc, element_size;
CMCOM_machine_geometry_t geometry;
unsigned4 *source;
```

```c
void CMCOM_physical_broadcast(dest, source, element_size)
```

```c
CMCOM_cm_address_t dest;
unsigned4 *source;
int4 element_size;
```

**Description:** The `CMCOM_broadcast` function "broadcasts" a single value to all elements of the `dest` array. The `mask` determines which values of `dest` are actually modified. The `CMCOM_physical_broadcast` function implements the physical portion of the operation.
2.9 Processor Read/Write Functions

The functions described in this section read (or write) an array value from a specific CM node.

2.9.1 CMCOM_read_from_processor, CMCOM_physical_read_from_processor

void CMCOM_read_from_processor(send_address, source, source_inc,  
dest, element_size)
CMCOM_send_address_t send_address;
CMCOM_cm_address_t source;
int4 source_inc, element_size;
void *dest;

void CMCOM_physical_read_from_processor(node_address, source,  
dest, element_size)
unsigned4 node_address, *dest;
CMCOM_cm_address_t source;
int4 element_size;

Description: The CMCOM_read_from_processor function reads a value of the source array from the element with the given send address.

CMCOM_physical_read_from_processor handles the physical component of the operation.

2.9.2 CMCOM_write_to_processor, CMCOM_physical_write_to_processor

void CMCOM_write_to_processor(send_address, dest, dest_inc,  
source, element_size)
CMCOM_send_address_t send_address;
CMCOM_cm_address_t dest;
int4 dest_inc, element_size;
void *source;

void CMCOM_physical_write_to_processor(node_address, dest,  
source, element_size)
unsigned4 node_address, *source;
CMCOM_cm_address_t dest;
int4 element_size;

Description: The CMCOM_write_to_processor function writes the source value into the dest array at the element with the given send address.
CMCOM_physical_write_to_processor handles the physical component of the operation.

2.9.3 CMCOM_physical_[read,write]_byte_[from,to]_processor

```c
void CMCOM_physical_read_byte_from_processor(pe_address, source, dest,
   element_size)

unsigned4 pe_address;
CMCOM_cm_address_t source;
char *dest;
int4 element_size;

void CMCOM_physical_write_byte_to_processor(pe_address, dest, source,
   element_size)

unsigned4 pe_address;
CMCOM_cm_address_t dest;
char *source;
int4 element_size;
```

**Description:** These functions read (or write) the number of bytes specified by element size from the memory of the processor specified by `pe_address`, starting at the specified `source` (or `dest`) CM address.

**Restrictions:** These functions are not defined in the CM-2 version of the RTS.

**Performance:** These functions are intended mainly for writing small chunks of data, usually less than the element size of an integer (4 bytes). As a result, they are slower than other read/write operations.
2.10 Array Read/Write Functions

The functions in this section are used to copy an array to (or from) the CM.

2.10.1 CMCOM_transfer_array_[from,to]_cm

```c
void CMCOM_transfer_array_from_cm(source, source_inc, geometry,
                                   extents, dest, dest_inc, element_size)

CMCOM_cm_address_t source;
int4 source_inc, dest_inc, element_size;
CMCOM_machine_geometry_t geometry;
int8 *extents;
unsigned4 *dest;

void CMCOM_transfer_array_to_cm(dest, dest_inc, geometry,
                                 extents, source, source_inc, element_size)

CMCOM_cm_address_t dest;
int4 dest_inc, source_inc, element_size;
int8 *extents;
CMCOM_machine_geometry_t geometry;
unsigned4 *source;
```

Description: These functions read (or write) an array between the partition manager and the nodes, with the specified geometry, increments, extents, etc.

Restrictions: The arrays on the partition manager and on the CM must both be sized to match the specified geometry, increment, and element size arguments.
2.10.2 CMCOM_transfer_vector_[from,to].cm

void CMCOM_transfer_vector_to_cm(dest,dest_inc,geometry,start_address,
                                 number_of_elements,source,element_size)

CMCOM_cm_address_t dest;
int4 dest_inc;
CMCOM_machine_geometry_t geometry;
CMCOM_send_address_t start_address;
int4 number_of_elements;
unsigned4 *source;
int4 element_size;

void CMCOM_transfer_vector_from_cm(source,source_inc,geometry,start_address,
                                    number_of_elements,dest,element_size)

CMCOM_cm_address_t source;
int4 source_inc;
CMCOM_machine_geometry_t geometry;
CMCOM_send_address_t start_address;
int4 number_of_elements;
void *dest;
int4 element_size;

Description: These functions read (or write) a portion of a CM array between the partition
manager and the nodes. The start_address argument specifies the send address of the first argu-
ment to be copied, and the number_of_elements argument specifies the number of array elements
copied. Array elements are copied in send address order (axis ordering is ignored). Hence, this
function effectively reads (or writes) a portion of a CM array as a very long vector, hence the name.

Restrictions: These functions are not defined in the CM-2 version of the RTS.
2.10.3 CMCOM_transfer_array_section_[from, to].cm

void CMCOM_transfer_array_section_from_cm(source, source_inc, geometry, extents, dest, dest_inc, element_size, lb, ub, st, cmaxis)

CMCOM_cm_address_t source;
int4 source_inc;
CMCOM_machine_geometry_t geometry;
int8 *extents;
unsigned4 *dest;
int4 dest_inc;
int4 element_size;
int8 *lb, *ub, *st;
int4 *cmaxis;

void CMCOM_transfer_array_section_to_cm(dest, dest_inc, geometry, extents, source, source_inc, element_size, lb, ub, st, cmaxis)

CMCOM_cm_address_t source;
int4 source_inc;
CMCOM_machine_geometry_t geometry;
int8 *extents;
unsigned4 *dest;
int4 dest_inc;
int4 element_size;
int8 *lb, *ub, *st;
int4 *cmaxis;

Description: These functions read (or write) a section of the source array. The array section copied is defined by the supplied lower bounds (lb), upper bounds (ub), and stride (st) arguments. Array elements are copied in column-major order.

The lb, ub, and st arguments are arrays of a length equal to the rank of the supplied geometry. lb[i] is the index of the first element that may be copied along axis i, ub[i] is the index of the last element that may be copied. (Both indices are inclusive.) The stride value st[i] determines which elements are actually copied.

The cmaxis argument is an array of axis numbers, and allows the axes to be reordered in the process of the copy. If cmaxis[i] = j, then axis j is the i'th fastest varying axis in column-major order.

Restrictions: These functions are not defined in the CM-2 version of the RTS.
2.11 General Communication Function (Send and Get)

The functions described in this section are used to perform multi-way point-to-point exchanges of data between processors.

2.11.1 CMCOM_send_[combiner, log.combiner]

```c
void CMCOM_send_[combiner] (dest, dest_inc, dest_geometry,
   send_address, send_address_inc,
   source, source_inc, mask, mask_inc,
   source_geometry, source_garbage_mask,
   many_collisions )

void CMCOM_send_[log_combiner] (dest, dest_inc, dest_geometry,
   send_address, send_address_inc,
   source, source_inc, mask, mask_inc,
   source_geometry, source_garbage_mask,
   element_size, many_collisions )
```

combiner = (Defined in Description section below)
log.combiner = (Defined in Description section below)

CMCOM_cm_address_t dest, send_address, source, mask;
int4 dest_inc, send_address_inc, source_inc, mask_inc;
CMCOM_machine_geometry_t dest_geometry, source_geometry;
CMCOM_cm_address_t source_garbage_mask;
int4 element_size;
log4 many_collisions;

**Description:** These functions send values from the source array to the dest array, according to the values of the send_address array. For each element of source, the corresponding value of the send_address array is the dest address to which the source value is to be sent. The mask argument controls which elements of the source array are sent, and hence implicitly controls which dest array elements are modified.

The source and dest arrays can have different geometries, as specified by the geometry arguments. The source garbage mask indicates which elements of the source are “genuine” data values.

The many_collisions argument selects whether a send strategy optimized for many collisions is used. (Note: This is not currently implemented, so this argument should be LOG_FALSE.)
When two or more values are sent to the same address, the values are combined according to the specified combiner or log_combiner function, and stored in the dest array.

The defined combiner options are:

- **u, s, f, df, c, dc, du, ds_add** — Addition (unsigned, signed, single and double float and complex, double unsigned and signed)
- **u, s, f, df, c, dc, du, ds_product** — Product (unsigned, signed, single and double float and complex, double unsigned and signed)
- **u, s, f, df, du, ds_max** — Maximum (unsigned, signed, single and double float, double unsigned and signed)
- **u, s, f, df, du, ds_min** — Minimum (unsigned, signed, single and double float, double unsigned and signed)

The defined log_combiner options are:

- **overwrite** — Only the last value to arrive is retained
- **logior** — Logical IOR
- **logxor** — Logical XOR
- **logand** — Logical AND

Restrictions: The u_add, u_max, u_min, u_product, s_add, s_product, f_product, df_product, c_product, and dc_product operators are not defined in the CM-2 RTS.

The ds and du operators are only defined in the CM-5/VU version of the RTS.

Note: Currently, the CM-2 versions of u_add and u_max have an element_size argument, just like the overwrite and logical operators.
2.11.2 CMCOM_send_to_queue

```c
void CMCOM_send_to_queue (queue_data, queue_data_inc, queue_data_geometry,
    queue_count, queue_count_inc, queue_count_geometry,
    send_address, send_address_inc, source, source_inc,
    mask, mask_inc,
    source_geometry, source_garbage_mask, element_size,
    queue_axis_index, queue_max_elements )
```

```c
CMCOM_cm_address_t mask, queue_count, queue_data, send_address;
CMCOM_cm_address_t source, source_garbage_mask;
CMCOM_machine_geometry_t queue_count_geometry, queue_data_geometry;
CMCOM_machine_geometry_t source_geometry;
int4 element_size, mask_inc, queue_axis_index;
int4 queue_data_inc, queue_count_inc, queue_max_elements;
int4 send_address_inc, source_inc;
```

**Description:** This function sends values from the source array to the queue_data array, according to the values of the send_address array. Multiple values sent to the same location are "queued" along an axis of queue_data. The queue_count array is modified to contain a count of the number of elements written to each location.

The queue_axis_index indicates which axis of the queue arrays serves as the "queue," and the queue_max_elements argument sets an upper limit on the number of elements a single location can queue up.

For each element of source, the corresponding value of the send_address array is the dest address to which the source value is to be sent.

The mask argument controls which elements of the source array are sent, and hence implicitly controls which elements of queue_data and queue_count are modified. The source and queue arrays can have different geometries, as specified by the geometry arguments. The source garbage mask indicates which elements of the source are "genuine" data values.

**Restrictions:** This function is defined only in the CM-2 RTS.
2.11.3 CMCOM_get

```c
void CMCOM_get (dest, dest_inc, send_address, send_address_inc,
    mask, mask_inc, dest_geometry, dest_garbage_mask,
    source, source_inc, source_geometry, element_size,
    many_collisions)
    CMCOM_cm_address_t dest, dest_garbage_mask, mask;
    CMCOM_cm_address_t send_address, source;
    CMCOM_machine_geometry_t dest_geometry, source_geometry;
    int4 dest_inc, element_size, mask_inc, send_address_inc, source_inc;
    log4 many_collisions;
```

**Description:** This function "gets" values from the `source` array and stores them in the `dest` array, according to the values of the `send_address` array. For each `dest` element, the corresponding `send_address` array value is the address of the `source` array element to "get." The `mask` argument controls which elements of the `dest` are modified. The `dest_garbage_mask` argument is used to determine which elements of the `dest` geometry contain "real" data values.

The `many_collisions` argument selects whether a send strategy optimized for many collisions is used. (Note: This feature is not currently implemented — this argument should always be `LOG_FALSE`.)
2.12 Cross-Geometry Move Function

The function described in this section moves values between arrays with different geometries.

2.12.1 CMCOM_cross_geometry_move

```c
void CMCOM_cross_geometry_move(dest, dest_inc, dest_geometry, source, 
    source_inc, mask, mask_inc, source_geometry, 
    source_garbage_mask, axis_mapping, 
    source_axis_coords, dest_axis_coords, 
    element_size)
```

Description: This function copies values from the `source` to the `dest` array, which can have different geometries (including different ranks), but must have the same total size and element type. Additionally, the axes that are selected for copying (see `axis_mapping` below) must have the same length as the axes to which they are mapped.

(Essentially, you can think of this function as a way to "rotate" a multidimensional array or array subset, and store it in a `dest` array with the same basic shape but possibly different dimensions — a greater number of axes, for example. You can not use this function to perform arbitrary rearrangements of data — use `CMCOM_send` and `CMCOM_get` instead.)

Values are transferred between the two geometries according to the values of the `axis_mapping`, `source_axis_coords`, and `dest_axis_coords` arguments:

- `axis_mapping` defines the mapping of axes between source and dest geometries. If the nth element of `axis_mapping` is i, then the nth source array axis is mapped to the ith `dest` array axis. A value of `CMCOM_no_axis` indicates an unmapped axis (one that doesn't exist in the source or is not transferred to the dest).

- `source_axis_coords` and `dest_axis_coords` define, for unmapped axes, the source coordinate along that axis that is read from, and the dest coordinate that is written into. (Essentially, these two arguments define which "plane" of the source array is copied to the dest, and which plane of the dest array receives it.) To map a source axis to every coordinate along the dest axis, `dest_axis_coords` should be `CMCOM_no_axis` For all axes that have a mapping in `axis_mapping`, the value of `source_axis_coords` should be `CMCOM_no_axis`.

The `mask` argument controls which elements of the `source` array are sent to the destination.
2.13 Rotate/Shift Communication Functions

The functions described in this section are used to shift array data along one or more geometry axes.

2.13.1 CMCOM_shift{.vector}, CMCOM_rotate{.vector}

```c
void CMCOM_[shift,rotate](dest, dest_inc, source, source_inc, mask, mask_inc,
   geometry, garbage_mask, distance,
   axis, axis_length, element_size)
CMCOM_cm_address_t dest, source, mask, garbage_mask;
CMCOM_machine_geometry_t geometry;
int4 dest_inc, source_inc, mask_inc, axis, element_size;
int8 distance, axis_length;

void CMCOM_[shift,rotate]_vector(dest, dest_inc, source, source_inc,
   mask, mask_inc, distance, distance_inc,
   geometry, garbage_mask, axis,
   axis_length, element_size)
CMCOM_cm_address_t dest, source, mask, distance, garbage_mask;
CMCOM_machine_geometry_t geometry;
int4 dest_inc, source_inc, mask_inc, distance_inc, axis, element_size;
int8 axis_length;
```

Description: These functions shift (or rotate) source array elements along the specified axis of the source geometry, and store the shifted result in the dest array. source and dest must be in the same geometry, and may be identical, but not in any other way overlapping.

Each element of the source array is shifted (or rotated) the same distance along the specified axis. The distance argument is the relative distance along the axis from which each dest element receives a value. (That is, a positive distance shifts values downward toward the low end of the axis, and a negative distance shifts values upward toward the high end.)

The shift functions shift without wrapping at the edges of the grid — values shifted off the edge of the grid are simply discarded, and dest elements that do not receive a shifted value are undefined. Diagramatically:

```
CMCOM_shift: DEST[...,i,...] = SOURCE[...,i+distance,...] if it exists
```

The rotate functions rotate with wrapping, so that values rotated off one end of an axis of the grid are inserted at the other end of the axis. In other words, if distance is greater in magnitude that
extent, it is reduced to the \emph{modulo} of itself and the extent (thus rotation of distance -10 along an axis with extent 3 is equivalent to a rotation of -1).

Diagramatically;

\begin{verbatim}
CMCOM_rotate: DEST[... ,i,... ] = SOURCE[... , (i+distance) % extent,... ]
\end{verbatim}

The \texttt{axis.length} argument specifies the "length" of the source axis. (The actual length may be greater than this value — this argument is useful for shifting subsections of an array, and can legally have a value of 0, meaning "do no shifting".)

For \texttt{CMCOM_shift} and \texttt{CMCOM_rotate}, the distance argument is a constant, and each element of the source array is shifted (or rotated) the same distance along the specified axis.

For \texttt{CMCOM_shift_vector} and \texttt{CMCOM_rotate_vector}, the distance argument is an array, and each element of the source array can be shifted (or rotated) by a different distance along the specified axis. Any element in \texttt{dest} to which multiple values are moved will receive one of those values, arbitrarily and unpredictably.

The \texttt{garbage_mask} argument is currently ignored — the garbage mask array is not used to determine axis lengths.

Restrictions: source and dest must be in the same geometry, but may have different values for \texttt{source-inc} and \texttt{dest-inc}. The source and dest may be identical, but not in any other way overlapping.

\subsection{2.13.2 CMCOM_physical_rotate}

\begin{verbatim}
void CMCOM_physical_rotate(dest, dest_inc, source, source_inc, geometry,
axis_pe_distance, axis, element_size, element_count)
CMCOM_cm_address_t dest, source;
CMCOM_machine_geometry_t geometry;
int4 dest_inc, source_inc, axis_pe_distance, axis;
int4 element_size, element_count;
\end{verbatim}

Description: This function rotates the subgrids of the source array by a number of nodes equal to \texttt{axis_pe_distance} along the specified \texttt{axis} of the physical grid of the supplied \texttt{geometry}.

\texttt{element_count} is the number of array elements in the subgrid.
2.14 Scans/Reductions/Spreads

The functions described in this section are used to perform scanning (parallel prefix), reduction, and spreading operations on CM arrays.

2.14.1 CMCOM_scan_[combiner], CMCOM_scan_[log.combiner]

```c
void CMCOM_scan_[combiner](dest, dest_inc, source, source_inc,
 segment, segment_inc, mask, mask_inc,
geometry, garbage_mask, axis, direction,
segment_mode, inclusion)

void CMCOM_scan_[log.combiner](dest, dest_inc, source, source_inc,
 segment, segment_inc, mask, mask_inc,
geometry, garbage_mask, axis, direction,
segment_mode, inclusion, element_size)
```

CMCOM_cm_address_t dest, source, segment, mask, garbage_mask;
CMCOM_machine_geometry_t geometry;
int4 dest_inc, source_inc, segment_inc, mask_inc, axis, element_size;
CMCOM_direction_t direction;
CMCOM_segment_mode_t segment_mode;
CMCOM_inclusion_t inclusion;
combiner = s_add, s_product, s_max, s_min, f_add, f_product, f_max, f_min,
df_add, df_product, df_max, df_min,
(non-CM-2) u_add, u_product, u_min, u_max, c_product, dc_product,
(CM-5/VU only) ds_add, ds_max, ds_min, ds_product, du_add, du_max,
du_min, du_product
(CM-2 different) c_add, dc_add

log.combiner = copy, logand, logior, logxor

Description: These functions perform a scan (parallel prefix) operation along the specified axis of the source array, and store the results in the dest array. The elements along the axis are cumulatively combined with the indicated combiner operation.

The mask argument controls which elements of the source are included in the scan, and similarly which elements of dest are actually modified. (In arithmetic scans, an array element that is excluded by the mask is treated as if it contains the identity value for the scan operation.)
The direction, inclusion mode, and segmentation of the scan are controlled by the other arguments:

- **direction** is one of the values CMCOM\_upward or CMCOM\_downward, and controls whether the scan is performed in increasing or decreasing order of axis coordinates.

- **inclusion** is one of the values CMCOM\_exclusive or CMCOM\_inclusive, and controls whether the scan result stored in each element of the dest array excludes or includes the corresponding element of the source array. **Note:** in CMCOM\_exclusive scans, the scan result for the first element of each segment is undefined (unless otherwise noted below).

- **segment\_mode** controls the segmentation of the scan, and is one of:
  - CMCOM\_none (for no segments)
  - CMCOM\_segment\_bit (for unidirectional segments)
  - CMCOM\_start\_bit (for bidirectional segments)

- **segment** is an array of logical values that specifies the starting elements of segments, as follows:
  - If segment\_mode is CMCOM\_none, the segment argument is ignored.
  - If segment\_mode is CMCOM\_segment\_bit, then each LOG\_TRUE in the segment array defines the starting element of a segment that always runs in increasing axis coordinate order (in other words, the location of the segment is unaffected by the scan direction).
  - If segment\_mode is CMCOM\_start\_bit, then each LOG\_TRUE in the segment array defines the starting element of a segment that runs in the current scan direction. (Upwards for upward scans, downward for downward scans.)

**Note:** When the segment\_mode is CMCOM\_start\_bit, the mask argument also controls which elements of the segment argument can define segments — a segment element excluded by the mask cannot begin a segment even if the segment element’s value is LOG\_TRUE. When the segment\_mode is CMCOM\_segment\_bit, the segment argument is unaffected by the mask.

For arithmetic scans only:

When the inclusion is CMCOM\_exclusive, the value of the first element in each segment depends on the segment mode. When the segment\_mode is CMCOM\_start\_bit, the first element in a segment (which would otherwise be undefined) receives the scan result from all of the contributing elements in the previous segment, or the scan operation’s identity value if there is no previous segment or if none of the previous segment’s elements were selected. When the segment\_mode is CMCOM\_segment\_bit, the first element in a segment receives the scan operation’s identity value.
For copy scans only:

The scan result of a segment is the value of the source at the first element in the segment selected by the mask. If no elements in the segment are selected, the result is zero.

When the inclusion is CMCOM_exclusive, the value of the first element in each segment depends on the segment mode. When the segment_mode is CMCOM_start_bit, the first element in a segment (which would otherwise be undefined) receives the scan result from all of the contributing elements in the previous segment, or zero if there is no previous segment or if none of the previous segment's elements were selected. When the segment_mode is CMCOM_segment_bit, the first element in a segment receives a zero value.

Restrictions: The dest and source arrays are allowed to be identical.

Logical, count, and singleword (s,u,f) arguments must be word-aligned. Doubleword and complex arguments must be double-word aligned. For logical and copy operations, element_size must be a multiple of 4.

The mask argument must be word-aligned, and the mask_inc must be either a multiple of 4 or CMCOM_PACKED_MASK.

The garbage_mask argument must be word-aligned.

The ds and du operators are defined only in the CM-5/VU version of the RTS.

The segment argument must be word-aligned, and the segment_inc must be a multiple of 4.

The following scan combinators are not available in the CM-2 RTS version: u_add, u_product, u_min, u_max, c_product, and dc_product.

The following combinators have different arguments in the CM-2 version: c_add, dc_add (the CM-2 version of these operators is described below).

The ds and du operators are only defined in the CM-5/VU version of the RTS.
Effects of Scan Arguments:

For example, here is a table listing the effects of various combinations of the `direction`, `inclusion`, and `segment_mode` arguments on a one-dimensional addition scan.

**Note:** For the mask and segment arrays, a `LOG_TRUE` value is represented by a “1”, and a `LOG_FALSE` value by a “0”. Unmodified destination array elements are indicated by a “.”.

```c
CMCOM_scan_s_add(dest, dest_inc, source, source_inc,
    segment, segment_inc, mask, mask_inc,
    geometry, garbage_mask, axis, direction,
    segment_mode, inclusion)
```

| mask array: | 1 1 1 1 0 0 0 0 1 1 0 0 1 1 1 0 |
| segment array: | 0 0 1 0 0 0 1 0 0 0 0 0 1 0 0 |
| source array: | 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 |

<table>
<thead>
<tr>
<th>direction</th>
<th>inclusion</th>
<th>segment_mode</th>
<th>dest array values:</th>
</tr>
</thead>
<tbody>
<tr>
<td>upward</td>
<td>exclusive</td>
<td>none</td>
<td>0 1 2 3 . . . 4 5 . 6 7 8 .</td>
</tr>
<tr>
<td>downward</td>
<td>exclusive</td>
<td>none</td>
<td>8 7 6 5 . . . 4 3 . 2 1 0 .</td>
</tr>
<tr>
<td>upward</td>
<td>inclusive</td>
<td>none</td>
<td>1 2 3 4 . . . 5 6 . 7 8 9 .</td>
</tr>
<tr>
<td>downward</td>
<td>inclusive</td>
<td>none</td>
<td>9 8 7 6 . . . 5 4 . 3 2 1 .</td>
</tr>
<tr>
<td>upward</td>
<td>exclusive</td>
<td>segment_bit</td>
<td>0 1 0 1 . . . 0 1 . 2 0 1 .</td>
</tr>
<tr>
<td>downward</td>
<td>exclusive</td>
<td>segment_bit</td>
<td>1 0 1 0 . . . 2 1 . 0 1 0 .</td>
</tr>
<tr>
<td>upward</td>
<td>inclusive</td>
<td>segment_bit</td>
<td>1 2 1 2 . . . 1 2 . 3 1 2 .</td>
</tr>
<tr>
<td>downward</td>
<td>inclusive</td>
<td>segment_bit</td>
<td>2 1 2 1 . . . 3 2 . 1 2 1 .</td>
</tr>
<tr>
<td>upward</td>
<td>exclusive</td>
<td>start_bit</td>
<td>0 1 2 1 . . . 2 3 . 4 5 1 .</td>
</tr>
<tr>
<td>downward</td>
<td>exclusive</td>
<td>start_bit</td>
<td>2 1 5 4 . . . 3 2 . 1 1 0 .</td>
</tr>
<tr>
<td>upward</td>
<td>inclusive</td>
<td>start_bit</td>
<td>1 2 1 2 . . . 3 4 . 5 1 2 .</td>
</tr>
<tr>
<td>downward</td>
<td>inclusive</td>
<td>start_bit</td>
<td>3 2 1 5 . . . 4 3 . 2 1 1 .</td>
</tr>
</tbody>
</table>
```
2.14.2 CMCOM_scan_[c/dc]_add (CM-2 version)

```c
void CMCOM_scan_[c/dc]_add (dest, dest_inc, source, source_inc,
    segment, segment_inc, geometry, axis,
    direction, segment_mode, element_count)

    CMCOM_cm_address_t dest, source, segment;
    CMCOM_machine_geometry_t geometry;
    int4 dest_inc, source_inc, segment_inc, axis, element_count;
    CMCOM_direction_t direction;
    CMCOM_segment_mode_t segment_mode;

Description: These functions perform scanning (parallel prefix) operations along the specified axis of the source array, and store the results in the dest array. The mask argument controls which elements of the dest are modified.

Note: These are the CM-2 version of the complex add-scan functions. See the general description of scan functions above for detailed descriptions and restrictions.
2.14.3 CMCOM_on_chip_pre_scan_[combiner]

```c
void CMCOM_on_chip_pre_scan_u_[combiner](scan_results, scan_results_inc,
    scan_segments, scan_segments_inc,
    dest, dest_inc, source, source_inc,
    segment, segment_inc, geometry, axis,
    direction, segment_mode, inclusion)

combiner = s_add, s_product, s_min, s_max, f_add, f_product, f_min, f_max,
    df_add, df_product, df_min, df_max (non-CM-2 only) u_add, u_product, u_min,
    u_max, c_product, dc_product, (CM-2 different) c_add, dc_add

CMCOM_cm_address_t scan_results, scan_segments, dest, source, segment;
int4 scan_results_inc, scan_segments_inc, dest_inc, source_inc;
int4 segment_inc, axis;
CMCOM_machine_geometry_t geometry;
CMCOM_direction_t direction;
CMCOM_segment_mode_t segment_mode;
CMCOM_inclusion_t inclusion;
```

**Description:** The on-chip pre-scan functions perform the initial phase of a virtual scan. These functions write values into the supplied physical array arguments (scan_results and optionally scan_segments), which can then be passed to physical functions. The on-chip scans also write pre-scanned values into the dest array; these values are to be combined with the physical results in the post-scan phase. The physical arrays are expected to have a total number of elements that is equal to the orthogonal_subgrid_length of the geometry.

The on-chip pre-scans work very much like the virtual functions; all of the directions, segment modes, and inclusions are supported. The dest values are written considering only the in-processor subgrid. The scan_results array gets the scan result from the last segment along the axis in the subgrid. The scan_segments array gets the logior of the segment bits along the subgrid.
2.14.4 CMCOM_on_chip_pre_scan_copy (non-CM-2 only)

```c
void CMCOM_on_chip_pre_scan_copy(scan_results, scan_results_inc, scan_segments,
    scan_segments_inc, scan_mask, scan_mask_inc,
    dest, dest_inc, source, source_inc, segment,
    segment_inc, mask, mask_inc, geometry, axis,
    direction, segment_mode, inclusion,
    element_size)

CMCOM_cm_address_t scan_results, scan_segments, scan_mask, dest,
    source, segment, mask;
int scan_results_inc, scan_segments_inc, scan_mask_inc, dest_inc,
    source_inc, segment_inc, mask_inc;
CMCOM_machine_geometry_t geometry;
int axis, element_size;
CMCOM_direction_t direction;
CMCOM_segment_mode_t segment_mode;
CMCOM_inclusion_t inclusion;
```

**Description:** The on-chip pre-scan copy function takes an optional mask argument and writes to a physical `scan_mask` array when the mask is non-null. The mask is assumed to have been previously conditionalized against the array's garbage mask if necessary. The segment values are read regardless of the mask; they are assumed to already have been conditionalized against the mask if necessary. The result written to the `scan_mask` is the physical result of a segmented logior on-chip pre-scan on the mask value.

The on-chip pre-scan copy works otherwise like the virtual copy-scan function, again operating only on the in-processor subgrid.

**Note:** See the general description of on chip pre-scan functions above for more information and restrictions.

**Restrictions:** In the CM-5 Version of the RTS, cases where the mask argument is NULL or the argument `segment_mode` is CMCOM_none are not yet supported for on-chip copy-scans.

Also, the `pre,post_scan.copy` operators currently do not yet exist in the CM-5/VU version of the RTS.
2.14.5  CMCOM_on_chip_pre_scan_[log.combiner]

    void CMCOM_on_chip_pre_scan_[log.combiner](scan_results, scan_results_inc,
        scan_segments, scan_segments_inc,
        dest, dest_inc, source, source_inc,
        segment, segment_inc, geometry, axis,
        direction, segment_mode, inclusion,
        element_size)

log_combiner = logior, logand, logxor

CMCOM_cm_address_t scan_results, scan_segments, dest, source, segment;
CMCOM_machine_geometry_t geometry;
int4 scan_results_inc, scan_segments_inc, dest_inc;
int4 source_inc, segment_inc, axis, element_size;
CMCOM_direction_t direction;
CMCOM_segment_mode_t segment_mode;
CMCOM_inclusion_t inclusion;

Description: These functions perform the pre-scan operations for the corresponding scan operators. See the general description of on chip pre-scan functions above for more information and restrictions.

2.14.6  CMCOM_onchip_pre_scan_[c/dc].add (CM-2 only)

    void CMCOM_onchip_pre_scan_[c/dc].add (scan_results, scan_results_inc,
        scan_segments, scan_segments_inc,
        dest, dest_inc, source, source_inc,
        segment, segment_inc, geometry,
        axis, direction, segment_mode,
        inclusion)

CMCOM_cm_address_t scan_results, scan_segments, dest, source, segment;
CMCOM_machine_geometry_t geometry;
int4 scan_results_inc, scan_segments_inc, dest_inc, source_inc;
int4 segment_inc, axis;
CMCOM_direction_t direction;
CMCOM_segment_mode_t segment_mode;
CMCOM_inclusion_t inclusion;
{}
2.14.7 CMCOM_on_chip_post_scan_[combiner]

```c
void CMCOM_on_chip_post_scan_[combiner](dest, dest_inc, segment, segment_inc,
    scan_values, scan_values_inc, geometry,
    axis, direction, segment_mode, inclusion)

combiner =
    s_add, s_product, s_min, s_max, f_add, f_product, f_min, f_max,
    df_add, df_product, df_min, df_max
(non-CM-2 only) u_add, u_product, u_min, u_max, c_product, dc_product
(CM-2 different) c_add, dc_add
```

CMCOM_cm_address_t dest, segment, scan_values;
CMCOM_machine_geometry_t geometry;
int4 dest_inc, segment_inc, scan_values_inc, axis;
CMCOM_direction_t direction;
CMCOM_segment_mode_t segment_mode;
CMCOM_inclusion_t inclusion;

**Description:** The on-chip post-scan functions perform the final phase of a virtual scan. The `scan_values` array is the result of the physical scan. It is assumed to have a number of elements that is equal to the `orthogonal-subgrid-length` of the `geometry`. These functions combine the physical result with the `dest` along the `axis`, writing to the `dest`.

**Restrictions:** In start-bit exclusive or segment-bit downward mode, elements in the `dest` including and following those corresponding to `LOG_FALSE` segment values are not combined with the physical result. In other segmented modes, elements following those corresponding to `LOG_TRUE` segment values are not combined.
2.14.8 CMCOM_on_chip_post_scan_copy (non-CM-2 only)

```c
void CMCOM_on_chip_post_scan_copy(dest, dest_inc, segment, segment_inc, mask,
mask_inc, scan_values, scan_values_inc,
scan_masks, scan_masks_inc, geometry, axis,
direction, segment_mode, inclusion,
element_size)
CMCOM_cm_address_t scan_results, scan_segments, scan_mask,
dest, source, segment, mask;
int scan_results_inc, scan_segments_inc, scan_mask_inc,
dest_inc, source_inc, segment_inc, mask_inc;
CMCOM_machine_geometry_t geometry;
int axis, element_size;
CMCOM_direction_t direction;
CMCOM_segment_mode_t segment_mode;
CMCOM_inclusion_t inclusion;
```

**Description:** The post-scan copy function combines the physical result (scan_values) with dest along the specified axis by writing the physical result to the dest. Segmented modes work as described above, but no combining to the dest is done along an axis if the corresponding scan_mask value is LOG_FALSE.

The on-chip post-scan copy function takes an optional mask argument and uses it and the physical scan_mask array when the mask is non-null. The mask is assumed to have been conditionalized against the array's garbage mask if necessary.

The segment values are read regardless of the mask; they are assumed to already have been conditionalized against the mask if necessary.

**Note:** See the general description of on chip pre-scan functions above for more information and restrictions.

**Restrictions:** In the CM-5 Version of the RTS, cases where the mask is NULL or the segment_mode is CMCOM_none are not yet supported for on-chip copy-scans.

Also, the [pre,post].scan_copy operators currently do not yet exist in the CM-5/VU version of the RTS.
2.14.9 CMCOM_onchip_post_scan_[log_combiner]

```c
void CMCOM_onchip_post_scan_[log_combiner](dest, dest_inc, segment, 
    segment_inc, scan_values, 
    scan_values_inc, geometry, axis, 
    direction, segment_mode, 
    inclusion, element_size)
```

log_combiner = logior, logand, logxor

CMCOM_cm_address_t dest, segment, scan_values;
CMCOM_machine_geometry_t geometry;
int4 dest_inc, segment_inc, scan_values_inc, axis;
CMCOM_direction_t direction;
CMCOM_segment_mode_t segment_mode;
CMCOM_inclusion_t inclusion;
int4 element_size;

**Description:** These functions perform the post-scan operations for the corresponding scan operators. See the general description of on chip post-scan functions above for more information and restrictions.

2.14.10 CMCOM_onchip_post_scan_[c/dc]_add (CM-2 only)

```c
void CMCOM_onchip_post_scan_[c/dc]_add (dest, dest_inc, source, source_inc, 
    scan_values, scan_values_inc, segment, 
    segment_inc, geometry, axis, direction, 
    segment_mode, inclusion)
```

CMCOM_cm_address_t dest, source, scan_values, segment;
CMCOM_machine_geometry_t geometry;
int4 dest_inc, source_inc, scan_values_inc, segment_inc, axis;
CMCOM_direction_t direction;
CMCOM_segment_mode_t segment_mode;
CMCOM_inclusion_t inclusion;

**Description:** These functions are the CM-2 operators for complex add-scan post-scans. See the general description of on chip post-scan functions above for more information and restrictions.
2.14.11 CMCOM_physical_scan_[combiner]

```
void CMCOM_physical_scan_[combiner](dest, dest_inc, source, source_inc, 
    segment, segment_inc, geometry, axis, 
    direction, segment_mode, element_count)

combiner =
    s_add, s_product, s_max, s_min, f_add, f_product, f_max, f_min,
    df_add, df_product, df_max, df_min,
    (non-CM-2 only) u_add, u_product, u_min, u_max, c_add, c_product,
    dc_add, dc_product
    (CM-5/VU only) ds_add, ds_max, ds_min, ds_product, du_add, du_max,
    du_min, du_product
```

CMCOM_cm_address_t dest, source, segment;
int4 dest_inc, source_inc, segment_inc, axis;
CMCOM_machine_geometry_t geometry;
CMCOM_direction_t direction;
CMCOM_segment_mode_t segment_mode;
int4 element_count;

**Description:** The physical scan functions are designed to implement the physical phase of the virtual scan functions. The physical scan functions do exclusive, optionally segmented, scan operations along the specified axis of the physical grid for the specified geometry.

The segment, direction, and segment_mode are as described for the CMCOM_scan_[combiner] functions.

When the segment_mode is CMCOM_start_bit, the first element in a segment (which would otherwise be undefined) receives the scan result from all of the contributing elements in the previous segment, or the scan operation’s identity value if there is no previous segment or if none of the previous segment’s elements were selected. When the segment_mode is CMCOM_start_bit, the first element in a segment receives the scan operation’s identity value.

**Restrictions:** The ds and du operators are defined only in the CM-5/VU version of the RTS.
2.14.12 CMCOM_physical_scan_log_combiner

```c
void CMCOM_physical_scan_log_combiner(dest, dest_inc, source, source_inc,
                                      segment, segment_inc, geometry, axis,
                                      direction, segment_mode,
                                      element_size, element_count)

log_combiner = logior, logand, logxor
```

CMCOM_cm_address_t dest, source, segment;
CMCOM_machine_geometry_t geometry;
int4 dest_inc, source_inc, segment_inc, axis, element_size, element_count;
CMCOM_direction_t direction;
CMCOM_segment_mode_t segment_mode;

Description: The physical scan functions are designed to implement the physical phase of the virtual scan functions. The physical scan functions do exclusive, optionally segmented, scan operations along the specified axis of the physical grid for the specified geometry.

For more information, see the description for the CMCOM_physical_scan_log_combiner functions.

2.14.13 CMCOM_physical_scan_copy

```c
void CMCOM_physical_scan_copy(dest, dest_inc, mask_dest, mask_dest_inc,
                               source, source_inc, segment, segment_inc,
                               mask, mask_inc, geometry, axis, direction,
                               segment_mode, element_size, element_count)
```

CMCOM_cm_address_t dest, source, mask_dest, mask, segment;
CMCOM_machine_geometry_t geometry;
int4 dest_inc, source_inc, segment_inc, mask_dest_inc, mask_inc
int4 axis, element_size, element_count;
CMCOM_direction_t direction;
CMCOM_segment_mode_t segment_mode;

Description: The physical scan functions are designed to implement the physical phase of the virtual scan functions. The physical scan functions do exclusive, optionally segmented, scan operations along the specified axis of the physical grid for the specified geometry.

The mask_dest array receives the result of a physical logior segmented scan of the mask values. If the mask is NULL, mask_dest is neither used nor affected.
The scan result of a segment is the value of the source at the first element in the segment selected by the mask. If there are no such elements, the scan result is zero.

In start-bit mode, the first element in a segment receives the scan result from all of the contributing elements in the previous segment, or zero if there is no previous segment or no contributing elements in that segment.

For more information, see the description for the CMCOM_physical_scan_[combiner] functions.

Note: This function is an exception to the general rule about masking. The dest argument may be modified for elements that are not selected by the mask. However, only dest elements selected by the mask are defined.

Restrictions: In the CM-5 Version of the RTS, cases where the mask argument is NULL or the argument segment_mode is CMCOM_none are not yet supported for physical copy-scans.
2.14.14 CMCOM_reduce_ [combiner, log_combiner]

```c
void CMCOM_reduce_[combiner] (dest, dest_inc, source, source_inc, mask,
    mask_inc, geometry, garbage_mask,
    axis, coordinate)

void CMCOM_reduce_[log_combiner] (dest, dest_inc, source, source_inc, mask,
    mask_inc, geometry, garbage_mask, axis,
    coordinate, element_size)

CMCOM_cm_address_t dest, source, mask, garbage_mask;
CMCOM_machine_geometry_t geometry;
int4 dest_inc, source_inc, mask_inc, axis, element_size;
int8 coordinate;

combiner = c_add, dc_add, s_add, s_product, s_min, s_max, count,
    f_add, f_product, f_min, f_max, df_add, df_product, df_max, df_min,
    c_product, dc_product
(CM-5/VU only) ds_add, ds_max, ds_min, ds_product, du_add,
    du_max, du_min, du_product
(non-CM-2 only) u_add, u_max, u_min, u_product
log_combiner = logand, logior, logxor
```

**Description:** These functions perform reduction operations along the specified `source` array axis, and store the results in `dest`. (A reduction operation is similar to a scan, except that the scan result from the last node along the axis is delivered to the axis element specified by `coordinate`.)

The `mask` argument controls which elements of the `source` are included in the scan. However, it does not affect which elements of `dest` are actually modified. In other words, all `dest` elements receive a scanned value.

**Note:** The `count` operator sets each element of `dest` to the number of `source` words that are non-zero. The source should not be a packed array.

**Restrictions:** The `dest` and `source` arrays are allowed to be identical.

Logical, count, and singleword (s,u,f) arguments must be word-aligned. Doubleword and complex arguments must be double-word aligned. For logical and copy operations, `element_size` must be a multiple of 4.

The `mask` argument must be word-aligned, and the `mask_inc` must be either a multiple of 4 or `CMCOM_PACKED_MASK`. The `garbage_mask` argument must be word-aligned.

The `ds` and `du` operators are defined only in the CM-5/VU version of the RTS.
2.14.15 CMCOM_on_chip_reduce_[combiner, log.combiner]

```c
void CMCOM_on_chip_reduce_[combiner] (dest, dest_inc, source, source_inc, geometry, axis);
CMCOM_cm_address_t dest, source;
int4 dest_inc, source_inc, axis;
CMCOM_machine_geometry_t geometry;
```

```c
void CMCOM_on_chip_reduce_[log.combiner] (dest, dest_inc, source, source_inc, geometry, axis, element_size);
CMCOM_cm_address_t dest, source;
int4 dest_inc, source_inc, axis, element_size;
CMCOM_machine_geometry_t geometry;
```

`combiner = s_add, f_add, df_add, s_product, f_product, df_product, df_min, f_min, s_min, df_max, f_max, s_max, c_add, dc_add, c_product, count, dc_product`

(non-CM-2 only) `u_product, u_add, u_max, u_min`

(CM-5/VU only) `ds_add, ds_max, ds_min, ds_product, du_add, du_max, du_min, du_product`

`log.combiner = logior, logxor, logand`

**Description:** These functions handle the on-chip component of the corresponding reduction operators.

**Restrictions:** The `ds` and `du` operators are defined only in the CM-5/VU version of the RTS.

2.14.16 CMCOM_on_chip_reduce_delivery

```c
void CMCOM_on_chip_reduce_delivery (dest, dest_inc, mask, mask_inc, geometry, source, source_inc, axis, coordinate, element_size);
CMCOM_cm_address_t dest, mask, source;
CMCOM_machine_geometry_t geometry;
int4 dest_inc, mask_inc, source_inc, axis, element_size;
int8 coordinate;
```

**Description:** This function handles the on-chip component of reduction operators.
2.14.17 CMCOM\_physical\_reduce\_[\texttt{combiner}]

**CM-2 Argument List:**

```c
void CMCOM\_physical\_reduce\_[\texttt{combiner}] (dest, dest\_inc, source, source\_inc,
geometry, axis, element\_count)

combiner =
c\_product, dc\_product, c\_add, dc\_add, f\_add, f\_product, f\_min, f\_max,
s\_add, s\_product, u\_min, u\_max, s\_min, s\_max, df\_add, df\_product,
df\_min, df\_max
```

CMCOM\_cm\_address\_t dest, source;
int4 dest\_inc, source\_inc, axis, element\_count;
CMCOM\_machine\_geometry\_t geometry;

**CM-5/CMSIM Argument List:**

```c
void CMCOM\_physical\_reduce\_[\texttt{combiner}] (dest, dest\_inc, source, source\_inc,
geometry, axis, physical\_coordinate,
geometry, axis, element\_count)

combiner = (same as CM-2 list above, plus:)
(no CM-2 version) u\_product, u\_add
(CM-5/VU version only) count, ds\_add, ds\_max, ds\_min,
ds\_product, du\_add, du\_max, du\_min, du\_product
```

CMCOM\_cm\_address\_t dest;
int4 dest\_inc;
CMCOM\_cm\_address\_t source;
int4 source\_inc;
CMCOM\_machine\_geometry\_t geometry;
int4 axis;
int4 physical\_coordinate;
int4 element\_count;

**Description:** These functions handle the physical component of the corresponding reduction operators.

**Restrictions:** The count, ds and du operators are defined only in the CM-5/VU version of the RTS.
2.14.18  CMCOM_physical_reduce_[log_combiner]

CM-2 Argument List:

```c
void CMCOM_physical_reduce_[log_combiner] (dest, dest_inc, source, source_inc,
group, axis, element_size,
```

```c
log_combiner = logior, logand, logxor
```

```c
CMCOM_cm_address_t dest, source;
int4 dest_inc, source_inc, axis, element_size, element_count;
CMCOM_machine_geometry_t geometry;
```

CM-5/CMSIM Argument List:

```c
void CMCOM_physical_reduce_[log_combiner] (dest, dest_inc, source, source_inc,
group, axis, physical_coordinate,
```

```c
log_combiner = logior, logand, logxor
```

```c
CMCOM_cm_address_t dest;
int4 dest_inc;
CMCOM_cm_address_t source;
int4 source_inc;
CMCOM_machine_geometry_t geometry;
int4 axis;
int4 physical_coordinate;
int4 element_size;
int4 element_count;
```

**Description:** These functions handle the physical component of the corresponding reduction operators.
**CMCOM_spread_[combiner, log_combiner]**

```c
void CMCOM_spread_[combiner] (dest, dest_inc, source, source_inc, mask, mask_inc, geometry, garbage_mask, axis)
combiner =
    f_add, f_product, f_min, f_max, df_add, df_product, df_min, df_max,
    c_product, dc_product, s_add, s_product, s_min, s_max, c_add, dc_add
(CM-2, CM-5/VU only) count
(CMSIM,CM-5/VU only) u_add, u_max, u_min, u_product
(CM-5/VU only) ds_add, ds_max, ds_min, ds_product, du_add, du_max,
    du_min, du_product

void CMCOM_spread_[log_combiner] (dest, dest_inc, source, source_inc, mask, mask_inc, geometry, garbage_mask, axis, element_size)
log_combiner = logior, logand, logxor
```

CMCOM_cm_address_t dest, garbage_mask, mask, source;
CMCOM_machine_geometry_t geometry;
int4 axis, dest_inc, element_size, mask_inc, source_inc;

**Description:** These functions perform spread operations along the specified axis of the source array, and store the results in the dest array. The mask argument controls which elements of the dest are modified.

(A spread operation is similar to a scan, except that the scan result from the last node along the axis is delivered to all nodes.)

**Note:** The count operator sets each element of dest to the number of source words that are non-zero. The source should not be a packed array.

**Restrictions:** The dest and source arrays are allowed to be identical.

Logical, count, and singleword (s,u,f) arguments must be word-aligned. Doubleword and complex arguments must be double-word aligned. For logical and copy operations, element_size must be a multiple of 4.

The mask argument must be word-aligned, and the mask_inc must be either a multiple of 4 or CMCOM_PACKED_MASK.

The garbage_mask argument must be word-aligned.

The ds and du operators are defined only in the CM-5/VU version of the RTS.
2.14.20 CMCOM_spread_copy

```c
void CMCOM_spread_copy (dest, dest_inc, source, source_inc, mask, mask_inc,
    geometry, garbage_mask, axis,
    coordinate, element_size)

CMCOM_cm_address_t dest, source, mask, garbage_mask;
CMCOM_machine_geometry_t geometry;
int4 dest_inc, source_inc, mask_inc, axis, element_size;
int8 coordinate;
```

**Description:** This function performs a copy-spread operation along the specified **axis** of the **source** array, and stores the results in the **dest** array. The **mask** argument controls which elements of the **dest** are modified.

2.14.21 CMCOM_multispread_copy

```c
void CMCOM_multispread_copy (dest, dest_inc, source, source_inc,
    mask, mask_inc, geometry, garbage_mask,
    axes_mask, coordinates, element_size)

CMCOM_cm_address_t dest, source, mask, garbage_mask;
CMCOM_machine_geometry_t geometry;
int4 dest_inc, source_inc, mask_inc, element_size;
int8 *coordinates;
unsigned4 axes_mask;
```

**Description:** This function performs a copy-spread operation along multiple axes of the **source** array, and stores the results in the **dest** array. The **mask** argument controls which elements of the **dest** are modified.

**Note:** The **coordinates** argument is an array of coordinate indices, indicating the element from which values are to be copied. The **axes_mask** argument is an unsigned integer, with a 1 bit corresponding to each axis along which values are to be spread.
2.14.22 CMCOM_on_chip_spread_copy_gather

    void CMCOM_on_chip_spread_copy_gather (dest, dest_inc, source, source_inc,
        geometry, axis, coordinate,
        element_size)

    CMCOM_cm_address_t dest, source;
    int4 dest_inc, source_inc, axis, coordinate, element_size;
    CMCOM_machine_geometry_t geometry;

Description: This function handles the on-chip component of spread operations.

2.14.23 CMCOM_on_chip_spread_delivery

    void CMCOM_on_chip_spread_delivery (dest, dest_inc, mask, mask_inc, geometry,
        source, source_inc, axis, element_size)

    CMCOM_cm_address_t dest, mask, source;
    CMCOM_machine_geometry_t geometry;
    int4 dest_inc, mask_inc, source_inc, axis, element_size;

Description: This function handles the on-chip component of spread operations.

2.14.24 CMCOM_physical_spread_[combiner]

    void CMCOM_physical_spread_[combiner] (dest, dest_inc, source, source_inc,
        geometry, axis, element_count)

    combiner = (CM-5/VU only) count, ds_add, ds_max, ds_min, ds_product,
        du_add, du_max, du_min, du_product, u_add, u_product
    (non-CMSIM) f_add, f_product, f_min, f_max, s_add, s_product,
        s_min, s_max, df_add, df_product, df_min, df_max,
        c_add, dc_add, dc_product, c_product
    (CM-2, CM-5/VU only) u_min, u_max

    CMCOM_cm_address_t dest, source;
    CMCOM_machine_geometry_t geometry;
    int4 dest_inc, source_inc, axis, element_count;

Description: These functions handle the physical component of the spread operations.

Restrictions: The ds and du operators are defined only in the CM-5/VU version of the RTS.
2.14.25 CMCOM_physical_spread_[log_combiner]

void CMCOM_physical_spread_[log_combiner] (dest, dest_inc, 
source, source_inc, 
geometry, axis, element_size, 
element_count)

log_combiner = (non-CMSIM) logior, logand, logxor

CMCOM_cm_address_t dest, source;
int4 dest_inc, source_inc;
CMCOM_machine_geometry_t geometry;
int4 axis, element_size, element_count;

Description: These functions handle the physical component of the corresponding spread operations.

2.14.26 CMCOM_physical_spread_copy

void CMCOM_physical_spread_copy(dest, dest_inc, source, source_inc, geometry, 
axis, physical_coordinate, element_size, 
element_count)

CMCOM_cm_address_t dest, source;
CMCOM_machine_geometry_t geometry;
int4 dest_inc, source_inc, axis, physical_coordinate;
int4 element_size, element_count;

Description: This function is part of the physical component of copy-spread operations.

2.14.27 CMCOM_physical_spread_copy_from_leader_always

void CMCOM_physical_spread_copy_from_leader_always 
(dest, dest_inc, source, source_inc, geometry, axis, coordinates, 
 element_size, element_count)
CMCOM_cm_address_t dest, source, coordinates;
CMCOM_machine_geometry_t geometry;
int4 dest_inc, source_inc, axis, element_size, element_count;

Description: This function is part of the physical component of copy-spread operations.
2.15 Array Reduction (Global) Functions

The functions described in this section are used to combine many values from a CM array into a single value, using one of several possible combining operations.

Note: While the combining operation is deterministic (and therefore repeatable), code should not depend on the order in which values are combined. For example, the combination order can affect whether or not overflow occurs. It can also result in loss of precision when large and small floating point numbers are combined.

The available combiners fall into two major categories: typed and logical. The four typed combiners (max, min, add, and product) have separate routines for each data type. The three logical combiners (logical "or", logical "exclusive or", and logical "and") operate on arbitrary 32-bit values.

2.15.1 CMCOM_global_[combiner]

[type] CMCOM_global_[combiner](source, source_inc, mask, mask_inc,
geometry, garbage_mask)
combiner = logior, logand, logxor, s_add, s_product, s_min, s_max, count,
f_add, f_product, f_min, f_max, df_add, df_product,
df_min, df_max, c_product, dc_product, c_add, dc_add
(non-CM-2 only) u_add, u_product, u_min, u_max
(CM-5/VU only) ds_add, ds_max, ds_min, ds_product, du_add, du_max,
du_min, du_product

CMCOM_cm_address_t source, mask, garbage_mask;
CMCOM_machine_geometry_t geometry;
int4 source_inc, mask_inc;

Description: These functions globally combine all the elements of the source CM array, using the indicated combiner operation, and return the combined value. The mask argument controls which elements of the source array are combined.

Restrictions: The u_add, u_product, u_min, and u_max versions are not defined in the CM-2 version of the RTS. The ds and du operators are defined only in the CM-5/VU version of the RTS.
2.15.2 CMCOM_on_chip_global_[combiner]

```c
void CMCOM_on_chip_global_[combiner] (dest, source, source_inc, mask,  
    mask_inc, garbage_mask, element_count)
    
    combiner = (CM-2 only) s_add, s_product, s_min, s_max, logior, logxor,  
        logand, count, f_add, f_product, f_min, f_max, df_add, df_product,  
        df_min, df_max, c_product, dc_product, c_add, dc_add
    
    CMCOM_cm_address_t dest, source, mask, garbage_mask;  
    int4 source_inc, mask_inc, element_count;
```

**Description:** These functions handle the on-chip part of the corresponding global_[combiner] operation, combining all values stored in the subgrid stored on each node. The dest array is assumed to be an array of the same type as the source with a subgrid size of 1 — that is, for each node only one subgrid element in the dest array is modified.

**Restrictions:** These operators are defined as functions only in the CM-2 RTS version. On the CM-5, the on-chip calculation is handled internally in the virtual global_[combiner] function.
2.15.3 CMCOM_physical_global_[combiner]

[type] CMCOM_physical_global_[combiner](source)
CMCOM_cm_address_t source;

combiner = logior, logand, logxor, s_add, s_product, s_max, s_min, f_add,
f_product, f_max, f_min, df_add, df_product, df_max, df_min,
c_product, dc_product, c_add, dc_add
(non-CM-2 only) u_add, u_product, u_min, u_max, count

type = unsigned4, int4, int8, real4, real8, cmpx8, cmpx16

Description: These functions handle the physical part of the corresponding global_[combiner]
operator, combining the values obtained for each CM node into a single result, which is returned.
The source argument is assumed to be an array of the specified type with a subgrid size of 1 —
that is, only a single value is retrieved from the subgrid by each node.
(It is also assumed that the values in this array have come from a call to the corresponding CM-
COM_on_chip_global_[combiner] function, but this is not a requirement.)
The legal combiner operations and corresponding returned types are:

u_add, u_product, u_min, u_max — unsigned4
logior, logand, logxor — unsigned4
s_add, s_product, s_max, s_min — int4
count — int8
f_add, f_product, f_max, f_min — real4
df_add, df_product, df_max, df_min — real8
c_product, c_add — cmpx8
dc_add, dc_product — cmpx16

Restrictions: The u_add, u_product, u_min, u_max, and count operators are not defined in
the CM-2 version of the RTS.
2.16 Rank (Sorting) Functions

The functions described in this section are used to perform simple numeric ranking and sorting of CM array elements.

2.16.1 CMCOM_[u,s,f,df,ds]_rank, CMCOM_[u,s,f,df,ds]_sort

void CMCOM_u_rank(dest, dest_inc, key, key_inc, segment, segment_inc, mask, mask_inc, geometry, garbage_mask, axis, direction, segment_mode, key_element_size)
void CMCOM_[s/f/df]_rank(dest, dest_inc, key, key_inc, segment, segment_inc, mask, mask_inc, geometry, garbage_mask, axis, direction, segment_mode)
void CMCOM_u_sort(dest, dest_inc, key, key_inc, segment, segment_inc, mask, mask_inc, geometry, garbage_mask, axis, direction, segment_mode)
void CMCOM_[s/f/df]_sort(dest, dest_inc, key, key_inc, segment, segment_inc, mask, mask_inc, geometry, garbage_mask, axis, direction, segment_mode)

CMCOM_cm_address_t dest, key, segment, mask, garbage_mask;
CMCOM_machine_geometry_t geometry;
int4 dest_inc, key_inc, segment_inc, mask_inc, axis, key_element_size;
CMCOM_direction_t direction;
CMCOM_segment_mode_t segment_mode;

Description: The rank functions determine the numerical ranking of the elements in the key array, and store that ranking in the dest array. The sort functions numerically sort the elements of the key array, and store the result in the dest array.

The dest and key array arguments may exactly overlap.

Restrictions: CMCOM_u_rank and CMCOM_u_sort treat keys as "big-endian" on the CM-5 (most significant word first) and "little-endian" on the CM-2 (least significant word first).

Currently, the CM5/VU RTS does not support segmented sort or rank. Also, only the CM5/VU RTS supports double-word signed integer sort and rank.

Performance: CMCOM_sort is equivalent to doing a CMCOM_rank and a send, but is faster in some cases.
Sorting is generally more efficient for big subgrids. The time per element for sort or rank improves as the subgrid size increases.

Sorting double-word keys (e.g. double floats) is approximately twice as slow as sorting single-word keys (e.g. floats or 4-byte integers). Sorting small integers (ranging from 0 to a value less than $2^{32}$) can be faster than sorting arbitrary integers.

Segmented and n-dimensional sorts and ranks can be significantly slower than 1-d unsegmented sorts and ranks.
2.17 CM-2 RTS Interface to Paris

The functions described in this section are used to perform low-level CM-2 operations such as translating between Paris fields and CM arrays.

Note: These functions are defined only in the CM-2 version of the RTS.

2.17.1 CMCOM_copy_[to/from]_paris_field

```c
void CMCOM_copy_to_paris_field(paris_field, source, source_inc, geometry, garbage_mask, element_size, paris_axis_mappings)

CM_field_id_t paris_field;
CMCOM_cm_address_t source;
int4 source_inc;
CMCOM_machine_geometry_t geometry;
CMCOM_cm_address_t garbage_mask;
int *paris_axis_mappings;

void CMCOM_copy_from_paris_field(dest, dest_inc, paris_field, geometry, garbage_mask, element_size, paris_axis_mappings)

CMCOM_cm_address_t dest;
int4 dest_inc;
CM_field_id_t paris_field;
CMCOM_machine_geometry_t geometry;
CMCOM_cm_address_t garbage_mask;
int *paris_axis_mappings;
```

Description: These functions copy array data from CMRTS array format to Paris field format. They are intended primarily to provide compatibility with useful Paris routines.

The `paris_axis_mappings` argument must be an array of integers that determines the correspondence of RTS array axes to Paris VP-set axes. (This argument functions much like the `axis_mapping` argument of `CMCOM_cross_geometry_move` — except that all axes must be mapped explicitly (there are no `CMCOM_no_axis` values here!).)

Restrictions: These functions are defined in the CM-2 RTS only.
3 The CMIP Interface

This section lists and describes the functions available in the CMIP software layer. The functions are listed by category, in the following order:

- Array copying functions
- Logical arithmetic functions
- Arithmetic functions
- Comparison instructions
- Comparison-preserving conversion functions
- Random number generation functions
- Indirect addressing functions
- Mask construction functions
- Scatter/gather functions
- CMIP utilities
- Send address operations

Note: While CMIP functions take machine arrays as arguments, these functions do not use geometries to determine the arrangement of data in the array. Instead, they typically take an element_size argument giving the number of bytes per element, and an element_count argument, giving the number of elements in the array subgrid.

Argument Overlap: Exact overlap of arguments (as described in Section 0.5.2) is permitted for any of the arithmetic and logical operations in the CMIP layer. For the remaining operations, the default rule still applies (prohibited unless otherwise noted).
3.1 CMIP Function Arguments

As described in the Concepts section, CMIP functions refer to arrays in terms of their actual memory locations. However, CMIP functions do all of their work “in-processor”, that is, without any communication between CM nodes. This means that each node only has to handle its own subgrid of the array. Since the elements of this subgrid are stored sequentially in the node’s memory, the information used to describe each array argument is different from that used in the CMCOM layer.

To be specific, a CMIP function refers to an array by the following two pieces of information:

- a CMCOM_cm_address_t value, which gives the array’s location in CM memory (or, equivalently, the location of the node’s subgrid)
- a subgrid increment, an integer giving the increment in words between successive subgrid elements

In addition, most CMIP functions have two additional arguments:

- an element count argument, which gives the total number of elements in the subgrid
- an element size argument, which gives the size of a single subgrid element in bytes

The element size and count arguments apply to all array arguments of the function. Together with the subgrid increment for each array argument, they define the size of each array’s subgrid, and the parts of that subgrid that can be affected by the CMIP operation.

Note: Any or all of the subgrid increment, element size, or element count arguments can differ from the actual increment, size or count of the array – this is useful in cases where you wish to apply a CMIP operation to only some elements of an array, or to only part of the data stored in each element. It is an error, however, to use this method to access memory addresses beyond the range permitted by the array’s size.

Also: Some CMIP functions omit the element size argument — these functions take the array increment to be the element size.

The arguments to CMIP functions can generally be divided into four groups:

1. CM array arguments (source, destination, mask, etc.) and their corresponding increments
2. the element size and element count arguments, which apply to all the array arguments
3. other values, such as an axis number, or coordinate
There are a number of "standard" arguments that most CMIP functions have, such as the source, destination, and mask array arguments. These arguments can be identified by their names:

- **source** (source1, source2, etc.) — The source array(s) from which values are read.
- **dest** (destarray, etc.) — The destination array into which the result is written.
- **mask** — The user mask argument (or NULL, to indicate that no masking is needed).
- **argument_inc** — Byte increment between adjacent elements in the argument array.
- **element_size** — The element size of the array subgrids.
- **element_count** — The number of elements in each array's subgrid.

These "standard" arguments are not described separately for each function; rather, only arguments unique to each function or unique meanings of the above arguments are described.

### 3.2 Array Copying Functions

The function described in this section is used to copy data from one array to another, possibly under control of a mask argument.

#### 3.2.1 CMIP_move

```c
void CMIP_move(dest, dest_inc, source, source_inc, mask, mask_inc,
               element_size, element_count)

CMCOM_cm_address_t dest, source, mask;
int4 dest_inc, source_inc, mask_inc, element_size, element_count;
```

**Description:** This function copies the **source** CM array to the **dest**, using the supplied **element_size** and **element_count** arguments to determine how large the arrays are. The **mask** argument determines which elements of the **dest** are modified.

**Note:** The **source** and **dest** arguments can have different increments — this is useful for copying only parts of array elements, or for performing other useful kinds of data shuffling.

The **CMIP_move** operator is highly optimized, and uses specialized algorithms for all cases of element size and alignment.

**Restrictions:** **source** and **dest** may be identical but may not in any other way overlap.
3.2.2 CMIP-{u,du}merge_with_constant

void CMIP_merge_with_constant (dest, dest_inc, source, source_inc, mask, mask_inc, garbage_mask, identity, element_size, element_count, invert)
CMCOM_cm_address_t dest, source, mask, garbage_mask;
int4 dest_inc, source_inc, mask_inc, element_size, element_count, invert;
int4 *identity;

void CMIP_u_merge_with_constant (dest, dest_inc, source, source_inc, mask, mask_inc, garbage_mask, identity, element_count, invert)
CMCOM_cm_address_t dest, source, mask, garbage_mask;
int4 dest_inc, source_inc, mask_inc, element_count, invert, identity;

void CMIP_du_merge_with_constant (dest, dest_inc, source, source_inc, mask, mask_inc, garbage_mask, identity, element_count, invert)
CMCOM_cm_address_t dest, source, mask, garbage_mask;
int4 dest_inc, source_inc, mask_inc, element_count, invert;
double identity;

Description: These functions move the elements of the source array to the dest array under control of the supplied mask argument. Where the mask is TRUE (non-zero), a value from source is copied. Where the mask is FALSE (zero), the constant identity value is stored in dest instead. If the invert argument is given as a non-zero value, the merging of the source and the identity is reversed, i.e. source value will be moved in positions where the mask value is FALSE and the identity value will be moved where the mask is TRUE.

CMIP_merge_with_constant can be applied to arrays of arbitrary data type — the argument element_size determines the size of the subgrid elements, and identity is a pointer to the actual identity value.

The CMIP-{u,du}_merge_with_constant functions are similar to the function CMIP_merge_with_constant, except that they assume the array arguments are of a specific size (unsigned integer or double unsigned integer), and thus lack an element_size argument. In addition, for these typed merge functions the identity argument is the actual identity value itself (word or doubleword), not a pointer to that value.

Restrictions: The value NULL (zero) for mask is not allowed, i.e. a mask must be supplied. Also, CMIP_du_merge_with_constant is not defined in the CMSIM RTS version.
3.3 Logical Arithmetic Functions

The functions in this section perform logical arithmetic operations on their arguments.

3.3.1 CMIP_logop, CMIP_lognot

void CMIP_log{op}(dest, dest_inc, source1, source1_inc, source2, source2_inc, mask, mask_inc, element_size, element_count)
op = and, ior, xor

CMCOM_cm_address_t dest, source1, source2, mask;
int4 dest_inc, source1_inc, source2_inc, mask_inc;
int4 element_size, element_count;

void CMIP_lognot(dest, dest_inc, source, source_inc, mask, mask_inc, element_size, element_count)
CMCOM_cm_address_t dest, source, mask;
int4 dest_inc, source_inc, mask_inc, element_size, element_count;

Description: These functions calculate the logical AND, IOR, XOR, and NOT operations on the elements of the supplied CM array source argument(s). The results are stored in the elements of the dest array. The mask argument determines which elements of the dest are modified.
3.4 Arithmetic Functions

The functions in this section perform standard arithmetic operations on their arguments.

3.4.1 CMIP_[type]_[add,subtract,product,div,mod,truncate]

```c
void CMIP_[add_type]_add(dest, dest_inc, source1, source1_inc,
    source2, source2_inc, mask, mask_inc, element_count)
void CMIP_[sub_type]_subtract(dest, dest_inc, source1, source1_inc,
    source2, source2_inc, mask, mask_inc, element_count)
void CMIP_[prod_type]_product(dest, dest_inc, source1, source1_inc,
    source2, source2_inc, mask, mask_inc, element_count)
void CMIP_[div_type]_div(dest, dest_inc, source1, source1_inc,
    source2, source2_inc, mask, mask_inc, element_count)
void CMIP_s_[mod/truncate](dest, dest_inc, source1, source1_inc,
    source2, source2_inc, mask, mask_inc, element_count)
CMCOM_cm_address_t dest, source1, source2, mask;
int4 dest_inc, source1_inc, source2_inc, mask_inc, element_count;
```

add_type = u,ds,du,s,f,df,c,dc
sub_type = ds,du,s,f,df,c,dc
prod_type = u,ds,du,int8,s,f,df,c,dc
div_type = f,df

Description: These functions calculate standard arithmetic functions on the elements of the supplied CM array source arguments, and store the results in the supplied dest argument. The mask argument controls which elements of the dest array are modified.

Restrictions:

These operators are not defined in the CM-2 RTS version:
u_add, int8_product, f_div and df_div

These operators are defined only in the CM-5/VU RTS version:
ds_add, du_add, ds_subtract, du_subtract,
u_product, ds_product, and ds_product
3.4.2 CMIP_[type]_[max,min]

    void CMIP_[type]_[max,min](dest, dest_inc, source1, source1_inc, 
    source2, source2_inc, mask, mask_inc, 
    element_size, element_count)

type = u,s,f,du,ds,df

    CMCOM_cm_address_t dest, source1, source2, mask;
    int4 dest_inc, source1_inc, source2_inc, mask_inc;
    int4 element_size, element_count;

Description: These functions calculate the numeric minimum and maximum of the elements of 
the supplied CM array source arguments, and store the results in the dest array. The mask 
argument controls which elements of the dest are modified.

Restrictions: These functions are defined only in the CM-5/VU version of the RTS.

3.4.3 CMIP[s,f,df].abs

CM-2, CM-5/SPARC, CMSIM Argument List:

    void CMIP_f_abs(dest, dest_inc, source, source_inc, 
    mask, mask_inc, element_count)

    CMCOM_cm_address_t dest, source, mask;
    int4 dest_inc, source_inc, mask_inc, element_size, element_count;

CM-5/VU Argument List:

    void CMIP_f_abs(dest, dest_inc, source, source_inc, 
    mask, mask_inc, element_size, element_count)

    CMCOM_cm_address_t dest, source, mask;
    int4 dest_inc, source_inc, mask_inc, element_size, element_count;

Description: These functions calculate the numeric absolute value of the elements of the supplied 
CM array source arguments, and store the results in the dest array. The mask argument controls which elements of the dest are modified.

Restrictions: These functions are not defined in the CMSIM RTS version. 
The CM-5/VU version of the function has an additional element_size argument.
3.4.4 CMIP\textunderscore \text{f,df\_sqrt}

```c
void CMIP_{f,df\_sqrt}(dest, dest_inc, source, source_inc,
mask, mask_inc, element_count)
CMCOM\_cm\_address\_t dest, source, mask;
int4 dest_inc, source_inc, mask_inc, element_count;
```

**Description:** These functions calculate the square root of the elements of the supplied CM array `source` arguments, and store the results in the `dest` array. The `mask` argument controls which elements of the `dest` are modified.

**Restrictions:** These functions are not defined in the CMSIM RTS version.

3.4.5 CMIP\textunderscore \text{u\_shift\_constant}

**CM-2, CM-5 Argument List:**

```c
void CMIP_u\_shift\_constant(dest, dest_inc, source, source_inc,
mask, mask_inc, amount, element_count)
CMCOM\_cm\_address\_t dest, source, mask;
int4 dest_inc, source_inc, mask_inc, amount, element_count;
```

**CMSIM Argument List:**

```c
void CMIP_u\_shift\_constant(dest, dest_inc, source, source_inc,
mask, mask_inc, amount, element_count)
int4 *dest, dest_inc, *source, source_inc, *mask, mask_inc;
int4 amount, element_count;
```

**Description:** This function performs an unsigned arithmetic shift on the elements of the supplied CM array `source` arguments, and store the results in the `dest` array. The `mask` argument controls which elements of the `dest` are modified.
3.5 Comparison Instructions

The functions in this section perform standard numeric comparisons on their arguments.

3.5.1 CMIP\_[u,f,df,du]\_eq, CMIP\_[u,s]\_lt

```c
void CMIP\_[u,f,df,du]\_eq(dest, dest_inc, source1, source1_inc,
    source2, source2_inc, mask, mask_inc, element_count)
void CMIP\_[u,s]\_lt(dest, dest_inc, source1, source1_inc,
    source2, source2_inc, mask, mask_inc, element_count)
```

**Description:** These functions perform standard comparison functions on the corresponding elements of the CM array `source` arguments, and store the results in the `dest` array, which must have an element type of log4. The `mask` argument determines which elements of the `dest` are modified.

**Restrictions:** The `f_eq` and `df_eq` comparisons are not defined in the CMSIM RTS version. The `du_eq` version is defined only in the CM-5/VU RTS version.

3.6 Comparison-Preserving Conversion Functions

The functions described in this section convert signed and floating-point values into unsigned values in a format that preserves comparison properties.

3.6.1 CMIP\_convert\_[s,f,df]\_[to,from]\_unsigned\_format

```c
void CMIP\_convert\_[s,f,df]\_[to,from]\_unsigned\_format
    (dest, dest_inc, source, source_inc, mask, mask_inc, element_count)
CMCOM\_cm\_address\_t dest, source, mask;
int4 dest\_inc, source\_inc, mask\_inc, element\_count;
```

**Description:** These functions convert signed and floating-point values into unsigned values in a manner that is efficient and that preserves the relative magnitude of the values (so that, for example, less-than or greater-than operations will still return the correct results).

**Note:** These functions are intended only to permit comparisons of signed and floating-point values using more efficient unsigned operators, and cannot be used to perform actual conversion of data.
3.7 Random Number Generation Functions

The functions described in this section implement the in-processor portion of random number generation.

3.7.1 CMIP_random_[df_to_s,df_to_ss]

```c
void CMIP_random_df_to_s(dest_addr, dest_inc, source_addr, source_inc, mask_addr, mask_inc, count)
void CMIP_random_df_to_ss(dest_addr, dest_inc, source_addr, source_inc, mask_addr, mask_inc, count)
CMCOM_cm_address_t dest_addr, source_addr, mask_addr;
int4 dest_inc, source_inc, mask_inc, count;
```

**Description:** These functions perform random number calculations on each node of the CM, using the supplied dest and seed arguments. The mask argument controls which elements of the dest argument are modified.

**Restrictions:** These functions are defined only in the CM-2 RTS version.
3.8 Indirect Addressing Functions

The functions in this section implement lookup tables and indirect array referencing.

Note: Since these routines are designed to be called from Fortran, array indices are 1-based, not 0-based as with other RTS functions.

3.8.1 CMIP_aref_1d

CM-2, CM-5 Argument List:

```c
void CMIP_aref_1d (dest, source, index, element_size, element_count)
CMCOM_cm_address_t dest, source, index;
int4 element_size, element_count;
```

CMSIM Argument List:

```c
void CMIP_aref_1d(dest, source, index, element_size, element_count)
int4 *dest, *source, *index;
int4 element_size, element_count;
```

Description: This routine implements a FORTRAN-style indirect addressing mechanism that treats the subgrid of each array argument as either a vector or a table of vectors.
The `dest` and `index` arguments are treated as vectors of length `element_count`, and `source` is treated as a table of `n` vectors of the same length, where `n` is as large as the `source` subgrid size will permit. Thus, the node-level view of this function’s operation is as shown above.

For each element `i`, `dest[i] = source[row i, col index[i]]`.

**Note:** Indexing is 1-based, as in FORTRAN, not zero-based.

For example:

<table>
<thead>
<tr>
<th>INDEX</th>
<th>DEST</th>
<th>SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>+-----+----+---------------------+----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>3</td>
<td>&lt;---</td>
<td>+---------------------+</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>+---------------------+</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>+---------------------+</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>+---------------------+</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>+---------------------+</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>+---------------------+</td>
</tr>
</tbody>
</table>

Because the table is column-ordered, the elements of the various arguments are located in memory as follows:

- `dest[i]` is at `dest + (4*i)` bytes
- `index[i]` is at `index + (01)` bytes
- `source[row, col]` is at `source + (columnsize * row) + (col * 4)` bytes
  - where `columnsize` is `(i * 4 * element_count)`

**Performance:** No optimizations for unaligned data are made. Performance is better with double aligned values in the vector and table arguments.
3.8.2 CMIP_aref_1d_any_order

```
CMIP_aref_1d_any_order(dest, dest_inc,
    source, source_row_inc, source_column_inc,
    index, index_inc, mask, mask_inc,
    garbage_mask, element_size, element_count)
CMCOM_cm_address_t dest, source, index, mask, garbage_mask;
int4 dest_inc, source_row_inc, source_column_inc;
int4 index_inc, mask_inc, element_size, element_count;
```

Description: This function is similar to CMIP_aref_1d. It implements a FORTRAN-style indirect addressing mechanism that treats the subgrid of each array argument as either a vector or a table of vectors.

The CMIP_aref_1d function assumes its array arguments have an increment, or stride, equal to their element size. The CMIP_aref_1d_any_order function allows you to specify the increments and element counts explicitly.

The node-level view of this function’s operation is identical to that given above for CMIP_aref_1d, with the exception of the formulas used for computing the location of array elements, and the addition of a mask array:

For each element i, if mask[i] then dest[i] = source[row i, col index[i]] Because the table is column-ordered, the array elements are located as follows:

```
dest[i] is at dest + (dest_inc * i)
index[i] is at index + (index_inc * i)
source[row,col] is at source + (source_column_inc * (col-1)) +
    (source_row_inc * row)
```

Restrictions: This function exists only in the CM-5/SPARC and CM-5/VU versions of the RTS.

3.8.3 CMIP_aset_1d

CM-2 Argument List:

```
void CMIP_aset_1d(array, source, index, mask, element_size,
    array_element_count, element_count)
CMCOM_cm_address_t array, source, index, mask;
int element_size, array_element_count, element_count;
```
CM-5/SPARC Argument List:

```c
void CMIP_aset ld(array, source, index, mask,
                  element_size, element_count)
CMCOM_cm_address_t array, source, index, mask;
int4 element_size, element_count;
```

CM-5/VU Argument List:

```c
void CMIP_aset ld (array, source, index, mask, mask_inc,
                   element_size, element_count)
CMCOM_cm_address_t array, source, index, mask;
int4 mask_inc, element_size, element_count;
```

CMSIM Argument List:

```c
void CMIP_aset ld(dest, source, index, mask,
                  element_size, element_count)
int4 *dest, *source, *index, *mask;
int4 element_size, element_count;
```

Description: This routine implements a FORTRAN-style indirect addressing mechanism that treats the subgrid of each array argument as either a vector or a table of vectors. The `index`, `source` and `mask` arguments are treated as vectors of length `element_count`, and `dest` is treated as a table of \( n \) vectors of the same length, where \( n \) is as large as the `source` subgrid size will permit.
Thus, the node-level view of this function's operation is:

<table>
<thead>
<tr>
<th>INDEX</th>
<th>SOURCE</th>
<th>MASK</th>
<th>DEST (table)</th>
</tr>
</thead>
<tbody>
<tr>
<td>++++</td>
<td>++++</td>
<td>++++</td>
<td>+++++++------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>++++</td>
<td>storage</td>
<td>++++</td>
<td>+++++++------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>++++</td>
<td>order</td>
<td>++++</td>
<td>+++++++------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>++++</td>
<td></td>
<td>++++</td>
<td>+++++++------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For each element \( i \), if \( \text{mask}[i] \) is non-zero then \( \text{source[row i, col index[i]]} = \text{dest[i]} \)

Note: indexing is 1-based, as in FORTRAN, not zero-based.
Because the table is column-ordered, the argument elements are located in memory as follows:

source[i] is at source + (4*i) bytes
index[i] is at index + (4*i) bytes
dest[row,col] is at dest + (columnsize * row) + (i * 4) bytes
where columnsize is (i * 4 * element_count)

**Performance:** No optimizations for unaligned data are made. Performance is better with double aligned values in the vector and table arguments.

3.8.4 CMIP_aset_1d_any_order

    CMIP_aset_1d_any_order(source, source_inc, 
                          dest, dest_row_inc, dest_column_inc, 
                          index, index_inc, 
                          mask, mask_inc, 
                          garbage_mask, 
                          element_size, 
                          element_count)

    CMCOM_cm_address_t source, dest, index, mask, garbage_mask;
    int4 source_inc, dest_row_inc, dest_column_inc;
    int4 index_inc, mask_inc, element_size, element_count;

**Description:** This function is similar to CMIP_aset_1d. It implements a FORTRAN-style indirect addressing mechanism that treats the subgrid of each array argument as either a vector or a table of vectors.

The CMIP_aset_1d function assumes its array arguments have an increment, or stride, equal to their element size. The CMIP_aset_1d_any_order function allows you to specify the increments and element counts explicitly.

The node-level view of this function's operation is identical to that given above for CMIP_aset_1d, with the exception of the formulas used for computing the location of array elements, and the addition of a mask array:

For each element i, if mask[i] then dest[row i, col index[i]] = source[i] Because the table is column-ordered, the array elements are located as follows:

source[i] is at source + (source_inc * i)
index[i] is at index + (index_inc * i)
dest[row,col] is at dest + (dest_column_inc * col-1) + (dest_row_inc * row)
Restrictions: This function exists only in the CM-5/SPARC and CM-5/VU versions of the RTS.

3.8.5 CMIP_lookup_in_table

CM-2, CM-5 Argument List:

```c
void CMIP_lookup_in_table (dest, source, index, element_size, element_count)
CMCOM_cm_address_t dest, source, index;
int4 element_size, element_count;
```

CMSIM Argument List:

```c
void CMIP_lookup_in_table(dest, source, index, element_size, element_count)  
int4 *dest, *source, *index;
int4 element_size, element_count;
```

Description: This function treats the subgrid of the supplied source array argument as a vector of values of length element_count, and uses the index array's values to select elements from this table to be stored in the dest array.

The mask argument controls which elements of the dest array are modified.

For each element i, dest[i] = source[index[i]]

Restrictions: The element_size value must be a multiple of 4, and the increments for source and dest are assumed to be element_size.

Performance: No optimizations for unaligned data are made. Performance is better with double aligned values in the array arguments.
3.9 Mask Construction Functions

The functions in this section pack and unpack the elements of garbage masks.

3.9.1 CMIP_pack_mask

```c
void CMIP_pack_mask(dest, sourcel, source1_inc, element_count)
CMCOM_cm_address_t dest, source1;
int4 source1_inc, element_count;
```

**Description:** This function converts the supplied `sourcel` user mask array into a garbage mask by "packing" its elements into garbage-mask format, and storing the packed result into the `dest` array argument.

**Restrictions:** This function is defined only in the CM-5/VU RTS versions.

3.9.2 CMIP_unpack_mask{.conditional}

```c
void CMIP_unpack_mask(dest, dest_inc, sourcel, element_count)
void CMIP_unpack_mask_conditional(dest, dest_inc, source1, mask, mask_inc, element_count)
CMCOM_cm_address_t dest, source1, mask;
int4 dest_inc, mask_inc, element_count;
```

**Description:** These functions convert the supplied `sourcel` garbage mask into a user mask array by "unpacking" its elements from garbage-mask format, and storing the unpacked result into the `dest` array argument.

The `{.conditional}` function includes a `mask` argument, which determines which elements of the `dest` are modified.

**Restrictions:** This function is defined only in the CM-5/VU RTS versions.
3.9.3 CMIP_compress_mask

```c
void CMIP_compress_mask (garbage_mask, src_garbage_mask,
                       user_mask, user_mask_inc, vp_ratio)
CMCOM_cm_address_t garbage_mask, src_garbage_mask, user_mask;
int4 user_mask_inc, vp_ratio;
```

**Description:** This function converts the user mask argument `user_mask` into a garbage mask by packing its elements into garbage mask form, then ANDing this compressed mask with the specified `src_garbage_mask` argument and finally storing the result into `garbage_mask`.

The element increment of the user mask is specified by `user_mask_inc`, and the element count of its subgrids is given by `vp_ratio`. (The name "vp-ratio" is a hold-over from CM-2 terminology, where it equivalently meant the number of elements stored per node.)

**Note:** The effect of this function is similar to that of the function `CMIP_pack_and_logand_mask`.

**Restrictions:** This function is not defined in the CM-2 or CMSIM RTS versions.

3.9.4 CMIP_pack_and_logand_mask

```c
void CMIP_pack_and_logand_mask(dest, source1, source1_inc,
                               source2, element_count)
CMCOM_cm_address_t dest, source1, source2;
int4 source1_inc, element_count;
```

**Description:** This function converts the user mask argument `source1` into a garbage mask by packing its elements into garbage mask form, then ANDing this compressed mask with the specified `source2` garbage mask argument and finally storing the result into `dest`.

The element increment of the user mask is specified by `source1_inc`, and the element count of its subgrids is given by `element_count`.

**Note:** This function is similar in effect to `CMIP_compress_mask`.

**Restrictions:** This function is defined only in the CM-5/VU RTS versions.
3.9.5 CMIP_make_triplet_mask

```c
void CMIP_make_triplet_mask (ctxt, mask, coord, lower, upper, stride, element_count)

CMCOM_cm_address_t ctxt;
CMCOM_cm_address_t mask;
CMCOM_cm_address_t coord;
int4 lower;
int4 upper;
int4 stride;
int4 element_count;
```

Description: This function defines a FORTRAN context mask, which is used to implement vector triplet subscripts of the form (lower:upper:stride).

The `coord` argument contains, for each of its array elements, the coordinate of that element along a given axis of the array (it is assumed that these coordinates have been calculated by some other function that requires a geometry argument, hence there is no geometry argument here).

The `lower`, `upper`, and `stride` arguments define a possibly strided set of coordinates along the same array axis that are to be selected by the context mask.

For each element of `coord`, the corresponding element of `ctxt` is determined as follows:

- If the `coord` value falls between `lower` and `upper` (inclusively) and is an exact multiple of `stride` greater than the `lower` bound, then the corresponding `ctxt` value is `LOG_TRUE`.
- Otherwise, the corresponding `ctxt` value is `LOG_FALSE`.

The user mask argument `mask` determines which elements of `ctxt` are actually modified — only `ctxt` elements selected by the `mask` are modified; all other `ctxt` elements are left unchanged.
3.10 Scatter/Gather Functions

The functions in this section implement scatter/gather operations.

3.10.1 CMIP_[scatter,gather]

```c
void CMIP_[scatter,gather] (dest, dest_inc, source, source_inc,
   offset, offset_inc, mask, mask_inc,
   element_size, element_count)
CMCOM_cm_address_t dest, source, offset, mask;
int4 dest_inc, source_inc, offset_inc, mask_inc;
int4 element_size, element_count;
```

**Description:** These functions perform masked scatter and gather operations. (These are inprocessor shufflings of array elements that involve no communication between nodes. A scatter operation “sends” array elements from the source to selected dest elements. A gather operation “gets” selected array elements from the source and stores them in dest.

For CMIP_scatter, each element in the source array is copied into dest at the location indexed by the corresponding offset value. The mask argument controls which elements of source are actually sent, and thus indirectly controls which elements of dest are modified.

For CMIP_gather, each element in the dest array receives a copy of the source element selected by the corresponding offset value. The mask argument directly controls which elements of dest are modified.

**Restrictions:** These functions are not defined in the CMSIM RTS version.
3.11 Send Address Operations

The functions described in this section implement send address conversions.

3.11.1 CMIP_[compress,expand].send_address

```c
void CMIP_[compress,expand].send_address(dest, dest_inc, source, source_inc,
                                         mask, mask_inc, element_count,
                                         onchip_bits)

CMCOM_cm_address_t dest, source, mask;
int4 dest_inc, source_inc, mask_inc, element_count;
unsigned4 onchip_bits;
```

Description: These functions convert between standard CM-5 8-byte send addresses and a "compressed" 4-byte format.
CMIP_compress_send_address compresses 8-byte send addresses into 4 bytes.
CMIP_expand_send_address expands compressed send addresses into 8 bytes.

Restrictions: These functions are defined only in the CM-5/VU version of the RTS.
The compress function assumes that all the significant bits in the send address will fit into the compressed format.
4 CMRTS Geometries

Both the CMRT and CMCOM layers of the RTS use geometry objects. These are data structures that define the size and shape of an array, as well as provide a description of how its elements are laid out in CM memory.

There are two different kinds of geometry objects, one used in the CMRT software layer, and one used in the CMCOM software layer. CMCOM geometry objects define the basic rank and shape of an array. CMRT geometry objects incorporate a CMCOM geometry and add information about the actual lengths of the array axes.

Note for CMF and C* Users: High-level CM languages use the CMRT layer to define parallel arrays, and thus use CMRT geometry objects to define the layout of data in those arrays.

This section presents an overview of array geometries and their properties, and then shows how the CMRT and CMCOM geometry-definition functions are used to define geometries with specific sets of properties. Performance observations and hints are also included.

Acknowledgement: Portions of this section, in particular the performance information, are derived from the original geometry manager document, written by David Gingold.

4.1 Array Geometries — Essential Concepts

The geometry object associated with a CM array does more than just specify the rank and axis lengths of the array. The geometry object also determines how the array’s elements are laid out in CM memory. This involves both specifying how the array elements are distributed among the CM processing elements, and how the array elements are laid out in each PE’s memory.

Here’s a simple example: Assuming that you have a two-dimensional array with axis lengths of (8,12), how do you store this array on a CM with four processing elements?

4.1.1 Array Subgrids and Physical Grids

It is standard in the RTS to divide an array into a number of equal-sized sections, called subgrids. These subgrids always have the same rank as the original array, and they have axis lengths that are either the same or shorter than those of the original array.

One subgrid is stored in the memory of each PE on the CM. This distribution of subgrids among the PEs defines a corresponding arrangement of processing elements, known as the physical grid. You can think of the physical grid as being an “array” of PEs, each containing a subgrid.

(remember that on a CM-5 with VUs, each VU is treated as a separate processing element – thus, there are four subgrids stored on a single processing node, one for each of the four banks of VU memory.)
For example, here's one way to divide an (8,12) array among four PEs. A subgrid size of (4,6) is used, resulting in a physical grid size of (2,2):

(Note: In this diagram, as in all others used in this section, the array axes are zero-based with coordinates increasing left to right and top to bottom. The numbers in the center of each subgrid represent the number of the processing element to which that subgrid has been assigned.)

This is just one of many possible ways of dividing an (8,12) array among four PEs. However, the possibilities are not unlimited. There are some constraints that limit the ways an array can be divided into subgrids:

- The number of available PEs is always a power of 2. It is legal to define an array geometry that does not use all the PEs, so long as the number of PEs used is still a power of 2. For example, the 4-PE CM in the above example permits arrays that use four PEs, two PEs, or only one PE.

- Each subgrid axis must evenly divide the corresponding array axis by a power of 2 — that is, each axis of the physical grid must have length that is a power of 2. (We'll see in Section 4.1.6 why these power-of-2 restrictions are important.)

- On the CM-2, there is an additional constraint imposed by the hardware — the number of elements in a subgrid must be a multiple of 4.

- On a CM-5 with vector units, there is a similar constraint — the number of elements in a subgrid must be a multiple of 8. (Note: this constraint will be removed in an upcoming version of the RTS.)
So for the (8,12) array example above, there are six possible subgrids, shown below. Each of these subgrid sizes corresponds to a different physical grid size, which is also listed for each case.

### Four-PE Grids:

<table>
<thead>
<tr>
<th>Subgrid Size: (4, 6)</th>
<th>Subgrid Size: (2, 12)</th>
<th>Subgrid Size: (8, 3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical Grid: (2, 2)</td>
<td>Physical Grid: (4, 1)</td>
<td>Physical Grid: (1, 4)</td>
</tr>
</tbody>
</table>

### Two-PE Grids:

<table>
<thead>
<tr>
<th>Subgrid Size: (4, 12)</th>
<th>Subgrid Size: (8, 6)</th>
<th>Subgrid Size: (8, 12)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical Grid: (2, 1)</td>
<td>Physical Grid: (1, 2)</td>
<td>Physical Grid: (1, 1)</td>
</tr>
</tbody>
</table>

### One-PE Grid:

<table>
<thead>
<tr>
<th>Subgrid Size: (8, 12)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical Grid: (1, 1)</td>
</tr>
</tbody>
</table>

#### 4.1.2 Efficient Subgrid Layout

Note that the last three examples above don’t use all the PEs — in particular, the last example uses only one PE to contain the entire array. This is typically not the best way to store an array in the CM. CM array operations execute most efficiently when neighboring array elements are stored in the memory of the same PE (that is, in the same subgrid). This allows many array operations to be handled by memory accesses, and reduces the amount of communication between PEs.

However, this doesn’t mean that all array elements should be stored in the subgrid on a single PE — this completely eliminates the advantage of parallelism provided by the CM processors. Instead, you should find a reasonable balance between large subgrids and an even distribution of array elements across the PEs.

By default, the RTS geometry definition functions always try to create geometries that use all available PEs and have as many neighboring elements within subgrids as possible. You can of course specifically direct the RTS geometry operators to create geometries with other distributions of array elements, if your application requires it.
4.1.3 Not All Arrays Are So Cooperative

An important point to note is that the array size (8,12) that we've been using in the examples so far is chosen specifically because it divides neatly into subgrids. However, there are some array sizes that don't divide up quite so neatly.

These arrays have a total number of elements that is:

- not divisible by the number of available PEs
- not divisible by any smaller power of 2.

For example, the elements of a two-dimensional array of size (7,11) cannot be divided in any way by a power of 2.

For the moment, we'll ignore these pathological arrays and assume that all arrays are of the nicely divisible variety. Once the methods for defining geometries have been explained, we'll see how to extend these methods to permit arrays of any size, including the pathological ones.

4.1.4 Processing Element Numbers and Axis Sequences

Once you've chosen the subgrid size, and thereby the physical grid size, the next geometry property you need to determine is the order of the PEs within the physical grid.

For all but one of the six (8,12) subgrid examples shown above, the order of PE numbers is uniquely determined — roughly speaking, the lowest-numbered PE is at the top or the left, the highest is at the bottom or right, with the PE numbers running sequentially in between.

The one exception is the (2,2) physical grid case, which can be numbered in either of two ways:

<table>
<thead>
<tr>
<th>Subgrid Size: (4, 6)</th>
<th>Subgrid Size: (4, 6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical Grid: (2, 2)</td>
<td>Physical Grid: (2, 2)</td>
</tr>
<tr>
<td>Axis Sequence: (0, 1)</td>
<td>Axis Sequence: (1, 0)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

For larger grids, this ambiguity is the rule rather than the exception, and it arises because of the way the PE numbers are arranged in the physical grid.
• PE numbers increase sequentially, from 0 to one less than the total number of PEs (the value of CMRT_PE_limit()).

• For any given geometry, there is a sequence of physical grid axes (the physical axis sequence) such that the PE numbers increase fastest (that is, by the smallest interval) along the first axis, then more slowly along each succeeding axis, increasing most slowly along the last one.

The (2,2) grid examples shown above correspond to axis sequences of (0,1) and (1,0) respectively, as indicated above each diagram. For physical grids of three, four, or more dimensions, the number of possible axis sequences increases geometrically.

A three-dimensional array, for example, has six possible physical axis sequences:

\[(0,1,2) \ (0,2,1) \ (1,0,2) \ (1,2,0) \ (2,0,1) \ (2,1,0)\]

Depending on your purposes, any one of these possible axis sequences may be as useful as any other. On the other hand, your particular application may make one of these axis sequences preferable to the others.

One reason that you might care about the sequence of physical axes is PE locality — how close together the processing elements are along specific array axes. PEs with adjacent numbers are usually close neighbors in the CM's communication grid. Thus, if you expect to be moving data along any one axis of an array more than any other, it's most efficient for that axis to be first in the physical axis sequence.

Note: This PE locality consideration becomes even more important on a CM-5 with vector units, where the first two axes in the sequence are much more efficient than the rest in terms of communication. We'll look at this case in detail in Section 4.4.

4.1.5 Array Element Addressing

The axis sequence for a given geometry is determined by the geometry's off-chip bits. To see where the term "off-chip" comes from, let's pause for a moment to look at the addressing of array elements.

When a particular array geometry is chosen, it implicitly assigns to each array element a unique send address, which identifies the element's location both in the array and in the CM hardware.

A send address has two components:

• A physical component, representing the PE in which the element is stored. (This is essentially the PE number, as described in the previous section.)

• A subgrid component, representing the element's position within its subgrid (that is, its offset in PE memory).
A send address can be represented as an unsigned integer of the form:

\[
\begin{array}{c}
\text{MSB} & \text{pppppppppppppppp...ppp} \\
\text{"off-chip" bits} & \text{ssssssssssssss...sss} \\
\text{LSB} & \text{LSB} \\
\text{"on-chip" bits} & \text{(physical)} \\
\text{(subgrid)} & \end{array}
\]

The physical component, or off-chip bits of a send address can range over all possible PE numbers (for example, 0,1,2 or 3 for a 4-PE CM). However, since the number of available PEs is usually constant, the physical component has a fixed length in bits, determined by the number of PEs. (If a geometry happens to use fewer than the total number of PEs, the most significant physical bits are always 0.)

The subgrid component, or on-chip bits of a send address can range from 0 to one less than the number of elements in a single subgrid. This subgrid size is determined by the size and shape of the subgrids in use. Since the size of the subgrid is dependent only on the size specified by the geometry, the number of subgrid bits in a send address can vary from geometry to geometry.

**Implementation Note:** In actual RTS code, the way send addresses are stored depends on the CM hardware version:

- **CM-2:** Send addresses are stored as a 32-bit integer — 12 bits for the physical component, and 20 bits for the subgrid component.
- **CM-5:** Send addresses are stored as two 32-bit integers, one for the physical component, one for the subgrid component.

**Note for CM-5 Users:** The terms “off-chip” and “on-chip” are a historical anachronism, and as such they can become confusing when applied to the CM-5 with VUs, where multiple PEs are located on the same chip. However, the terms are too prevalent to be totally abandoned, so just remember them this way: “off-chip” means outside of PE memory, and “on-chip” means within PE memory. This definition applies without qualification to all CM models.

**4.1.6 Off-Chip Bits and Axis Sequences**

Now we’re in a position to explain how you can use off-chip bits to specify the axis sequence of the physical grid for a given geometry.

Since PE numbers always increase steadily along physical grid axes, and since the physical axes themselves are by definition a power of 2 in length, the physical bits of a send address can be neatly split into regions that correspond to the individual axes. These regions of bits are known as the off-chip bits for each axis.
For example, in the case of a three-dimensional array with a physical grid size of (8, 16, 4) and a physical axis sequence of (0, 1, 2), the physical component of each send address would be 9 bits long, and divided into off-chip bits for each axis as follows:

<table>
<thead>
<tr>
<th>Axis</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axis Length:</td>
<td>$4=2^2$</td>
<td>$16=2^4$</td>
<td>$8=2^3$</td>
</tr>
<tr>
<td>Off-Chip Bits:</td>
<td>2</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Position:</td>
<td>7</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Send Address:</td>
<td>pp</td>
<td>pppp</td>
<td>ppp sssss....</td>
</tr>
</tbody>
</table>

The least significant set of off-chip bits represents the first axis in the physical axis sequence (the physical grid axis along which PE numbers increase the fastest). Similarly, the most-significant set of off-chip bits represents the last axis in the axis sequence (the grid axis along which the PE numbers vary the slowest).

**Note for CM-5/VU users:** The two lowest off-chip bits have special importance on a CM-5 with vector units. For more information, see Section 4.4.

So how is the axis sequence actually determined? Which axes actually correspond to which sets of off-chip bits? The answer is that it's up to YOU! When you define a geometry, you get to specify the set of off-chip bits assigned to each physical grid axis. The only restrictions are:

- The off-chip bits assigned to each axis must be contiguous. (See the note below.)
- There must be no gaps between the off-chip bits assigned to different axes.
- The least significant off-chip bits region must include the least significant physical bit.

**Note:** One of the RTS geometry definition functions lets you use bit-masks to specify the off-chip bits for each axis. In this case, the contiguity restriction is removed, but the other restrictions still hold. We'll look at this in more detail in Section 4.3, but for now we'll assume that off-chip bits are always contiguous for all axes.

The assignment of off-chip bits to axes is stored in the geometry object as two separate fields:

- The **off-chip length** field — for each axis, the number of physical bits assigned as off-chip bits.
- The **off-chip position** field — for each axis, the position of the first physical bit included in the field, counting up from bit 0, the least significant physical bit.

The off-chip length and position fields for each axis must define a contiguous range of bits, with no intervening gaps, and the lowest off-chip position must be 0 (the least significant physical bit).
For example, here again are the seven possible ways of dividing our original (8,12) array example into subgrids, along with the number of off-chip bits assigned to each axis, and the starting bit of the off-chip bit region assigned to each axis:

**Four-PE Grids:**

<table>
<thead>
<tr>
<th>Subgrid Size:</th>
<th>(4, 6)</th>
<th>Subgrid Size:</th>
<th>(4, 6)</th>
<th>Subgrid Size:</th>
<th>(2, 12)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Off-Chip Length:</td>
<td>(1, 1)</td>
<td>Off-Chip Length:</td>
<td>(1, 1)</td>
<td>Off-Chip Length:</td>
<td>(2, 0)</td>
</tr>
<tr>
<td>Off-Chip Position:</td>
<td>(0, 1)</td>
<td>Off-Chip Position:</td>
<td>(1, 0)</td>
<td>Off-Chip Position:</td>
<td>(0, 2)</td>
</tr>
</tbody>
</table>

```
0 2
1 3
```

```
0 1
2 3
```

```
0
1
```

**Two-PE Grids:**

<table>
<thead>
<tr>
<th>Subgrid Size:</th>
<th>(4, 12)</th>
<th>Subgrid Size:</th>
<th>(8, 6)</th>
<th>Subgrid Size:</th>
<th>(8, 12)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Off-Chip Length:</td>
<td>(1, 0)</td>
<td>Off-Chip Length:</td>
<td>(0, 1)</td>
<td>Off-Chip Length:</td>
<td>(0, 0)</td>
</tr>
<tr>
<td>Off-Chip Position:</td>
<td>(0, 1)</td>
<td>Off-Chip Position:</td>
<td>(1, 0)</td>
<td>Off-Chip Position:</td>
<td>(0, 0)</td>
</tr>
</tbody>
</table>

```
0
1
```

Notice that there’s no rule saying that the off-chip bits for a geometry have to occupy all the available physical bits. If there are any physical bits left over, this simply means that the geometry doesn’t use all the available PEs.
Additionally, it is legal for one or more axes of the physical grid to have no off-chip bits at all. An axis with no off-chip bits is represented entirely within PE memory — that is, the subgrid length along that axis is the same as the axis length of the array. Sequential elements along such an axis are not divided between PEs. (A special case of this type of axis is serial axes, described later, which are specifically guaranteed to have no off-chip bits.)

4.1.7 Send-Order and NEWS-Order Axes

The CM-2 adds one slight complication to this simple model: send-order and NEWS-order axes. The CM-2 communications hardware permits quick exchange of data along the hypercube axes of its communications network. This efficient form of data exchange is termed NEWS communication to distinguish it from the normal point-to-point routing operation of the CM-2's internal network, which is called send communication.

Because NEWS communication takes advantage of the hypercube structure of the CM-2, it is most efficient between PEs with PE numbers that differ by only a single bit (or in other words, PEs with numbers that differ exactly by a single power of 2). These PEs are, by definition, exactly one hypercube wire away from each other, and thus can exchange data very rapidly using NEWS communication.

The CM-2 RTS includes the option of "shuffling" the normal order of subgrids along any physical axis of a geometry, so that adjacent array subgrids are stored on PEs with numbers that differ by a power of 2. This shuffling is accomplished by Gray coding the normal order of subgrids along the axis. (For the Curious: Gray coding is described in more detail in the technical discussion at the end of this section.) An axis for which the subgrids have been shuffled in this manner is described as being in NEWS order. By contrast, the normal, unshuffled arrangement of subgrids is referred to as send order. Each axis in a geometry can be in either NEWS order or send order at your discretion — typically, you'll only use NEWS ordering for axes along which you expect to perform NEWS communication (that is, axes along which you expect to execute operations such as CMRT_news, CMRT_cshift, or CMRT_eoshift).

Note for CM-5 Users: Send ordering versus NEWS ordering is only an issue on the CM-2. The CM-5 uses the same communications hardware for all communication functions, and thus does not require NEWS ordering. All array axes are always send-ordered on the CM-5.

4.1.8 Subgrid Structure

The arrangement of subgrid elements in PE memory is similar to the arrangement of PEs in the physical grid. For any given geometry, there is a subgrid axis sequence such that the subgrid axis increment (the distance in memory between consecutive subgrid elements) is smallest for the first axis, larger for the next, and largest for the last axis.
The axis increment for the first axis is defined to be 1, and the increment for each succeeding axis is the product of the lengths of the preceding axes. So for a subgrid with axis lengths of (2,3,4), if the subgrid axis sequence is (0,1,2), then the subgrid axis increments are (1,2,6), and the elements of the subgrid are stored in memory in the following order (reading across first, then down):

\[
\begin{align*}
(0,0,0) & (1,0,0) (0,1,0) (1,1,0) (0,2,0) (1,2,0) \\
(0,0,1) & (1,0,1) (0,1,1) (1,1,1) (0,2,1) (1,2,1) \\
(0,0,2) & (1,0,2) (0,1,2) (1,1,2) (0,2,2) (1,2,2) \\
(0,0,3) & (1,0,3) (0,1,3) (1,1,3) (0,2,3) (1,2,3)
\end{align*}
\]

However, you don't have quite as much control over the arrangement of subgrid elements as you do with the physical grid axis sequence. The subgrid axis sequence is almost completely determined by axis number. By default, the axis sequence is arranged so that the indices of array elements vary in one of two ways:

- **Axis co-variant (row-major ordering, lowest axis varies fastest)** — The subgrid axis sequence is the same as the axis numbering.

- **Axis contra-variant (column-major ordering, highest axis varies fastest)** — The subgrid axis sequence is the opposite of the axis numbering (this is the default for high-level languages like C* and CM Fortran).

(Note: The terms axis co-variant and axis contra-variant were proposed by Guy Steele. See the discussion of the terms row-major and column-major in Section 4.10.1.)

All the RTS geometry-definition functions have an argument, `highest_axis_varies_fastest` that allows you to choose which of the two axis sequencing methods to use.

**Programming Note:** If you are using the RTS functions to manipulate arrays created by higher-level languages (like CM Fortran and C*), be sure to use the appropriate default axis sequence, as listed above, when creating arrays of your own.

**Implementation Note:** A future version of the RTS will allow selection of a more arbitrary subgrid axis sequence.
Serial Axes — Guaranteeing PE Locality

There is one other way that you can control the subgrid axis sequence: you can specify that one or more axes are *serial*. This has two effects. First, a serial axis is guaranteed to be represented entirely in PE memory — that is, the number of off-chip bits for the axis is zero. Second, the axis increment of a serial axis is guaranteed to be larger than the axis increment of any non-serial axis.

Thus, if any axes of an array are declared to be serial, the subgrid axis sequence is the same as defined above with the exception that all serial axes are shifted (in order) to the end of the sequence. This means that the axis increments of serial axes are always larger than the increments of non-serial axes. If two or more axes are declared serial, their axis increments are defined in order just as for non-serial axes.

One way to visualize this is to note that by this definition, non-serial axes are packed "tighter" in PE memory than serial axes. In other words, as you proceed through memory examining each of the elements of a subgrid, the subgrid indices of the elements always increase slower along serial axes than along non-serial axes. In algorithmic terms, all the non-serial indices of the elements must be incremented through a complete iteration before any of the serial indices are incremented once.

As an example, consider the three-dimensional subgrid example used above — axis lengths of (2,3,4), subgrid axis ordering (0,1,2). If axis 1 is declared to be serial, the order of subgrid elements in memory changes to the following (again reading across, then down):

\[
(0,0,0) (1,0,0) (0,0,1) (1,0,1) (0,0,2) (1,0,2) (0,0,3) (1,0,3) \\
(0,1,0) (1,1,0) (0,1,1) (1,1,1) (0,1,2) (1,1,2) (0,1,3) (1,1,3) \\
(0,2,0) (1,2,0) (0,2,1) (1,2,1) (0,2,2) (1,2,2) (0,2,3) (1,2,3)
\]

What's the rationale behind all this? Serial axes, by definition, are guaranteed to be represented entirely within PE memory, whereas non-serial axes may or may not cross between PEs. Thus, any array operation that moves data mostly or entirely along a serial axis will execute more efficiently than one that moves data along a non-serial axis.

Additionally, since non-serial axes are packed "tighter" in PE memory, you can use serial axis declaration to indirectly determine which array elements will be closer together in PE memory. In the example above, notice how all the subgrid elements at coordinate 0 of axis 1 are grouped together, as are all the elements at coordinates 1 and 2.

In effect, the serial axes of an array form a "sub-sub-grid," each element of which is a grid composed of non-serial axes. This means that the non-serial axes can together be treated as an array of smaller rank, and the serial axes can be used to "loop" over consecutive subarrays. The Fortran compiler, in particular, takes advantage of this property of serial arrays.
4.2 Handling Arrays of Arbitrary Size — Garbage Masks

Now it’s time to return to the pathological arrays that we set aside earlier — those arrays that don’t divide neatly into subgrids.

These arrays have a total number of elements that is:

- not divisible by the number of available PEs
- not divisible by any smaller power of 2.

A good example is an array of size (7,11), which cannot be divided into subgrids for any power-of-2 number of processing elements. To store these types of arrays using the methods described above, an extra layer of information is needed.

The strategy for these types of arrays is to increase the length of one or more of the array’s axes until the total number of array elements can be divided by a power of 2, then to define a corresponding geometry in CM memory. A companion array of logical values is defined at the same time, to indicate which memory locations actually contain real array values, and which contain unused, or garbage, values. This extra logical array is referred to as the garbage mask of the geometry.

In the RTS, the garbage mask is an array of packed binary values, stored as part of the geometry object. It contains one value for each array element defined by the geometry. Elements with a 1 in the garbage mask contain actual array data, and elements with a 0 in the garbage mask are garbage. (The garbage mask bits in each packed word are stored in order from least to most significant bit — that is, the first subgrid element is flagged by bit 0, the second by bit 1, etc.)

Note: Some functions, particularly those that operate along a single axis of an array, take an extent argument that gives the length of the axis, and thus do not need to check the garbage mask to determine which elements are garbage.

As an example, here’s a possible geometry for the (7,11) array mentioned above. The elements marked with an “x” are garbage values:
Garbage values generally form a fringe along the high ends of array axes, so that real array data can be indexed as if the garbage values did not exist. Garbage elements may be written to by RTS operations, but are never read.

Note: The RTS does not currently permit an array to have garbage elements interspersed with real array elements along any axis.

4.3 Geometries with Non-contiguous Off-chip Bits

A recent addition to the geometry-creation tools of the RTS is a geometry operator that allows you to use bit masks to specify the off-chip bits for each axis of a geometry. This means that the bits assigned to each axis do not have to be contiguous.

As an example, assume that we have a two-dimensional array, which we want to store in the memory of a 16-node CM-5. The geometry of this array requires four off-chip bits. Each of these bits can be assigned to either of the two axes, 0 and 1. The send-addresses for such a geometry would look like:

\[
\text{Send Address: } \quad \text{pppp ssss....}
\]

If we assume for simplicity that there is only one array element per PE, then there are sixteen possible PE arrangements.

The figure on the next page shows all sixteen of these cases, giving for each one the assignment of off-chip bits to axes 0 and 1.
Physical Bit Assignments
(Assumes 2D array, 16 nodes)

Each of the four off-chip bits can be assigned to axis 0 or axis 1:

```
  P P P P
```

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>0 0 1</td>
<td>0 0 1</td>
<td>0 0 1</td>
</tr>
<tr>
<td>0 1 0</td>
<td>0 1 1</td>
<td>0 1 1</td>
<td>0 1 1</td>
</tr>
<tr>
<td>1 0 0</td>
<td>1 0 0</td>
<td>1 0 0</td>
<td>1 0 0</td>
</tr>
<tr>
<td>1 1 0</td>
<td>1 1 0</td>
<td>1 1 0</td>
<td>1 1 0</td>
</tr>
</tbody>
</table>
```

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>0 0 1</td>
<td>0 0 1</td>
<td>0 0 1</td>
</tr>
<tr>
<td>0 1 0</td>
<td>0 1 1</td>
<td>0 1 1</td>
<td>0 1 1</td>
</tr>
<tr>
<td>1 0 0</td>
<td>1 0 0</td>
<td>1 0 0</td>
<td>1 0 0</td>
</tr>
<tr>
<td>1 1 0</td>
<td>1 1 0</td>
<td>1 1 0</td>
<td>1 1 0</td>
</tr>
</tbody>
</table>
```

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>0 0 1</td>
<td>0 0 1</td>
<td>0 0 1</td>
</tr>
<tr>
<td>0 1 0</td>
<td>0 1 1</td>
<td>0 1 1</td>
<td>0 1 1</td>
</tr>
<tr>
<td>1 0 0</td>
<td>1 0 0</td>
<td>1 0 0</td>
<td>1 0 0</td>
</tr>
<tr>
<td>1 1 0</td>
<td>1 1 0</td>
<td>1 1 0</td>
<td>1 1 0</td>
</tr>
</tbody>
</table>
```

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>0 0 1</td>
<td>0 0 1</td>
<td>0 0 1</td>
</tr>
<tr>
<td>0 1 0</td>
<td>0 1 1</td>
<td>0 1 1</td>
<td>0 1 1</td>
</tr>
<tr>
<td>1 0 0</td>
<td>1 0 0</td>
<td>1 0 0</td>
<td>1 0 0</td>
</tr>
<tr>
<td>1 1 0</td>
<td>1 1 0</td>
<td>1 1 0</td>
<td>1 1 0</td>
</tr>
</tbody>
</table>
```

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>0 0 1</td>
<td>0 0 1</td>
<td>0 0 1</td>
</tr>
<tr>
<td>0 1 0</td>
<td>0 1 1</td>
<td>0 1 1</td>
<td>0 1 1</td>
</tr>
<tr>
<td>1 0 0</td>
<td>1 0 0</td>
<td>1 0 0</td>
<td>1 0 0</td>
</tr>
<tr>
<td>1 1 0</td>
<td>1 1 0</td>
<td>1 1 0</td>
<td>1 1 0</td>
</tr>
</tbody>
</table>
```

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>0 0 1</td>
<td>0 0 1</td>
<td>0 0 1</td>
</tr>
<tr>
<td>0 1 0</td>
<td>0 1 1</td>
<td>0 1 1</td>
<td>0 1 1</td>
</tr>
<tr>
<td>1 0 0</td>
<td>1 0 0</td>
<td>1 0 0</td>
<td>1 0 0</td>
</tr>
<tr>
<td>1 1 0</td>
<td>1 1 0</td>
<td>1 1 0</td>
<td>1 1 0</td>
</tr>
</tbody>
</table>
```
4.4 Vector Unit Array Layout

The layout of arrays on a CM-5 with vector units is a little more involved. Taking advantage of the processing power of the VUs means paying close attention to the choice of axis order for a geometry and the assignment of off-chip bits to its axes.

4.4.1 Vector Unit Grids

Each processing node contains 4 vector units, and each of these VUs is assigned a subgrid of the array. This effectively gives you four times as many PEs or, equivalently, two more off-chip bits to use in defining your geometries. However, the arrangement of VU subgrids is constrained by the fact that each group of 4 VUs is located within a single processing node.

This means that you must divide your array up so that:

- The size of the subgrid assigned to each processing node is the same for all nodes
- Each node subgrid is further divisible into four VU grids that have the same size and shape

In other words, you parcel out an array into equal subgrids for the processing nodes, and then parcel out each subgrid into equal portions for the VUs.

For example, here’s the (8, 12) array example we’ve been using all along, subdivided into VU grids:

```

```

The VU grids within a subgrid can be laid out in any of a number of arrangements, much like the CM processing nodes in the physical grid.
For example:

**Four Sample VU Subgrid Divisions**

(Note that, for convenience, the last of the layouts shown above assumes a (6, 4) subgrid, instead of the (4, 6) subgrid shown in the other examples.)

There is a distinction, however, between the physical grid and VU grids: The VU grids must use all four VUs—you can't omit some of the VUs, the way you can omit some of the CM processing nodes in the physical grid.

4.4.2 Vector Unit Addressing

The addition of vector unit hardware to a CM-5 adds two new off-chip bits to the send-address model described previously. On a CM-5 with vector units, send addresses can be represented as:

\[
\text{MSB} \ \ ppppppppppp \ldots pppv \ \ sssssssss \ldots sss \ \ \text{LSB}
\]

"off-chip" bits "on-chip" bits

(physical) (subgrid)

The two least-significant off-chip bits represent the location of the vector units in the CM hardware. The "v" bit indicates one of the two vector unit chips on each processing node, and the "V" bit indicates one of the two VUs on that chip.

This means that a geometry axis that includes the "V" bit will have much higher communication performance than any other axis, because some or all of its operations are handled within a single VU chip.
When you're choosing a geometry for a CM-5 with VUs, therefore, the rule of thumb is to put the axis that requires the best performance first in the axis sequence, and try to limit it to only the two VU bits.

### 4.4.3 VU Bits and Axis Sequences

The addition of VU bits to send addresses complicates the simple picture of axis sequences and PE layout presented in previous sections. For example, since the VU bits are the lowest-order bits in the send address, the PE numbering must pass through all four VUs on a single node before advancing to the next node.

Thus, depending on how many VU bits (0, 1, or 2) an array axis includes in its off-chip bits, the number of VUs per node along that axis may be 1, 2 or 4.

As an example, assume that we have a two-dimensional array, which we want to store in the memory of a 4-node CM-5 with the standard 4 VUs per node. In this case, the geometry requires only four off-chip bits: two physical bits and two VU bits. Each of these bits can be assigned to either of the two axes, 0 and 1. The send-addresses for such a geometry would look like:

\[ \text{Send Address: } ppV \; ss \ldots \]

If we assume for simplicity that there is only one array element per VU, then there are four possible VU arrangements, and four possible processing node layouts, making a total of sixteen cases in all.

The figures on the next two pages shows all sixteen of these cases, giving for each one the assignment of off-chip bits to axes 0 and 1.
**Physical/VU Bit Assignments**  
(Assumes 2D array, 4 nodes, 4 VUs per node)

Each of the four off-chip bits can be assigned to axis 0 or axis 1:

```
  p  p  v  v
```

<table>
<thead>
<tr>
<th>Bit</th>
<th>Axis 0</th>
<th>Axis 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Node 0</td>
<td>Node 1</td>
</tr>
<tr>
<td>1</td>
<td>Node 1</td>
<td>Node 2</td>
</tr>
<tr>
<td>2</td>
<td>Node 2</td>
<td>Node 3</td>
</tr>
<tr>
<td>3</td>
<td>Node 3</td>
<td>Node 0</td>
</tr>
</tbody>
</table>

![Diagram showing bit assignments to nodes](image_url)
Physical/VU Bit Assignments, Cont.
(Assumes 2D array, 4 nodes, 4 VUs per node)

Each of the four off-chip bits can be assigned to axis 0 or axis 1:

```
 p p v v
```
4.5 Array Geometries — A Technical Description

Now, for those who like more exact specifications, here is a specific description of the relationship between CM array indices and the physical and subgrid components of array element send addresses.

4.5.1 Physical and Subgrid Coordinates

Once again, we’ll use the (8,12) array and 4-PE CM we’ve been using all along, only this time, we’ll answer a more specific question: Given the coordinates of an element in the (8,12) array, how can we determine the exact location (PE and memory offset) of the element in the CM’s memory?

When we choose a particular subgrid size to use for an array, we implicitly choose a physical grid size as well. Thus, selecting a subgrid size defines two separate kinds of coordinates that we can use to describe the location of an array element:

- The physical coordinates — the location in the physical grid
- The subgrid coordinates — the location within the subgrid

For example, if we choose a subgrid size of (4,6), the arrangement of subgrids and physical grid is:

(For the moment we’ll ignore the issue of PE numbering, and return to it later.)
If we choose a particular array element, (3,9) for example (marked with a "*" in the diagram above), its physical and subgrid coordinates can be determined for each axis as follows:

\[
\begin{align*}
\text{physical-coordinate[axis]} &= \text{coordinate[axis]} \div \text{subgrid-length[axis]} \\
\text{subgrid-coordinate[axis]} &= \text{coordinate[axis]} \mod \text{subgrid-length[axis]}
\end{align*}
\]

So element (3,9) of the array has the following physical and subgrid coordinate pairs:

\[
\begin{align*}
\text{physical-coordinates} &= (3 \div 4, 9 \div 6) = (0, 1) \\
\text{subgrid-coordinates} &= (3 \mod 4, 9 \mod 6) = (3, 3)
\end{align*}
\]

Once we have determined the physical and subgrid coordinates of an array element, we must convert these two sets of coordinates into the two components (physical and subgrid) of a send address. This involves two main steps:

- converting the physical coordinates into a PE number
- converting the subgrid coordinates into a memory offset

### 4.5.2 From Physical Coordinates to PE Numbers

The mapping between physical coordinates and PE numbers is as follows.

Since the number of available PEs is a power of two, and the length of the physical grid axes are also powers of two, we can represent a PE number as a string of bits, and divide that string into regions of off-chip bits, one for each physical grid axis.

The length of each physical grid axis determines the number of off-chip bits assigned to that axis — this number of bits is stored in the geometry object as the off-chip length of the axis.

The position of the off-chip bits region for each axis is one more than the sum of the off-chip lengths for all preceding axes in the physical axis sequence. This value is stored in the geometry object as the off-chip position for the axis.

The off-chip bits lengths and positions for each axis must define contiguous ranges of bits, with no intervening gaps, and the lowest off-chip position must be zero. Otherwise, any axis is free to use any region of off-chip bits. An axis may also have no off-chip bits, in which case it is represented entirely within PE memory.
The off-chip positions and lengths for the physical axes are used to convert an array element’s physical coordinates into a PE number as follows:

\[
P_{E,\text{number}} = \sum_{\text{axis}=0}^{\text{rank}-1} \text{physical-coordinate[axis]} \cdot 2^{\text{off-chip-bits-position[axis]}}
\]

On the CM-2, as noted previously, physical grid axes can be stored in either send order or NEWS order. Each geometry axis has an order attribute that determines which order to use for that axis. An axis that is NEWS-ordered uses Gray coding to change the order of physical coordinates along the axis.

Gray coding is a method of arranging a sequence of consecutive integers so that the binary representations of any two adjacent integers differ by only one bit. The RTS uses a specific type of Gray coding, called reflected binary Gray coding. For example, the reflected binary Gray code ordering of the first 16 decimal integers is:

0 1 3 2 6 7 5 4 12 13 15 14 10 11 9 8

A simple method of converting to and from reflected binary Gray code is given by the following C code procedures:

```c
static int4 to_gray_code(x)
{
    int4 x;
    { /* right-shift by 1 and XOR with original value */
        return (x ^ (x >> 1));
    }
}

static int4 from_gray_code(x)
{
    int4 result = 0;
    /* XOR original value with all right-shifted versions of it */
    while (x) { result ^= x; x >>= 1; }
    return (result);
}
```

The off-chip bits of a NEWS-ordered axis are set according to the Gray code of the physical coordinates. So for NEWS-ordered axes on the CM-2, the complete coordinate-to-PE-number translation is:

\[
\text{processor} = \sum_{\text{axis}=0}^{\text{rank}-1} \text{to_gray_code[axis]} \cdot 2^{\text{off-chip-bits-position[axis]}}
\]
Getting back to our central example for this section, on a 4-PE CM the possible PE numbers are 2 bits in length, and range from 0 to 3. Since we are using the (4,6) subgrid size, the physical grid is (2,2), and one bit of the PE number is assigned to each axis. Thus, the off-chip length for both axes is 1, and the possible off-chip positions are 0 and 1.

There are two possible physical axis sequences: (0,1) and (1,0). In this example, we'll use the (0,1) ordering: axis 0 is assigned off-chip position 0, and axis 1 is assigned position 1. An array element with physical coordinates of \( x, y \) will be stored in the memory of PE number \((x \cdot 2^0) + (y \cdot 2^1)\).

This means that element (3,9), which has physical coordinates of (0,1), is stored in the memory of PE number 2: \((0 \cdot 2^0) + (1 \cdot 2^1) = 2\).

### 4.5.3 Subgrid Coordinates and Memory Offsets

The mapping between subgrid coordinates and PE memory offsets is as follows.

A subgrid axis ordering is chosen, either axis co-variant or axis contra-variant (with some axes possibly declared to be serial). A subgrid axis increment is calculated for each axis. As described above, the increment for the first axis in the sequence is 1, and for each succeeding axis it is the product of the lengths of all preceding axes in the sequence.

The subgrid index, or position in memory, of each subgrid element is determined as follows:

\[
\text{subgrid\_index} = \sum_{\text{axis}=0}^{\text{rank}-1} \text{subgrid\_coordinate}[\text{axis}] \cdot \text{subgrid\_axis\_increment}[\text{axis}]
\]

In our (8,12) array example, element (3,9) has subgrid coordinates (3,3). Let's assume that we're using Fortran order, with no serial axes. This means that the subgrid axis sequence is (1,0).

The subgrid size is (4,6), so the axis increments are (1,6), and the order of subgrid elements in memory is:

<table>
<thead>
<tr>
<th>Index</th>
<th>Element:</th>
<th>Index</th>
<th>Element:</th>
<th>Index</th>
<th>Element:</th>
<th>Index</th>
<th>Element:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(0,0)</td>
<td>6</td>
<td>(1,0)</td>
<td>12</td>
<td>(2,0)</td>
<td>18</td>
<td>(3,0)</td>
</tr>
<tr>
<td>1</td>
<td>(0,1)</td>
<td>7</td>
<td>(1,1)</td>
<td>13</td>
<td>(2,1)</td>
<td>19</td>
<td>(3,1)</td>
</tr>
<tr>
<td>2</td>
<td>(0,2)</td>
<td>8</td>
<td>(1,2)</td>
<td>14</td>
<td>(2,2)</td>
<td>20</td>
<td>(3,2)</td>
</tr>
<tr>
<td>3</td>
<td>(0,3)</td>
<td>9</td>
<td>(1,3)</td>
<td>15</td>
<td>(2,3)</td>
<td>21</td>
<td>(3,3)</td>
</tr>
<tr>
<td>4</td>
<td>(0,4)</td>
<td>10</td>
<td>(1,4)</td>
<td>16</td>
<td>(2,4)</td>
<td>22</td>
<td>(3,4)</td>
</tr>
<tr>
<td>5</td>
<td>(0,5)</td>
<td>11</td>
<td>(1,5)</td>
<td>17</td>
<td>(2,5)</td>
<td>23</td>
<td>(3,5)</td>
</tr>
</tbody>
</table>

Element (3,9) of the array, with subgrid coordinates (3,3), has an index of 21, as you can see from either the list above or by working through the subgrid-index equation:

\[
\text{subgrid\_index} = (3 \cdot 1) + (3 \cdot 6)
\]
4.6 Defining CMCOM and CMRT Geometries

This section describes the geometry-definition functions of the CMCOM and CMRT layers, and shows how to use them to specify the geometry properties described in the preceding section.

4.6.1 Machine and Array Geometries

There are two types of geometries:

- A CMCOM-level geometry, also known as a machine geometry, specifies the overall size and shape of an array, but does not contain information about the actual contents or axis extents of an array.

- A CMRT-level geometry, also known as an array geometry, adds a garbage mask and a list of axis extents to a CMCOM geometry, defining the exact size and shape of a particular array.

The garbage mask is the primary distinction between CMCOM and CMRT geometries — in fact, a CMRT geometry is little more than a CMCOM geometry with a garbage mask attached. This reduces the amount of redundancy between arrays that have the same machine geometry but slightly different array geometries — multiple arrays can share the same machine geometry object.

There is a CMCOM function that creates an appropriate garbage mask array for a given CMCOM geometry and set of array axis extents. The CMRT geometry operations use this function to create a garbage mask in two cases: when any of the real array’s axis lengths are smaller than the corresponding machine geometry axis lengths, or where the machine geometry does not use all of the available CM processing elements.

Implementation Note: Both array and machine geometries are interned (stored for later use), so that when a call to a geometry-definition function requests a geometry that has previously been defined, the existing geometry object is returned, rather than a new one. This allows many similarly shaped arrays to share the same geometry data structure, reducing both redundancy and wasted storage space.
4.6.2 Creating a CMCOM Machine Geometry

The CMCOM operation that creates a machine geometry is:

```c
CMCOM_machine_geometry_t
    CMCOM_intern_geometry(rank, orders, off_chip_positions,
                        off_chip_lengths, subgrid_axis_lengths,
                        is_serial, highest_axis_varies_fastest);
CMCOM_machine_geometry_t
    CMCOM_intern_specific_geometry(rank, orders, axis_permutation,
                                   off_chip_masks, subgrid_axis_lengths)
```

```c
int4 rank;
CMCOM_order_t *orders;
int4 *off_chip_positions, *off_chip_lengths;
int4 *axis_permutation, *subgrid_axis_lengths;
unsigned4 *off_chip_masks;
log4 *is_serial, highest_axis_varies_fastest;
```

The arguments are as follows:

- **rank** — The rank of the array.
- **orders** — Array of `CMCOM_order_t` values; indicates the ordering (send or NEWS order) for each axis. (Only used on the CM-2. On the CM-5 it is ignored.)
- **off_chip_positions** — Array of integer; the starting positions of the off-chip bits for each axis.
- **off_chip_lengths** — Array of integers; the number of off-chip bits used for each axis
- **axis_permutation** — Specifies an axis permutation (see description below)
- **off_chip_masks** — Array of integers; off-chip bit masks ("1" bits select off-chip bits)
- **subgrid_axis_lengths** — Array of integers; the subgrid axis length for each axis.
- **is_serial** — Array of logical values; indicates which array axes (if any) are serial.
- **highest_axis_varies_fastest** — A single logical value; indicates which subgrid axis sequence is used. A `LOG_TRUE` value selects axis contra-variant order (as used by the C* and CM Fortran compilers). A `LOG_FALSE` value selects axis co-variant order.

**Axis Permutation** The `axis_permutation` argument specifies the geometry's physical/subgrid axis ordering, as well as which axes are serial. It must be either `NULL`, `LOG_TRUE`, or an array of integers.
If it is an array, it must contain the integers 0 through n-1, where n is the rank of the array. These integers must be in either ascending order (indicating an axis contra-variant ordering, the default for C* and CM Fortran arrays) or in descending order (indicating an axis co-variant ordering). Axes that are to be treated as serial may be moved (in order) to the end of the permutation array. No other permutation of axes is currently supported. If an array argument is provided and it contains a permutation not supported by the RTS, an error is signalled. For example:

0, 1, 2, 3, 4, 5, ...  
no serial axes, lowest axis varies fastest

..., 5, 4, 3, 2, 1, 0  
no serial axes, highest axis varies fastest

0, 2, 4, 6, 7, ..., 1, 5  
axis 1 and 5 are serial, lowest remaining axis fast

..., 5, 4, 1, 0, 3, 2  
axis 3 and 2 are serial, highest remaining axis fast

If the axis_permutation argument is NULL, a non-serial ascending axis order is assumed (the first case shown above). If the argument is LOG_TRUE then a non-serial descending order is assumed (the second case shown above).

4.6.3 CMCOM Geometry Objects

CMCOM_intern_geometry returns a CMCOM_machine_geometry_t data structure, which has the following slots:

- **int4 rank**
  The number of axes in the grid.

- **log4 highest_axis_varies_fastest**
  The axis ordering specified when the geometry was created.

- **struct CMCOM_axis_descriptor axis_descriptors[CMCOM_MAX_RANK]**
  Descriptors for each axis (see below).

- **int4 product_subgrid_lengths**
  The total number of elements in the subgrid. For the CM-2, this is always a multiple of four.

- **log4 array_is_power_of_two**
  True if all the subgrid lengths are powers of two.

- **int4 total_off_chip_length**
  The total of all the off-chip bits lengths, equal to the log2 of the number of processors when all processors are used.
An axis descriptor is a structure that contains the following information about an axis.

- `int4 off_chip_position`
  The off-chip bits position.

- `int4 off_chip_length`
  The off-chip bits length.

- `int4 subgrid_length`
  The subgrid-length.

- `int4 is_serial`
  True if the axis was declared as serial when the geometry was allocated.

- `int8 axis_length`
  \[ \text{subgrid.length} \times 2^{\text{off.chip.bits.length}} \]

- `unsigned4 off_chip_mask`
  \[ (2^{\text{off.chip.bits.length}} - 1) \times 2^{\text{off.chip.bits.position}} \]

- `int4 subgrid.axis.increment`
  As defined above.

- `int4 subgrid.outer.increment`
  \[ \text{subgrid.axis.increment} \times \text{subgrid.length} \]

- `int4 subgrid.outer.count`
  The product of the subgrid lengths of all axes that have larger subgrid axis increments than this axis.

- `int4 subgrid.orthogonal.length`
  The product of the subgrid lengths of all other axes.

- `int4 power.of_two`
  True when subgrid.outer.increment is an integer power of two.

- `int4 subgrid.bits.position`
  \[ \log_2(\text{subgrid.axis.increment}), \text{valid when power.of.two is true.} \]

- `int4 subgrid.bits.length`
  \[ \log_2(\text{subgrid.length}), \text{valid when power.of.two is true.} \]

- `int4 subgrid.bits.mask`
  \[ (2^{\text{subgrid.bits.length}} - 1) \times 2^{\text{subgrid.bits.position}}, \text{valid when power.of.two is true.} \]
• CMCOM.order_t order

The axis order (on the CM-2 only).

(Note that most of the above information is redundant — it is stored in the geometry object to avoid the need to repeatedly recalculate it later on.)

4.6.4 Creating a CMRT Array Geometry

There are three CMRT functions used to create array geometries:

```c
CMRT_array_geometry_t
CMRT_intern_array_geometry
(rank, lower_bounds, upper_bounds, orders,
 weights, is_serial, highest_axis_varies_fastest);
int4 rank, *weights;
int8 *lower_bounds, *upper_bounds;
CMCOM_order_t *orders;
log4 *is_serial, highest_axis_varies_fastest;

CMRT_array_geometry_t
CMRT_intern_detailed_array_geometry
(rank, lower_bounds, upper_bounds, orders,
 off_chip_positions, off_chip_lengths, subgrid_lengths,
 is_serial, highest_axis_varies_fastest);
int4 rank;
int8 *lower_bounds, *upper_bounds;
CMCOM_order_t *orders;
int4 *off_chip_positions, *off_chip_lengths, *subgrid_lengths;
log4 *is_serial, highest_axis_varies_fastest;

CMRT_array_geometry_t
CMRT_intern_specific_array_geometry
(rank, lower_bounds, upper_bounds, orders,
 axis_order, physical_masks, subgrid_extents);
int4 rank;
int8 *lower_bounds, *upper_bounds;
CMCOM_order_t *orders;
int4 *axis_order
int4 *physical_masks, *subgrid_extents;
```
The CMRT\_intern\_array\_geometry function lets you provide a looser description of the kind of array geometry that would meet your needs, and have the CMRT software itself use an algorithm to choose a set of off-chip bit lengths and positions that meets the specified requirements.

The CMRT\_intern\_detailed\_array\_geometry function lets you define an array geometry with specific properties — specific off-chip bit positions and lengths, specific subgrid sizes, etc.

The CMRT\_intern\_specific\_array\_geometry function is similar, but lets you specify bit masks to assign off-chip bits to array axes — this lets you specify noncontiguous regions of off-chip bits.

These functions share the following arguments:

- **rank** — The rank of the array.
- **lower\_bounds** — List of integers, representing the lowest axis coordinate occupied by real array data. In the current implementation, this must be 0 for each axis. (Future versions of the RTS may permit non-zero values for this argument.)
- **upper\_bounds** — List of integers, representing the highest axis coordinate occupied by real array data.
- **orders** — Array of CMCOM\_order\_t values; indicates the ordering (send or NEWS order) for each axis. (Only used on the CM-2. On the CM-5 it is ignored.)

The extent, or length, of each axis is equal to one more than the difference between its upper\_bound and lower\_bound values. Where the extent of an array geometry axis is shorter than the corresponding axis in the underlying machine geometry, a garbage mask is automatically generated and used to mask out the extra elements along the axis.

The CMRT\_intern[\_detailed]\_array\_geometry functions share the following arguments:

- **is\_serial** — Array of logical values; indicates which array axes (if any) are serial.
- **highest\_axis\_varies\_fastest** — A single logical value; indicates which subgrid axis sequence is used. A LOG\_TRUE value selects axis contra-variant order (as used by C* and CM Fortran). A LOG\_FALSE value selects axis co-variant order.

For the function CMRT\_intern\_detailed\_array\_geometry, the arguments rank, orders, subgrid\_lengths, is\_serial, and highest\_axis\_varies\_fastest are all passed directly to CMCOM\_intern\_geometry to define the underlying machine geometry, and thus have the same meanings and purpose as described above. The appropriate values for off\_chip\_positions, off\_chip\_lengths are computed from the other arguments.

Similarly, for CMRT\_intern\_detailed\_array\_geometry the arguments are passed directly to CMCOM\_intern\_specific\_geometry to define the geometry.
For **CMRT\_intern\_array\_geometry**, there are no *off\_chip\_positions*, *off\_chip\_lengths*, or *subgrid\_lengths* arguments.

Instead, the following argument is used to specify the type of machine geometry to be created:

- **weights** — A list of integers, one for each array axis, representing the relative frequency of data movement along the axis. (For example, if axis 0 is used twice as often as axis 1, the weight for axis 0 should be twice that for axis 1. If no special load-balancing is required between the axes, every axis should be given the same weight, usually 1 for all axes.)

The **CMRT\_intern\_array\_geometry** function defines its machine geometry based on the supplied axis extents and weight values, using the following rules:

1. If none of the axes are specified as serial, a set of off-chip bit lengths, off-chip bit positions, and subgrid axis lengths is chosen that minimizes the product of the subgrid lengths. This allows the array to be stored in a minimal amount of PE memory, and ensures that in-processor operations are executed most efficiently for the given array size.

2. If any axes are specified as serial, the off-chip bits lengths and subgrid lengths of the non-serial axes are calculated as if the serial axes did not exist. Then each serial axis is assigned zero off-chip bits, and given a subgrid length that is equal to the extent of the axis — this guarantees that serial axes are always represented within PE memory.

3. If the subgrid lengths calculated above for non-serial axes have factors of two in them, the supplied *weights* values are used to trade off axis subgrid lengths against off-chip bits with the goal of reducing the number of off-chip bits assigned to axes with higher weights. This increases the proportion of these axes that is represented in PE memory, with a corresponding increase in performance along those axes.

4. A geometry is only allowed to use fewer than the total number of available CM processing elements if doing so does not increase the product of the subgrid lengths. In practice, this is only true for arrays with very few elements.

**Note:** The CMRT geometry manager allows, as a special case, *null geometries*, which have zero elements. A null array geometry is created whenever one or more of the specified axis extents is zero. In this case, no machine geometry is associated with the array geometry.

A *null array*, not suprisingly, is a CMRT array with a null geometry. CMRT functions generally either do nothing with null array arguments, or, where they return values, return an appropriate identity value.

**Important:** Null arrays are a CMRT-level convenience, and do not exist at the CMCOM level.
4.6.5 CMRT Geometry Objects

The CMRT geometry functions return a `CMRT_array_geometry_t` data structure, which contains the following elements:

- **int4 rank**
  The number of axes in the geometry. This is also specified in the machine geometry; where both exist the information is redundant.

- **int8 lower_bounds[CMCOM_MAX_RANK]**
  The lower bounds of the array axes. As mentioned above, the lower bound of each axis is zero.

- **int8 upper_bounds[CMCOM_MAX_RANK]**
  The upper bounds of the array axes. As mentioned above, the upper bound of each axis is one less than the axis extent.

- **int8 extents[CMCOM_MAX_RANK]**
  \[ = 1 + \text{upper\_bounds}[\text{axis}] - \text{lower\_bounds}[\text{axis}] \]

- **int8 number_of_elements**
  The product of the extents of the axes.

- **CMCOM_machine_geometry_t machine_geometry**
  The associated CMCOM-level machine geometry. This is NULL if the number of elements in the array is zero.

- **CMRT_cm_location_t garbage_mask**
  The garbage mask associated with the array geometry. This is NULL if the array geometry’s extents are equal to the machine geometry’s extents.
4.7 Printing Information About Geometries

There are two CMRTS functions that you can use to display information about CMRT and CMCOM geometry objects:

```c
void CMCOM_print_geometry(geometry);
CMCOM_machine_geometry_t geometry;

void CMRT_print_array_geometry(geometry);
CMRT_array_geometry_t geometry;
```

Both functions print their information to the standard output. The CMRT-level function calls the CMCOM-level function to display information about the machine geometry.

For example, here's the output of `CMCOM_print_geometry` for a (842) machine geometry with (4,6) subgrids:

```
Machine geometry id: 0x3782c8, rank: 2, row major
Machine geometry elements: 96
Overall subgrid size: 24

Axis 0:
Extent: 8 (2 physical x 4 subgrid)
Off-chip: 1 bits, mask = Ox1
Subgrid: length = 4, axis-increment = 1

Axis 1:
Extent: 12 (2 physical x 6 subgrid)
Off-chip: 1 bits, mask = Ox2
Subgrid: length = 6, axis-increment = 4
```
And here's the output of `CMRT_print_array_geometry` for a (7,12) array that uses the machine geometry described above:

Array geometry id: Ox6cbb48  
Rank: 2  
Number of elements: 77  
Extents: [7 11]  

Machine geometry id: 0x3782c8, rank: 2, row major  
Machine geometry elements: 96  
Overall subgrid size: 24  

Axis 0:  
Extent: 8 (2 physical x 4 subgrid)  
Off-chip: 1 bits, mask = 0x1  
Subgrid: length = 4, axis-increment = 1  

Axis 1:  
Extent: 12 (2 physical x 6 subgrid)  
Off-chip: 1 bits, mask = 0x2  
Subgrid: length = 6, axis-increment = 4  

Note: The row major reported by these print functions is a mistake, it should say axis co-variant. This will be corrected in a future RTS version.

4.8 Writing CMCOM-level Subgrid Loops  

Algorithms that need to operate directly on the data stored in arrays at the CMCOM layer largely fall into two categories: those that need to operate on all elements in the subgrid regardless of the array's layout (typical of in-processor arithmetic), and those that need to do operations along an axis in a subgrid (typical of on-chip phases of communication operations).

The first case is simple. Given the address of the data and its increment (the distance in memory between consecutive elements in the subgrid), one uses a single loop:

```
etlement-address = address  
etlement-stride = increment  
loop product-subgrid-lengths times  
    /* Process element at element-address */  
    element-address += element-stride  
endloop```
The second case, doing computation along an axis, is more difficult. The following example shows how this is done with three nested loops, using the pre-computed looping information from the appropriate axis descriptor:

\[
\begin{align*}
\text{outer-loop-base} &= \text{address} \\
\text{outer-loop-stride} &= \text{subgrid-outer-increment} \times \text{increment} \\
\text{inner-loop-stride} &= \text{increment} \\
\text{subgrid-axis-stride} &= \text{subgrid-axis-increment} \times \text{increment} \\
\end{align*}
\]

\[
\text{loop subgrid-outer-count times} \\
\text{inner-loop-base} &= \text{outer-loop-base} \\
\text{loop subgrid-axis-increment times} \\
\text{element-address} &= \text{inner-loop-base} \\
/* \text{Start one axis} */ \\
\text{loop subgrid-length times} \\
/* \text{Process element at element-address} */ \\
\text{element-address} &\leftarrow \text{subgrid-axis-stride} \\
\text{endloop} \\
\text{inner-loop-base} &\leftarrow \text{inner-loop-stride} \\
\text{endloop} \\
\text{outer-loop-base} &\leftarrow \text{outer-loop-stride} \\
\text{endloop}
\]

Note that this algorithm works for any array layout, only needing information from one array descriptor. Code written this way will operate correctly if the geometry layout changes.

\textbf{Note:} It is possible to optimize this kind of algorithm to avoid having to keep multiple address pointers, and to write special cases for where the \text{subgrid-axis-increment} or \text{subgrid-outer-count} is one. This is left as an exercise for the reader.
4.9 Performance Issues

Without attempting to completely categorize all the performance issues involved in data layout and the CMRTS, here are some of the basic ones involved:

4.9.1 The Physical Layout

The communication performance along an axis of an array depends upon the number of on-chip and off-chip operations needed to perform the operation:

Subgrid Size, Off-Chip Operations, and Performance

<table>
<thead>
<tr>
<th>Large Subgrids, Few Off-Chip Operations</th>
<th>Small Subgrids, Many Off-Chip Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Diagram of Large Subgrids, Few Off-Chip Operations]</td>
<td>![Diagram of Small Subgrids, Many Off-Chip Operations]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Short Edges, Few Off-Chip Operations</th>
<th>Long Edges, Many Off-Chip Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Diagram of Short Edges, Few Off-Chip Operations]</td>
<td>![Diagram of Long Edges, Many Off-Chip Operations]</td>
</tr>
</tbody>
</table>

The number of on-chip operations is typically proportional to the number of subgrid elements. (The more elements there are within a subgrid, the more communication operations can be performed within the subgrid.)

The number of off-chip operations is typically inversely proportional to the size of the subgrid axis relative to the other subgrid axes. (The longer a subgrid axis, the more elements there are along its edges that must communicate with the corresponding elements of other subgrids.)

When mapping an array onto the machine, given possible machine geometries that have equal subgrid sizes (product_subgrid_lengths) but different subgrid_length and off_chip_bits_length
values for their axes, allocating fewer off-chip bits to an axis results in better communication performance along that axis, at the expense of communication performance along other axes.

The CMRT geometry operators try to take advantage of this when evaluating the weights argument (but can do so only after the minimum subgrid size constraint is satisfied). In the case where the axis extents are all powers of two, the geometry operations have maximum freedom to make weights decisions. When the axis extents are not powers of 2, the geometry operators have less freedom to trade off axis subgrid lengths against off-chip bits.

4.9.2 The Effect of Subgrid Size

There is a trade-off between subgrid size and performance of in-processor operations. Larger subgrid sizes reduce the amount of communication between PEs, but correspondingly increase the amount of looping that must be performed by a single PE to handle all the array elements in its memory. For this reason, the geometry definition functions are designed to minimize the subgrid size as much as possible.

4.9.3 Garbage Masks

If an array geometry has an associated garbage mask, some of the communication functions have to do extra work to avoid the garbage data. This is usually done by creating temporary arrays that have an identity value in locations that are garbage. This necessarily reduces performance.

An axis is said to be **padded** if its extent in the machine geometry grid is greater than than its extent in the array geometry (in other words, if there are any garbage elements along the axis). The “rotate” (wrapping NEWS) functions must do a significant number of extra operations when operating along axes that are padded.

4.9.4 Unused PE Memory

A geometry that does not require all of the PEs will still cause a subgrid-sized region of memory to be allocated on each of the unused PEs. This memory region is effectively unused, and contains no useful array data. However, this does **not** mean that the contents of this unused memory are preserved.

All RTS operations are free to modify the contents of this unused memory in the course of operating on the “real” data in the array. In general, the unused memory will be a “mirror” of the actual data, but the actual content of the unused memory at any given time is not guaranteed to be the same as the actual data.
4.9.5 Serial Axes and CM Fortran

Serial axes have special meaning for the CM Fortran compiler. In particular, the compiler allows
the elements of an array at a particular location along a serial axis to be treated directly as as an
array of lower dimensionality, without doing any extra data movement. It is important for a CM
Fortran user to understand the serial axis functionality presented by the compiler. We present here
the functionality offered by the RTS.

Exactly how serial axes are treated in the geometry manager is described above. Those rules have
the following implications:

- Elements along a serial axis will always be on the same physical processor. A serial axis always
  has an off-chip bit length of zero. Communication operations along this axis will therefore be
  optimized, as argued above in the performance discussion.

- Since the CMRT intern-array-geometry function must satisfy the subgrid constraints before
  considering serial axes, it never needs to pad a serial axis.

- If an array has a serial axis, it is possible to look at the elements of this array at a position
  along the serial axis as a sub-array of lower dimensionality; the sub-array appears as such in
  memory.

For example, suppose we create a CMRT geometry where our axis extents are (2, 7, 11), and where
the first axis is serial. Using the CMRT intern-array-geometry function, the parallel axes will be
laid out exactly as they are for the (7, 11) array, and the serial axis will have a higher subgrid
increment than the other axes.

The geometry is described as:

```
Array geometry id: 0x6cce48
  Rank: 3
  Number of elements: 154
  Extents: [2 7 11]
Machine geometry id: 0x3782c8, rank: 3, row major
  Machine geometry elements: 192
  Overall subgrid size: 48
Axis 0 (serial axis):
  Extent: 2 (1 physical x 2 subgrid)
  Off-chip: 0 bits, mask = 0x0
  Subgrid: length = 2, axis-increment = 24
Axis 1:
  Extent: 8 (2 physical x 4 subgrid)
  Off-chip: 1 bits, mask = 0x1
  Subgrid: length = 4, axis-increment = 1
```
Axis 2:
   Extent: 12 (2 physical x 6 subgrid)
   Off-chip: 1 bits, mask = 0x2
   Subgrid: length = 6, axis-increment = 4

The resulting memory layout of the subgrid elements is (reading first across and then down):

   (0, 0, 0) (0, 1, 0) (0, 2, 0) (0, 3, 0)
   (0, 0, 1) (0, 1, 1) ... (0, 3, 5)
   (1, 0, 0) (1, 1, 0) (1, 2, 0) (1, 3, 0)
   (1, 0, 1) (1, 1, 1) ... (1, 3, 5)

The subgrid elements (0, 0, 0) through (0, 3, 5) are laid out exactly as the subgrid elements are in the two-dimensional case, the same is true for (1, 0, 0) through (1, 3, 5). For Fortran this means that an (n,:,:,:) array can be viewed as an array of lower dimensionality without having to move data.
4.10 Miscellaneous Terminology Issues

4.10.1 Row-Major and Column-Major

There has been much confusion over how the run-time system orders subgrid elements in memory and what this has to do with row-major and column-major orderings. The layout of subgrid elements has changed from time to time only because the issues were not well understood from the start. Here are some definitions and explanations of the key concepts (with many thanks to Guy Steele, who made this all clear):

Rows, Columns, and Ordered Pairs

In an ordered pair, which is the row and which is the column?

Mathematicians are not consistent on this issue. Matrix notation generally writes [row, column], but Cartesian notation generally writes (x, y) where y is the vertical (or column) axis. Among computer languages the matrix notation is generally accepted; the (row, column) order is the convention for both C and Fortran. (But there are exceptions—graphics code often uses the Cartesian notation.)

Row-Major and Column-Major Ordering

Given two-dimensional representation of rows and columns, how are the elements in the grid ordered, say, in memory?

An equivalent question is how ordered pairs in the form (row, column) are sorted. Row-major order is the sorting order where the row is the major key (that is, the row numbering changes most slowly); column-major order is the sorting order where the column is the major key (that is, the column number changes most slowly).

With all elements of an array sorted into a linear memory space, the axis increment is the distance in memory between adjacent elements along an axis. In row-major order, the row’s axis increment is 1; neighbors along a row are neighbors in memory. In column-major order, the column’s axis increment is 1; neighbors along a column are neighbors in memory.

Fortran is column-major, C is row-major. The fact that video systems scan images horizontally makes video signals naturally row-major. (Alex Vasilevsky claims that he stands his television on its side so that it can be more Fortran-compatible.)

Enumerating the Coordinates

Are coordinates written such that axes are numbered left-to-right or right-to-left?

This is only an issue when one needs to refer to axes by number. (C, for example, does not define a numbering for for its axes, and C programmers are not consistent on how they think the axes should be numbered.) Arguments can be made for adopting either numbering scheme.

CM Fortran numbers axes from left to right. C* numbers parallel axes from left to right, but, consistent with C, does not adopt a numbering scheme for axes of in-processor arrays.
Axis Co-Variant and Axis Contra-Variant

Given numbered axes, how are axis increments calculated?

(Of all these issues, this is the only one that makes sense in the context of the run-time system itself, since it deals only with numbered axes, not written axes.)

There are two ways to calculate axis increments. For, say a four-dimensional grid, they are:

Axis co-variant layout (axis increments increase with axis number):

\[
\begin{align*}
\text{axis} & : \text{ increment} \\
0 & : 1 \\
1 & : \text{extent}[\text{axis 0}] \\
2 & : \text{extent}[\text{axis 0}] \times \text{extent}[\text{axis 1}] \\
3 & : \text{extent}[\text{axis 0}] \times \text{extent}[\text{axis 1}] \times \text{extent}[\text{axis 2}]
\end{align*}
\]

Axis contra-variant layout (axis increments decrease with axis number):

\[
\begin{align*}
\text{axis} & : \text{ increment} \\
0 & : \text{extent}[\text{axis 3}] \times \text{extent}[\text{axis 2}] \times \text{extent}[\text{axis 1}] \\
1 & : \text{extent}[\text{axis 3}] \times \text{extent}[\text{axis 2}] \\
2 & : \text{extent}[\text{axis 3}] \\
3 & : 1
\end{align*}
\]

When calculating subgrid strides, the CMCOM software will use either a co-variant or contra-variant layout, depending upon the value of the highest-axis-varies-fastest flag (contra-variant if the flag is true, co-variant if the flag is false).

In either case axes specified as serial are considered a separate group of axes for the purpose of assigning increments, such that serial axes always have larger increments than non-serial axes. The serial axes are given strides that are either co-variant or contra-variant, depending upon the highest-axis-varies-fastest flag.

Note: The terms axis co-variant and axis contra-variant were proposed by Guy Steele.
5 The CMTRA Interface

This section lists and describes the routines available in the CMTRA library. This library was designed to directly support the arithmetic intrinsics of the CMF and C* languages. (For more detailed descriptions of the arguments and function of these routines, refer to the reference manuals for these languages.)

The library consists of arithmetic routines written in the low-level assembly language (CMIS or DPEAC) of the CM. These routines are normally called from code generated by compilers and so work in a different way from the CMCOM and CMIP routines. Specifically, these routines are designed to operate directly on data that has been previously loaded into the vector registers on the CM2/CM200 Floating Point Accelerator or the CM5 Vector Units. These routines are also intended to be called from within subgrid loop code on the CM processing nodes — these routines cannot be called from Partition Manager code, or in any other way.

Note: These routines will only be of use to you if you are familiar with programming the CM at the assembly-code level. Also, these routines are NOT available in the SPARC (CM5 without VU) version of the RTS.

The expected input and output registers are listed for each routine, and are given using the internal vector register names. For reference, here is the mapping of vector register names to register numbers in the two systems:

CM2/CM200: Vectors are 4 words or double words in length.

<table>
<thead>
<tr>
<th>V0</th>
<th>R0 through R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>R4 through R7</td>
</tr>
<tr>
<td>V2</td>
<td>R8 through R11</td>
</tr>
<tr>
<td>V3</td>
<td>R12 through R15</td>
</tr>
<tr>
<td>V4</td>
<td>R16 through R19</td>
</tr>
<tr>
<td>V5</td>
<td>R20 through R24</td>
</tr>
<tr>
<td>V6</td>
<td>R24 through R27</td>
</tr>
<tr>
<td>V7</td>
<td>R28 through R31</td>
</tr>
</tbody>
</table>

CM5/VU: Each vector register is 8 single-precision elements in length. For double precision, pairs of vector registers are used. For example, if a routine takes its input from V2, and that input is double precision, such input will actually lie in V2 and V3.

<table>
<thead>
<tr>
<th>V2/V3</th>
<th>R16 through R31</th>
</tr>
</thead>
<tbody>
<tr>
<td>V4/V5</td>
<td>R32 through R47</td>
</tr>
<tr>
<td>V6/V7</td>
<td>R48 through R63</td>
</tr>
<tr>
<td>V8/V9</td>
<td>R64 through R79</td>
</tr>
<tr>
<td>V10/V11</td>
<td>R80 through R95</td>
</tr>
<tr>
<td>V12/V13</td>
<td>R96 through R111</td>
</tr>
<tr>
<td>V14/V15</td>
<td>R112 through R127</td>
</tr>
</tbody>
</table>
The CMTRA routines may also modify "scalar registers" as listed. For the CM2, the "scalar registers" are sequencer registers. For these, the PEAC naming is used, e.g. S0 through S31. For the CM5, the "scalar registers" are the Vector Unit registers in the range R0 through R15.

On the CM2, these routines may modify the CM2 Sequencer registers, as shown. Here again, the PEAC (CMIS) naming is used e.g. the data registers A1 through A12, counter registers C1 through C4, pointer registers P1 through P12, and so forth.

On the CM5, SPARC ABI (C) register usage is obeyed, thus SPARC register usage follows C subroutine calling conventions.

On the CM2/CM200, some stack memory may be used as temporary space, as shown (in 32-bit words, or "slices").

On the CM5, The Vector Mask and Vector Mask Buffer registers in the Vector Units may be modified, as shown.

The descriptions in this section have a different format from that used in the preceding sections. For each CMTRA routine, the following information is provided:

- The name of the routine and its type: Float, Double, etc.
- A description of the routine’s purpose
- The input registers used for operands
- The output registers used to return the result
- The highest vector register that is modified
- The lowest scalar register that is modified
- For CMIS routines:
  - The sequencer registers modified, if any
  - The CMIS registers modified, if any
  - The amount of stack memory used, in 4-byte slices
- For VU routines:
  - Whether the vector mask register is modified
  - The approximate cycle count
5.1 CMIS routines

The routines listed below are available in the CMIS (CM-2) version of CMTRA.

5.1.1 BTEST (Integer)

Description: Tests whether a given bit of an integer value is set.

Registers:
Input: V0 V1
Output: V0

Usage Information:
Highest modified vector register: V1
Lowest scalar register modified: S31
Stack Memory Used (4-byte slices): 4

5.1.2 IBCLR (Integer)

Description: Clear (set to zero) a given bit of an integer value.

Registers:
Input: V0 V1
Output: V0

Usage Information:
Highest modified vector register: V1
Lowest scalar register modified: S31
Stack Memory Used (4-byte slices): 4

5.1.3 IBITS (Integer)

Description: Extract a subfield of bits from an integer value.

Registers:
Input: V0 V1 V2
Output: V0

Usage Information:
Highest modified vector register: V2
Lowest scalar register modified: S31
Stack Memory Used (4-byte slices): 4
5.1.4 IBSET (Integer)

Description: Set a given bit of an integer value to 1.

Registers:
- Input: V0 V1
- Output: V0

Usage Information:
- Highest modified vector register: V1
- Lowest scalar register modified: S31
- Stack Memory Used (4-byte slices): 4

5.1.5 ACOS (Float)

Description: Trigonometric Arccosine in radians

Registers:
- Input: V0
- Output: V0

Usage Information:
- Highest modified vector register: V4
- Lowest scalar register modified: S31
- Sequencer registers modified: A11
- Stack Memory Used (4-byte slices): 0

5.1.6 CCOS (Complex)

Description: Trigonometric Cosine

Registers:
- Input: V0 V1
- Output: V0 V1 V2 V3

Usage Information:
- Highest modified vector register: V6
- Lowest scalar register modified: S30
- Sequencer registers modified: A10 A11
- Stack Memory Used (4-byte slices): 8
5.1.7 CDCOS (Dcomplex)

**Description:** Trigonometric Cosine

**Registers:**
- Input: V0 V1
- Output: V0 V1 V2 V3

**Usage Information:**
- Highest modified vector register: V6
- Lowest scalar register modified: S30
- Sequencer registers modified: A10 A11
- Stack Memory Used (4-byte slices): 8

5.1.8 COS (Float)

**Description:** Trigonometric Cosine

**Registers:**
- Input: V0
- Output: V0

**Usage Information:**
- Highest modified vector register: V2
- Lowest scalar register modified: S30
- Sequencer registers modified: A11
- Stack Memory Used (4-byte slices): 0

5.1.9 COSH (Float)

**Description:** Trigonometric Hyperbolic Cosine

**Registers:**
- Input: V0
- Output: V0

**Usage Information:**
- Highest modified vector register: V1
- Lowest scalar register modified: S30
- Sequencer registers modified: A11
- Stack Memory Used (4-byte slices): 4
5.1.10 DACOS (Double)

Description: Trigonometric Arcosine in radians

Registers:
- Input: V0
- Output: V0

Usage Information:
- Highest modified vector register: V4
- Lowest scalar register modified: S31
- Sequencer registers modified: A11
- Stack Memory Used (4-byte slices): 0

5.1.11 DCOS (Double)

Description: Trigonometric Cosine

Registers:
- Input: V0
- Output: V0

Usage Information:
- Highest modified vector register: V4
- Lowest scalar register modified: S30
- Sequencer registers modified: A11
- Stack Memory Used (4-byte slices): 0

5.1.12 DCOSH (Double)

Description: Trigonometric Hyperbolic Cosine

Registers:
- Input: V0
- Output: V0

Usage Information:
- Highest modified vector register: V2
- Lowest scalar register modified: S30
- Sequencer registers modified: A11
- Stack Memory Used (4-byte slices): 4
5.1.13  CDEXP (Dcomplex)

Description: Computes power of e (base of natural logarithms)

Registers:
  Input: V0 V1
  Output: V0 V1 V2 V3

Usage Information:
  Highest modified vector register: V6
  Lowest scalar register modified: S30
  Sequencer registers modified: A10 A11
  Stack Memory Used (4-byte slices): 4

5.1.14  CEXP (Complex)

Description:

Registers:
  Input: V0 V1
  Output: V0 V1 V2 V3

Usage Information:
  Highest modified vector register: V5
  Lowest scalar register modified: S30
  Sequencer registers modified: A10 A11
  Stack Memory Used (4-byte slices): 4

5.1.15  DEXP (Double)

Description: Computes power of e (base of natural logarithms)

Registers:
  Input: V0
  Output: V0

Usage Information:
  Highest modified vector register: V2
  Lowest scalar register modified: S31
  Stack Memory Used (4-byte slices): 4
5.1.16 EXP (Float)

**Description:** Computes power of e (base of natural logarithms)

**Registers:**
- Input: V0
- Output: V0

**Usage Information:**
- Highest modified vector register: V1
- Lowest scalar register modified: S31
- Stack Memory Used (4-byte slices): 4

5.1.17 AINT (Float)

**Description:** Truncate real value to whole number real value.

**Registers:**
- Input: V0
- Output: V0

**Usage Information:**
- Highest modified vector register: V3
- Lowest scalar register modified: S31
- Stack Memory Used (4-byte slices): 0

5.1.18 ANINT (Float)

**Description:** Round real value to whole number real value.

**Registers:**
- Input: V0
- Output: V0

**Usage Information:**
- Highest modified vector register: V3
- Lowest scalar register modified: S31
- Stack Memory Used (4-byte slices): 0
5.1.19  DINT (Double)

Description: Truncate double value to whole number double value.

 Registers:
   Input: V0
   Output: V0

Usage Information:
   Highest modified vector register: V3
   Lowest scalar register modified: S31
   Stack Memory Used (4-byte slices): 0

5.1.20  DNINT (Double)

Description: Round double value to whole number double value.

 Registers:
   Input: V0
   Output: V0

Usage Information:
   Highest modified vector register: V3
   Lowest scalar register modified: S31
   Stack Memory Used (4-byte slices): 0

5.1.21  CDLOG (Dcomplex)

Description: Natural logarithm (base e)

 Registers:
   Input: V0 V1
   Output: V0 V1 V2 V3

Usage Information:
   Highest modified vector register: V5
   Lowest scalar register modified: S31
   Sequencer registers modified: A11
   Stack Memory Used (4-byte slices): 4
5.1.22 CLOG (Complex)

Description: Natural logarithm (base e)

Registers:
Input: V0 V1
Output: V0 V1 V2 V3

Usage Information:
Highest modified vector register: V5
Lowest scalar register modified: S31
Sequencer registers modified: A11
Stack Memory Used (4-byte slices): 4

5.1.23 DLOG (Double)

Description: Natural logarithm (base e)

Registers:
Input: V0
Output: V0

Usage Information:
Highest modified vector register: V2
Lowest scalar register modified: S31
Stack Memory Used (4-byte slices): 4

5.1.24 LOG (Float)

Description: Natural logarithm (base e)

Registers:
Input: V0
Output: V0

Usage Information:
Highest modified vector register: V2
Lowest scalar register modified: S31
Stack Memory Used (4-byte slices): 4
5.1.25 DLOG10 (Double)

Description: Logarithm (base 10)

Registers:
Input: V0
Output: V0

Usage Information:
Highest modified vector register: V2
Lowest scalar register modified: S31
Sequencer registers modified: A11
Stack Memory Used (4-byte slices): 4

5.1.26 LOG10 (Float)

Description: Logarithm (base 10)

Registers:
Input: V0
Output: V0

Usage Information:
Highest modified vector register: V2
Lowest scalar register modified: S31
Sequencer registers modified: A11
Stack Memory Used (4-byte slices): 4

5.1.27 C.C_POWER (Complex)

Description: Complex to Complex Power

Registers:
Input: V0 V1 V2 V3
Output: V0 V1 V2 V3

Usage Information:
Highest modified vector register: V6
Lowest scalar register modified: S30
Sequencer registers modified: A10 A11
Stack Memory Used (4-byte slices): 8
5.1.28  C.F_POWER (Complex)

**Description:** Complex to Float Power

**Registers:**
- Input: V0 V1 V2 V3 V4 V5 V6
- Output: V0 V1 V2 V3

**Usage Information:**
- Highest modified vector register: V6
- Lowest scalar register modified: S30
- Sequencer registers modified: A10 A11
- Stack Memory Used (4-byte slices): 4

5.1.29  C.IC_POWER (Complex)

**Description:** Complex to Integer Complex Power

**Registers:**
- Input: V0 V1 V2 V3 V4 V5 V6
- Output: V0 V1 V2 V3

**Usage Information:**
- Highest modified vector register: V6
- Lowest scalar register modified: S31
- Sequencer registers modified: A1
- CMIS registers modified: C1 C3 P4
- Stack Memory Used (4-byte slices): 1

**Restrictions:** Integer argument must be the same for all nodes.
5.1.30  C_I_POWER (Complex)

Description: Complex to Integer Power

Registers:
- Input: V0 V1 V2 V3 V4 V5
- Output: V0 V1 V2 V3

Usage Information:
- Highest modified vector register: V6
- Lowest scalar register modified: S31
- Sequencer registers modified: A11
- CMIS registers modified: C1
- Stack Memory Used (4-byte slices): 12

5.1.31  D_D_POWER (Double)

Description: Double to Double Power

Registers:
- Input: V0 V1
- Output: V0

Usage Information:
- Highest modified vector register: V5
- Lowest scalar register modified: S31
- Sequencer registers modified: A11
- Stack Memory Used (4-byte slices): 4
5.1.32 D_IC_POWER (Double)

Description: Double to Integer Complex Power

Registers:
- Input: V0 V1 V2 V3
- Output: V0 V1

Usage Information:
- Highest modified vector register: V2
- Lowest scalar register modified: S31
- Sequencer registers modified: A1
- CMIS registers modified: C1 C3 P4
- Stack Memory Used (4-byte slices): 1

Restrictions: Integer argument must be the same for all nodes.

5.1.33 D_I_POWER (Double)

Description: Double to Integer Power

Registers:
- Input: V0 V1
- Output: V0

Usage Information:
- Highest modified vector register: V4
- Lowest scalar register modified: S31
- CMIS registers modified: C1
- Stack Memory Used (4-byte slices): 4

5.1.34 F_FPOWER (Float)

Description: Float to Float Power

Registers:
- Input: V0 V1
- Output: V0

Usage Information:
- Highest modified vector register: V3
- Lowest scalar register modified: S31
- Sequencer registers modified: A11
- Stack Memory Used (4-byte slices): 4
5.1.35 F_IC_POWER (Float)

Description: Float to Integer Complex Power

Registers:
Input: V0 V1 V2 V3
Output: V0 V1

Usage Information:
- Highest modified vector register: V2
- Lowest scalar register modified: S31
- Sequencer registers modified: A1
- CMIS registers modified: C1 C3 P4
- Stack Memory Used (4-byte slices): 1

Restrictions: Integer argument must be the same for all nodes.

5.1.36 F_I_POWER (Float)

Description: Float to integer power

Registers:
Input: V0 V1
Output: V0

Usage Information:
- Highest modified vector register: V4
- Lowest scalar register modified: S31
- Sequencer registers modified: A11
- Stack Memory Used (4-byte slices): 4

5.1.37 I_I_POWER (Integer)

Description: Integer to integer power

Registers:
Input: V0 V1
Output: V0

Usage Information:
- Highest modified vector register: V4
- Lowest scalar register modified: S31
- Stack Memory Used (4-byte slices): 4
5.1.38  **Z_D_POWER** (Dcomplex)

**Description:** Unsigned Integer to Double Power

**Registers:**
- Input: V0 V1 V2 V3 V4 V5
- Output: V0 V1 V2 V3

**Usage Information:**
- Highest modified vector register: V6
- Lowest scalar register modified: S30
- Sequencer registers modified: A1 All
- Stack Memory Used (4-byte slices): 4

5.1.39  **Z_IC_POWER** (Dcomplex)

**Description:** Unsigned Integer to Integer Complex Power

**Registers:**
- Input: V0 V1 V2 V3 V4 V5
- Output: V0 V1 V2 V3

**Usage Information:**
- Highest modified vector register: V6
- Lowest scalar register modified: S31
- Sequencer registers modified: A1
- CMIS registers modified: C1 C3 P4
- Stack Memory Used (4-byte slices): 1

**Restrictions:** Integer argument must be the same for all nodes.

5.1.40  **Z_I_POWER** (Dcomplex)

**Description:** Unsigned Integer to Integer Power

**Registers:**
- Input: V0 V1 V2 V3 V4 V5
- Output: V0 V1 V2 V3

**Usage Information:**
- Highest modified vector register: V6
- Lowest scalar register modified: S31
- CMIS registers modified: C1
- Stack Memory Used (4-byte slices): 12
5.1.41 Z.Z_POWER (Dcomplex)

Description: Unsigned Integer to Unsigned Integer Power

Registers:
Input: V0 V1 V2 V3
Output: V0 V1 V2 V3

Usage Information:
Highest modified vector register: V6
Lowest scalar register modified: S30
Sequencer registers modified: A.10 A.11
Stack Memory Used (4-byte slices): 12

5.1.42 ISHFT (Integer)

Description: Integer Arithmetic Shift

Registers:
Input: V0 V1
Output: V0

Usage Information:
Highest modified vector register: V1
Lowest scalar register modified: S31
Stack Memory Used (4-byte slices): 4

5.1.43 ISHFTC (Integer)

Description: Integer Logical Shift

Registers:
Input: V0 V1 V2
Output: V0

Usage Information:
Highest modified vector register: V4
Lowest scalar register modified: S30
Stack Memory Used (4-byte slices): 4
5.1.44 ASIN (Float)

Description: Trigonometric Arcsine in radians

Registers:
Input: V0
Output: V0

Usage Information:
Highest modified vector register: V4
Lowest scalar register modified: S31
Sequencer registers modified: A11
Stack Memory Used (4-byte slices): 0

5.1.45 CDSIN (Dcomplex)

Description: Trigonometric Sine

Registers:
Input: V0 V1
Output: V0 V1 V2 V3

Usage Information:
Highest modified vector register: V6
Lowest scalar register modified: S30
Sequencer registers modified: A10 A11
Stack Memory Used (4-byte slices): 8

5.1.46 CSIN (Complex)

Description: Trigonometric Sine

Registers:
Input: V0 V1
Output: V0 V1 V2 V3

Usage Information:
Highest modified vector register: V6
Lowest scalar register modified: S30
Sequencer registers modified: A10 A11
Stack Memory Used (4-byte slices): 8
5.1.47 DASIN (Double)

Description: Trigonometric Arcsine in radians

Registers:
   Input: V0
   Output: V0

Usage Information:
   Highest modified vector register: V4
   Lowest scalar register modified: S3I
   Sequencer registers modified: A1I
   Stack Memory Used (4-byte slices): 0

5.1.48 DSIN (Double)

Description: Trigonometric Sine

Registers:
   Input: V0
   Output: V0

Usage Information:
   Highest modified vector register: V4
   Lowest scalar register modified: S30
   Sequencer registers modified: A1I
   Stack Memory Used (4-byte slices): 0

5.1.49 DSINH (Double)

Description: Trigonometric Hyperbolic Sine

Registers:
   Input: V0
   Output: V0

Usage Information:
   Highest modified vector register: V4
   Lowest scalar register modified: S30
   Sequencer registers modified: A1I
   Stack Memory Used (4-byte slices): 8
5.1.50 SIN (Float)

Description: Trigonometric Sine

Registers:
   Input: V0
   Output: V0

Usage Information:
   Highest modified vector register: V2
   Lowest scalar register modified: S30
   Sequencer registers modified: A11
   Stack Memory Used (4-byte slices): 0

5.1.51 SINH (Float)

Description: Trigonometric Hyperbolic Sine

Registers:
   Input: V0
   Output: V0

Usage Information:
   Highest modified vector register: V4
   Lowest scalar register modified: S30
   Sequencer registers modified: A11
   Stack Memory Used (4-byte slices): 8

5.1.52 CDSQRT (Dcomplex)

Description: Squareroot Function

Registers:
   Input: V0 V1
   Output: V0 V1 V2 V3

Usage Information:
   Highest modified vector register: V4
   Lowest scalar register modified: S31
   Stack Memory Used (4-byte slices): 0
5.1.53  CSQRT (Complex)

Description: Squareroot Function

Registers:
  Input: V0 V1
  Output: V0 V1 V2 V3

Usage Information:
  Highest modified vector register: V4
  Lowest scalar register modified: S31
  Stack Memory Used (4-byte slices): 0

5.1.54  ATAN (Float)

Description: Trigonometric Arctangent (1-argument form) in radians

Registers:
  Input: V0
  Output: V0

Usage Information:
  Highest modified vector register: V4
  Lowest scalar register modified: S31
  Sequencer registers modified: A11
  Stack Memory Used (4-byte slices): 4

5.1.55  ATAN2 (Float)

Description: Trigonometric Arctangent (2-argument form) in radians

Registers:
  Input: V0 V1
  Output: V0

Usage Information:
  Highest modified vector register: V4
  Lowest scalar register modified: S31
  Stack Memory Used (4-byte slices): 4
5.1.56 DATAN (Double)

Description: Trigonometric Arctangent (1-argument form) in radians

Registers:
- Input: V0
- Output: V0

Usage Information:
- Highest modified vector register: V4
- Lowest scalar register modified: S31
- Sequencer registers modified: A11
- Stack Memory Used (4-byte slices): 4

5.1.57 DATAN2 (Double)

Description: Trigonometric Arctangent (2-argument form) in radians

Registers:
- Input: V0 V1
- Output: V0

Usage Information:
- Highest modified vector register: V4
- Lowest scalar register modified: S31
- Stack Memory Used (4-byte slices): 4

5.1.58 DTAN (Double)

Description: Trigonometric Tangent

Registers:
- Input: V0
- Output: V0

Usage Information:
- Highest modified vector register: V4
- Lowest scalar register modified: S30
- Stack Memory Used (4-byte slices): 4
5.1.59 DTANH (Double)

**Description:** Trigonometric Hyperbolic Tangent

**Registers:**
- Input: V0
- Output: V0

**Usage Information:**
- Highest modified vector register: V4
- Lowest scalar register modified: S30
- Sequencer registers modified: A10 A11
- Stack Memory Used (4-byte slices): 8

5.1.60 TAN (Float)

**Description:** Trigonometric Tangent

**Registers:**
- Input: V0
- Output: V0

**Usage Information:**
- Highest modified vector register: V3
- Lowest scalar register modified: S30
- Stack Memory Used (4-byte slices): 0

5.1.61 TANH (Float)

**Description:** Trigonometric Hyperbolic Tangent

**Registers:**
- Input: V0
- Output: V0

**Usage Information:**
- Highest modified vector register: V4
- Lowest scalar register modified: S30
- Sequencer registers modified: A10 A11
- Stack Memory Used (4-byte slices): 8
5.2 DPEAC routines

The routines listed below are available in the DPEAC (CM-5) version of CMTRA, and also in the CM Simulator Library.

5.2.1 BTEST (single precision)

**Description**: Tests whether a given bit of an integer value is set.

**Registers**:
- Input: V2 V4
- Output: V2

**Usage Information**:
- Highest modified vector register: V4
- No scalar registers are modified.
- Vector Mask Modified: Yes
- Approximate cycle count: 48

5.2.2 KBTEST (double precision)

**Description**: Tests whether a given bit of an integer value is set.

**Registers**:
- Input: V2 V4
- Output: V2

**Usage Information**:
- Highest modified vector register: V5
- No scalar registers are modified.
- Vector Mask Modified: Yes
- Approximate cycle count: 48
5.2.3 IBCLR (single precision)

Description: Clear (set to zero) a given bit of an integer value.

 Registers:
   Input: V2 V4
   Output: V2

Usage Information:
   Highest modified vector register: V4
   No scalar registers are modified.
   Vector Mask Modified: No
   Approximate cycle count: 32

5.2.4 KBCLR (double precision)

Description: Clear (set to zero) a given bit of an integer value.

 Registers:
   Input: V2 V4
   Output: V2

Usage Information:
   Highest modified vector register: V5
   No scalar registers are modified.
   Vector Mask Modified: No
   Approximate cycle count: 32

5.2.5 IBITS (single precision)

Description: Extract a subfield of bits from an integer value.

 Registers:
   Input: V2 V4
   Output: V2

Usage Information:
   Highest modified vector register: V6
   No scalar registers are modified.
   Vector Mask Modified: No
   Approximate cycle count: 64
5.2.6 KBITS (double precision)

Description: Extract a subfield of bits from an integer value.

Registers:
   Input: V2 V4
   Output: V2

Usage Information:
   Highest modified vector register: V7
   Lowest scalar register modified: R14
   Vector Mask Modified: No
   Approximate cycle count: 64

5.2.7 KDIV (double precision)

Description: Integer Division

Registers:
   Input: V2 V4
   Output: V2

Usage Information:
   Highest modified vector register: V13
   Lowest scalar register modified: R15
   Vector Mask Modified: Yes
   Approximate cycle count: 250

5.2.8 IBSET (single precision)

Description: Set a given bit of an integer value to 1.

Registers:
   Input: V2 V4
   Output: V2

Usage Information:
   Highest modified vector register: V4
   No scalar registers are modified.
   Vector Mask Modified: No
   Approximate cycle count: 32
5.2.9 KBSET (double precision)

Description: Set a given bit of an integer value to 1.

Registers:
- Input: V2 V4
- Output: V2

Usage Information:
- Highest modified vector register: V5
- No scalar registers are modified.
- Vector Mask Modified: No
- Approximate cycle count: 32

5.2.10 ACOS (single precision)

Description: Trigonometric Arccosine in radians

Registers:
- Input: V2
- Output: V2

Usage Information:
- Highest modified vector register: V9
- Lowest scalar register modified: R15
- Vector Mask Modified: Yes
- Approximate cycle count: 392

5.2.11 CCOS (single precision)

Description: Trigonometric Cosine

Registers:
- Input: V2 V4
- Output: V2 V4

Usage Information:
- Highest modified vector register: V13
- Lowest scalar register modified: R14
- Vector Mask Modified: Yes
- Approximate cycle count: 1616
5.2.12 CDCOS (double precision)

Description: Trigonometric Cosine

Registers:
  Input: V2 V4
  Output: V2 V4

Usage Information:
  Highest modified vector register: V15
  Lowest scalar register modified: R14
  Vector Mask Modified: Yes
  Approximate cycle count: 2336

5.2.13 COS (single precision)

Description: Trigonometric Cosine

Registers:
  Input: V2
  Output: V2

Usage Information:
  Highest modified vector register: V9
  No scalar registers are modified.
  Vector Mask Modified: Yes
  Approximate cycle count: 400

5.2.14 COSH (single precision)

Description: Trigonometric Hyperbolic Cosine

Registers:
  Input: V2
  Output: V2

Usage Information:
  Highest modified vector register: V7
  No scalar registers are modified.
  Vector Mask Modified: Yes
  Approximate cycle count: 416
5.2.15 DACOS (double precision)

Description: Trigonometric Arccosine in radians

Registers:
- Input: V2
- Output: V2

Usage Information:
- Highest modified vector register: V9
- Lowest scalar register modified: R15
- Vector Mask Modified: Yes
- Approximate cycle count: 408

5.2.16 DCOS (double precision)

Description: Trigonometric Cosine

Registers:
- Input: V2
- Output: V2

Usage Information:
- Highest modified vector register: V11
- No scalar registers are modified.
- Vector Mask Modified: Yes
- Approximate cycle count: 672

5.2.17 DCOSH (double precision)

Description: Trigonometric Hyperbolic Cosine

Registers:
- Input: V2
- Output: V2

Usage Information:
- Highest modified vector register: V9
- No scalar registers are modified.
- Vector Mask Modified: Yes
- Approximate cycle count: 592
5.2.18  CDEXP (double precision)

Description: Computes power of e (base of natural logarithms)

Registers:
  Input: V2 V4
  Output: V2 V4

Usage Information:
  Highest modified vector register: V15
  Lowest scalar register modified: R14
  Vector Mask Modified: Yes
  Approximate cycle count: 2080

5.2.19  CEXP (single precision)

Description: Computes power of e (base of natural logarithms)

Registers:
  Input: V2 V4
  Output: V2 V4

Usage Information:
  Highest modified vector register: V13
  Lowest scalar register modified: R14
  Vector Mask Modified: Yes
  Approximate cycle count: 1360

5.2.20  DEXP (double precision)

Description: Computes power of e (base of natural logarithms)

Registers:
  Input: V2
  Output: V2

Usage Information:
  Highest modified vector register: V7
  Lowest scalar register modified: R14
  Vector Mask Modified: Yes
  Approximate cycle count: 544
5.2.21 EXP (single precision)

Description: Computes power of e (base of natural logarithms)

Registers:
   Input: V2
   Output: V2

Usage Information:
   Highest modified vector register: V7
   Lowest scalar register modified: R14
   Vector Mask Modified: Yes
   Approximate cycle count: 352

5.2.22 AINT (single precision)

Description: Truncate real value to whole number real value.

Registers:
   Input: V2
   Output: V2

Usage Information:
   Highest modified vector register: V3
   Lowest scalar register modified: R15
   Vector Mask Modified: Yes
   Approximate cycle count: 112

5.2.23 ANINT (single precision)

Description: Round real value to whole number real value.

Registers:
   Input: V2
   Output: V2

Usage Information:
   Highest modified vector register: V3
   Lowest scalar register modified: R15
   Vector Mask Modified: Yes
   Approximate cycle count: 128
5.2.24 DINT (double precision)

Description: Truncate double value to whole number double value.

Registers:
  Input: V2
  Output: V2

Usage Information:
  Highest modified vector register: V3
  Lowest scalar register modified: R15
  Vector Mask Modified: Yes
  Approximate cycle count: 112

5.2.25 DNINT (double precision)

Description: Round double value to whole number double value.

Registers:
  Input: V2
  Output: V2

Usage Information:
  Highest modified vector register: V3
  Lowest scalar register modified: R15
  Vector Mask Modified: Yes
  Approximate cycle count: 128

5.2.26 CDLOG (double precision)

Description: Natural Logarithm (base e)

Registers:
  Input: V2 V4
  Output: V2 V4

Usage Information:
  Highest modified vector register: V11
  Lowest scalar register modified: R11
  Vector Mask Modified: Yes
  Approximate cycle count: 1240
5.2.27 CLOG (single precision)

**Description:** Natural logarithm (base \( e \))

**Registers:**
- Input: V2 V4
- Output: V2 V4

**Usage Information:**
- Highest modified vector register: V11
- Lowest scalar register modified: R11
- Vector Mask Modified: Yes
- Approximate cycle count: 1176

5.2.28 DLOG (double precision)

**Description:** Natural Logarithm (base \( e \))

**Registers:**
- Input: V2
- Output: V2

**Usage Information:**
- Highest modified vector register: V9
- Lowest scalar register modified: R11
- Vector Mask Modified: Yes
- Approximate cycle count: 568

5.2.29 LOG (single precision)

**Description:** Natural logarithm (base \( e \))

**Registers:**
- Input: V2
- Output: V2

**Usage Information:**
- Highest modified vector register: V9
- Lowest scalar register modified: R11
- Vector Mask Modified: Yes
- Approximate cycle count: 520
5.2.30  DLOG10 (double precision)

Description: Logarithm (base 10)

Registers:
   Input: V2
   Output: V2

Usage Information:
   Highest modified vector register: V9
   Lowest scalar register modified: R11
   Vector Mask Modified: Yes
   Approximate cycle count: 568

5.2.31  LOG10 (single precision)

Description: Logarithm (base 10)

 Registers:
   Input: V2
   Output: V2

Usage Information:
   Highest modified vector register: V9
   Lowest scalar register modified: R11
   Vector Mask Modified: Yes
   Approximate cycle count: 520

5.2.32  C_C_POWER (single precision)

Description: Complex to Complex Power

Registers:
   Input: V2 V4 V6 V8
   Output: V2 V4

Usage Information:
   Highest modified vector register: V13
   Lowest scalar register modified: R11
   Vector Mask Modified: Yes
   Approximate cycle count: 3544
5.2.33 C.IC_POWER (single precision)

Description: Complex to Integer Complex Power

Registers:
- Input: V2 V4 V6
- Output: V2 V4

Usage Information:
- Highest modified vector register: V11
- No scalar registers are modified.
- Vector Mask Modified: Yes
- Approximate cycle count: 460

Restrictions: Integer argument must be the same for all nodes.

5.2.34 C.I_POWER (single precision)

Description: Complex to Integer Power

Registers:
- Input: V2 V4 V6
- Output: V2 V4

Usage Information:
- Highest modified vector register: V13
- Lowest scalar register modified: R15
- Vector Mask Modified: Yes
- Approximate cycle count: 5932

5.2.35 C.K_POWER (single precision)

Description: Complex to Double Precision Integer Power

Registers:
- Input: V2 V4 V6
- Output: V2 V4

Usage Information:
- Highest modified vector register: V13
- Lowest scalar register modified: R15
- Vector Mask Modified: Yes
- Approximate cycle count: 11000
5.2.36  D_D_POWER (double precision)

Description: Double to Double Power

Registers:
  Input: V2 V4
  Output: V2

Usage Information:
  Highest modified vector register: V11
  Lowest scalar register modified: R11
  Vector Mask Modified: Yes
  Approximate cycle count: 1192

5.2.37  D_IC_POWER (double precision)

Description: Double to Integer Complex Power

Registers:
  Input: V2 V4
  Output: V2

Usage Information:
  Highest modified vector register: V5
  No scalar registers are modified.
  Vector Mask Modified: Yes
  Approximate cycle count: 140

Restrictions: Integer argument must be the same for all nodes.

5.2.38  D_I_POWER (double precision)

Description: Double to Integer Power

Registers:
  Input: V2 V4
  Output: V2

Usage Information:
  Highest modified vector register: V7
  Lowest scalar register modified: R15
  Vector Mask Modified: Yes
  Approximate cycle count: 2160
5.2.39 D_K_POWER (double precision)

Description: Double to Double Precision Integer Power

Registers:
  Input: V2 V4
  Output: V2

Usage Information:
  Highest modified vector register: V9
  Lowest scalar register modified: R15
  Vector Mask Modified: Yes
  Approximate cycle count: 4320

5.2.40 F_F_POWER (single precision)

Description: Float to Float Power

Registers:
  Input: V2 V4
  Output: V2

Usage Information:
  Highest modified vector register: V11
  Lowest scalar register modified: R11
  Vector Mask Modified: Yes
  Approximate cycle count: 1032

5.2.41 F_IC_POWER (single precision)

Description: Float to Integer Complex Power

Registers:
  Input: V2 V4
  Output: V2

Usage Information:
  Highest modified vector register: V5
  No scalar registers are modified.
  Vector Mask Modified: Yes
  Approximate cycle count: 150

Restrictions: Integer argument must be the same for all nodes.
5.2.42  F_I_POWER (single precision)

Description: Float to Integer Power

Registers:
   Input: V2 V4
   Output: V2

Usage Information:
   Highest modified vector register: V7
   Lowest scalar register modified: R15
   Vector Mask Modified: Yes
   Approximate cycle count: 2176

5.2.43  F_K_POWER (single precision)

Description: Float to Double Precision Integer Power

Registers:
   Input: V2 V4
   Output: V2

Usage Information:
   Highest modified vector register: V9
   Lowest scalar register modified: R15
   Vector Mask Modified: Yes
   Approximate cycle count: 4300

5.2.44  I_I_POWER (single precision)

Description: Integer to Integer Power

Registers:
   Input: V2 V4
   Output: V2

Usage Information:
   Highest modified vector register: V5
   No scalar registers are modified.
   Vector Mask Modified: Yes
   Approximate cycle count: 480
5.2.45 Z.IC_POWER (double precision)

Description: Unsigned Integer to Integer Complex Power

Registers:
Input: V2 V4 V6
Output: V2 V4

Usage Information:
Highest modified vector register: V11
No scalar registers are modified.
Vector Mask Modified: Yes
Approximate cycle count: 450

Restrictions: Integer argument must be the same for all nodes.

5.2.46 Z.I POWER (double precision)

Description: Unsigned Integer to Integer Power

Registers:
Input: V2 V4 V6
Output: V2 V4

Usage Information:
Highest modified vector register: V13
Lowest scalar register modified: R15
Vector Mask Modified: Yes
Approximate cycle count: 5360

5.2.47 Z.Z_POWER (double precision)

Description: Unsigned Integer to Unsigned Integer Power

Registers:
Input: V2 V4 V6 V8
Output: V2 V4

Usage Information:
Highest modified vector register: V15
Lowest scalar register modified: R11
Vector Mask Modified: Yes
Approximate cycle count: 3416
5.2.48  **Z_K_POWER (double precision)**

**Description:** Unsigned Integer to Double Precision Integer Power

**Registers:**
- Input: V2 V4 V6
- Output: V2 V4

**Usage Information:**
- Highest modified vector register: V13
- Lowest scalar register modified: R15
- Vector Mask Modified: Yes
- Approximate cycle count: 10600

5.2.49  **K_K_POWER (double precision)**

**Description:** Double Precision Integer Power

**Registers:**
- Input: V2 V4
- Output: V2

**Usage Information:**
- Highest modified vector register: V9
- No scalar registers are modified.
- Vector Mask Modified: Yes
- Approximate cycle count: 512

5.2.50  **ISHFT (single precision)**

**Description:** Integer Arithmetic Shift

**Registers:**
- Input: V2 V4
- Output: V2

**Usage Information:**
- Highest modified vector register: V4
- No scalar registers are modified.
- Vector Mask Modified: Yes
- Approximate cycle count: 64
5.2.51 KSHFT (double precision)

Description: Double Precision Integer Arithmetic Shift

Registers:
- Input: V2 V4
- Output: V2

Usage Information:
- Highest modified vector register: V5
- No scalar registers are modified.
- Vector Mask Modified: Yes
- Approximate cycle count: 64

5.2.52 ISHFTC (single precision)

Description: Integer Logical Shift

Registers:
- Input: V2 V4
- Output: V2

Usage Information:
- Highest modified vector register: V6
- No scalar registers are modified.
- Vector Mask Modified: Yes
- Approximate cycle count: 192

5.2.53 KSHFTC (double precision)

Description: Double Precision Integer Logical Shift

Registers:
- Input: V2 V4
- Output: V2

Usage Information:
- Highest modified vector register: V11
- Lowest scalar register modified: R14
- Vector Mask Modified: Yes
- Approximate cycle count: 192
5.2.54 ASIN (single precision)

Description: Trigonometric Arcsine in radians

Registers:
Input: V2
Output: V2

Usage Information:
Highest modified vector register: V9
Lowest scalar register modified: R15
Vector Mask Modified: Yes
Approximate cycle count: 392

5.2.55 CDSIN (double precision)

Description: Trigonometric Sine

Registers:
Input: V2 V4
Output: V2 V4

Usage Information:
Highest modified vector register: V15
Lowest scalar register modified: R14
Vector Mask Modified: Yes
Approximate cycle count: 2336

5.2.56 CSIN (single precision)

Description: Trigonometric Sine

Registers:
Input: V2 V4
Output: V2 V4

Usage Information:
Highest modified vector register: V13
Lowest scalar register modified: R14
Vector Mask Modified: Yes
Approximate cycle count: 1600
5.2.57 DASIN (double precision)

Description: Trigonometric Arcsine in radians

Registers:
  Input: V2
  Output: V2

Usage Information:
  Highest modified vector register: V9
  Lowest scalar register modified: R15
  Vector Mask Modified: Yes
  Approximate cycle count: 408

5.2.58 DSIN (double precision)

Description: Trigonometric Sine

Registers:
  Input: V2
  Output: V2

Usage Information:
  Highest modified vector register: V11
  Lowest scalar register modified: R15
  Vector Mask Modified: Yes
  Approximate cycle count: 704

5.2.59 DSINH (double precision)

Description: Trigonometric Hyperbolic Sine

Registers:
  Input: V2
  Output: V2

Usage Information:
  Highest modified vector register: V9
  Lowest scalar register modified: R14
  Vector Mask Modified: Yes
  Approximate cycle count: 800
5.2.60  **SIN (single precision)**

**Description:** Trigonometric Sine

**Registers:**
- Input: V2
- Output: V2

**Usage Information:**
- Highest modified vector register: V9
- Lowest scalar register modified: R15
- Vector Mask Modified: Yes
- Approximate cycle count: 432

5.2.61  **SINH (single precision)**

**Description:** Trigonometric Hyperbolic Sine

**Registers:**
- Input: V2
- Output: V2

**Usage Information:**
- Highest modified vector register: V9
- Lowest scalar register modified: R14
- Vector Mask Modified: Yes
- Approximate cycle count: 608

5.2.62  **CDSQRT (double precision)**

**Description:** Squareroot

**Registers:**
- Input: V2 V4
- Output: V2 V4

**Usage Information:**
- Highest modified vector register: V7
- No scalar registers are modified.
- Vector Mask Modified: Yes
- Approximate cycle count: 440
5.2.63  CSQRT (single precision)

Description: Squareroot

Registers:
  Input: V2 V4
  Output: V2 V4

Usage Information:
  Highest modified vector register: V7
  No scalar registers are modified.
  Vector Mask Modified: Yes
  Approximate cycle count: 488

5.2.64  ATAN (single precision)

Description: Trigonometric Arctangent (1-argument form) in radians

Registers:
  Input: V2
  Output: V2

Usage Information:
  Highest modified vector register: V7
  Lowest scalar register modified: R11
  Vector Mask Modified: Yes
  Approximate cycle count: 512

5.2.65  ATAN2 (single precision)

Description: Trigonometric Arctangent (2-argument form) in radians

Registers:
  Input: V2
  Output: V2 V4

Usage Information:
  Highest modified vector register: V7
  Lowest scalar register modified: R11
  Vector Mask Modified: Yes
  Approximate cycle count: 496
5.2.66 DATAN (double precision)

**Description:** Trigonometric Arctangent (1-argument form) in radians

**Registers:**
- Input: V2
- Output: V2

**Usage Information:**
- Highest modified vector register: V7
- Lowest scalar register modified: R11
- Vector Mask Modified: Yes
- Approximate cycle count: 528

5.2.67 DATAN2 (double precision)

**Description:** Trigonometric Arctangent (2-argument form) in radians

**Registers:**
- Input: V2 V4
- Output: V2

**Usage Information:**
- Highest modified vector register: V7
- Lowest scalar register modified: R11
- Vector Mask Modified: Yes
- Approximate cycle count: 512

5.2.68 DTAN (double precision)

**Description:** Trigonometric Tangent

**Registers:**
- Input: V2
- Output: V2

**Usage Information:**
- Highest modified vector register: V11
- Lowest scalar register modified: R15
- Vector Mask Modified: Yes
- Approximate cycle count: 792
5.2.69 DTANH (double precision)

Description: Trigonometric Hyperbolic Tangent

Registers:
   Input: V2
   Output: V2

Usage Information:
   Highest modified vector register: V11
   Lowest scalar register modified: R14
   Vector Mask Modified: Yes
   Approximate cycle count: 952

5.2.70 TAN (single precision)

Description: Trigonometric Tangent

Registers:
   Input: V2
   Output: V2

Usage Information:
   Highest modified vector register: V9
   Lowest scalar register modified: R15
   Vector Mask Modified: Yes
   Approximate cycle count: 536

5.2.71 TANH (single precision)

Description: Trigonometric Hyperbolic Tangent

Registers:
   Input: V2
   Output: V2

Usage Information:
   Highest modified vector register: V9
   Lowest scalar register modified: R14
   Vector Mask Modified: Yes
   Approximate cycle count: 776