CCM File Format Specification

The structure for the application-neutral portion of the STAR file format is described below. The CCM file is layered on top of the ADF file format. The semantics of the format is basically the same as the original specification – organizational nodes are in **Bold** type and are suffixed by an **M** indicating that multiple occurrences of this node are possible or an **S** indicating only one occurrence is possible. All plain-faced nodes represent actual ADF nodes. Single values are simply identified by their type, realizing that this corresponds to a one element array in ADF. Arrays are specified by their type, dimension, and size. Blue text indicates nodes that are optional in some sense and thus won't be hard-coded into the API. Only those nodes that are both critical for multiple applications and are sufficiently generic that they are highly unlikely to change as new modeling emerges will be hard coded into the API. Leaf nodes marked as optional (ProstarFaceId's for instance) should use the CCMIOWriteOpt(|1|2)(i|f|d) API functions. Non-leaf optional nodes that are specific to one solver's needs, such as the RestartInfo node, may need to use the low-level API.

1. **General: S**. A node to store global data for the file.
   1.1. Version (int)
   1.2. Title (string)
2. **States: S.** A node to store a collection of states, each of which can consist of a mesh, solution, and model description node. Support for parallel decomposition is built into this node.
   2.1. **(State): M** (One node for each state in the file). The title for each state node must be a unique name or number describing the state.
      2.1.1. Label (string). String to describe the state more fully than the title permits.
      2.1.2. ProblemDescriptionId (int) The id of the problem description for this state. This id refers to a specific **ProblemDescription** subnode below. If this node is non-existent, it is assumed that there is no problem description associated with this state.
      2.1.3. **Processor-#: M** (One node for each processor in the state)
         2.1.3.1. VerticesFile (string). The file containing the vertices. If this node is blank or non-existent, the vertices is read from the current file.
         2.1.3.2. VerticesId (int). The id of the vertices for this processor. This id refers to a specific **Vertices** subnode of the **Mesh** node below. If this node is non-existent, it is assumed that there is no mesh associated with this processor and state.
         2.1.3.3. TopologyFile (string). The file containing the topology. If this node is blank or non-existent, the topology is read from the current file.
         2.1.3.4. TopologyId (int). The id of the mesh topology for this processor. If this node is non-existent, it is assumed that there is no mesh associated with this processor and state

2.1.3.5. InitialFieldFile (string). The file containing the initial field. If this node is blank or non-existent, the initial field is read from the current file.

2.1.3.6. InitialFieldId (int). The id of the field specified in the **Fields** node to be used as the initial field in this processor. If this node is non-existent, no initial field is specified.

2.1.3.7. SolutionFile (string). The file containing the solution. If this node is blank or non-existent, the solution is read from the current file.

2.1.3.8. SolutionId (int). The id of the field specified in the **Field** node which stores solution data for this processor. If this node is non-existent, no solution data is specified.

2.1.3.9. **LagrangianData: S**.

2.1.3.9.1. **LagrangianData-#: M**.

2.1.3.9.1.1. Label (string). A string label used to identify this data.

2.1.3.9.1.2. PositionVectorsFile (string). The file containing the Lagrangian data. If this node is blank or non-existant, the Lagrangian data is read from the current file.

2.1.3.9.1.3. PositionVectorsId (int). The id of the position vectors. This id is a reference to a **Vertices** node. (The **VertexMapId** node is the map of particle ids.)

2.1.3.9.1.4. SolutionFile (string). The file containing the Lagrangian data. If this node is blank or non-existant, the Lagrangian data is read from the current file.

2.1.3.9.1.5. SolutionId (int). The id of a **FieldSet** node.

3. **Maps: S,** A node to store id maps to be used in subsequent nodes. These maps are basically just a list of ids which can then be used to index an array in mesh or post data. The API will provide methods to create a new map (which will return a unique id for the map), write to a map, and access the map if needed. Most of the actual map lookups should be handled internal to the API (during reading of a mapped node, for instance), with only the id of the map being passed around if needed.

   3.1.1. **Map-#: M**. A numbered map. The map is referred to by a unique id number managed by the API.

   3.1.1.1. Label (string). A string label used to identify this map.

   3.1.1.2. MaximumMappedId (int). The maximum id specified in the map. This can be used to quickly determine the validity of the data to be read to ensure no array overflows occur.

   3.1.1.3. IdMap (int array, 1, map size). The map from local to global ids. When used for cell data below, for instance, the Nth element of the cell data array will correspond to cell data at the IdMap[N]th cell.

4. **Meshes: S**. A node to store any number of meshes. Each **Vertices** and **Topology** node contained in this node can be either a physical node in this file or an ADF link to a node in another file. In this way, information shared between meshes (such as halo cell information) can be properly referenced. For a parallel case in which the mesh is split into different files, each file should have subnodes for all other meshes with the same id representing the same partition of mesh in all files. In each file, however, all subnodes will be ADF links except for the one containing the actual data for that file.

4.1. **Vertices-#: M** (One node for each set of vertex coordinates stored in the file). This node stores the vertex coordinate information. It is separated from Topology so moving mesh cases can reuse common topology as much as possible.

4.1.1.1. Label (string). A string label used to identify these vertices.

4.1.1.2. Dimension (int). The dimension of the vertices. Valid values are 2 or 3.

4.1.1.3. ScaleFactor (float). Conversion from model units to SI. The specified vertices are unscaled.

4.1.1.4. Coordinates (float/double array, 2, [2,3], # vertices). Vertex coordinates.

4.1.1.5. VertexMapId (int). An id from the **Maps** node used to associate the above coordinates with global vertex ids.

4.1.1.6. VertexDependency (int array, 1, 7 * # vertex dependencies) { dependent vertex, master vert 1, master vert 2, slave vert 1, slave vert 2, flag, cyclic region number}. Used to move vertices generated by cp match and cyclic face resolution that the user doesn't know about. For cyclic boundaries, slave vert 2 may be -1 if a vertex point on one side exactly matches an edge on the other side and the dependent vertex is on the slave. The flag is used to determine which pair of vertices from cyclic boundaries is transformed: 0 (the master), 1 (the slave). Regular couples will be set to -1. The final entry is the region id for cyclic boundaries, or -1 if not a cyclic boundary.

4.2. **FaceBasedTopology-#: M** (One node for each face-based topology stored in the file). This nodes stores the face-based topology information for a mesh.

4.2.1. Label (string): A string label used to identify this mesh topology.

4.2.2. InternalFaces: S

4.2.2.1. NumFaces (int). Number of internal faces.

4.2.2.2. Vertex (int array, 1, (undetermined)). {number of vertices, vertex list, …}. The list of internal face vertices, specified in terms of global vertex numbers.

4.2.2.3. FaceMapId (int). An id from the **Maps** node used to associate the internal faces with global face ids.

4.2.2.4. Cells (int array, 2, 2, # internal faces). The two cells neighboring each face, specified in terms of global cell ids. Face is specified such that the right handed normal points from cell(1,id) to cell(2, id).

4.2.2.5. ProstarFaceId (int array, 2, 2, # internal faces) The prostar face id of each face relative to its two cells.

4.2.3. NumBoundaryTypes (int). The number of sets of boundary faces specified in the next node.

4.2.4. **BoundaryFaces-#: M.** This node occurs any number of times and is used to specify boundary faces grouped by id. These Ids will typically correspond to different boundary regions (collections of boundary faces which share a common boundary condition). These nodes are differentiated by including the boundary id in the node name.

4.2.4.1. NumFaces (int). Number of boundary faces in this group.

4.2.4.2. Vertex (int array, 1, (undetermined)). {number of vertices, vertex list, …}. The list of boundary face vertices, specified in terms of global vertex numbers.

4.2.4.3. FaceMapId (int). An id from the **Maps** node used to associate the boundary faces with global face ids.

4.2.4.4. Cells (int array, 1, # boundary faces). The cell neighboring each face, specified using global cell ids. Face is specified such that the right handed normal points from cell(1,id) to outside.

4.2.4.5. ProstarFaceId (int array, 1, # boundary faces) The prostar face id of each face relative to its cell.

4.2.4.6. GlobalBoundId (int array, 1, # boundary faces). This is a global number that identifies each boundary face. When this file is generated from prostar, this field will contain the prostar boundary id.

4.2.4.7. RadiationPatches (int array, 1, # boundary faces). This is needed for radiation calculations.

4.2.4.8. ProstarRegionNumber (int). Gives the prostar region number that these faces belong to. Negative values are from baffles that are not part of a region; the value is the negative of the prostar cell type.

4.2.5. Cells: S

4.2.5.1. NumCells (int). The number of cells in the topology.

4.2.5.2. CellType (int array, 1, # cells). The cell type of each cell. This is an integer flag stored for each cell which can later be used to group cells together to form subdomains and domains.

4.2.5.3. CellMapId (int). An id from the **Maps** node used to associate all cell data in this node (CellType, ParentCellId, CellTopologyType) with global cell ids.

4.2.5.4. ParentCellId (int array, 1, # cells). Parent Cell id.

4.2.5.5. CellTopologyType (int array, 1, # cells). The topology type of each cell (see Appendix A.1 for valid values).

4.2.5.6. HaloCells (int array, 1, # halo cells). Halo Cell List.

4.2.5.7. OtherProcessor (int array, 1, # halo cells). Processor number of other processor. OtherCell (int array, 1, # halo cells). Other Cell id.

4.2.6. **Interfaces: S** Interfaces represent a pairing of boundary faces across an interface like a cyclic boundary or internal baffle.

4.2.6.1. FaceIds (int array, 2, 2, # interfaces). The face->face pairing across all interfaces. Each row of this array contains two global face ids of boundary faces from a **BoundaryFaces** node. If the interface is a baffle, the first item will be the positive baffle face and the second item will be the negative baffle face.

4.2.6.2. Types (int array, 1, # interfaces). Specifies what the type of interface. 1: baffle, 2: cyclic, 3: double boundary

4.2.6.3. ProstarBaffles (int array, 1, 4 * # baffles) { baffleIdx, face1, face2, cellType } Stores the definitions of prostar's baffles.

5. **ProblemDescriptions: S.** A node to add additional information such as string labels and types to various Ids described in the mesh.

5.1. **ProblemDescription-**#: (One node for each set of problem description information in the file).

5.1.1. **CellType-#: M.** (One node for each cell type id listed in node 4.2.4.2). A node to describe each cell type.

5.1.1.1. Label (string). A string label describing this cell type. Could be blank and is not guaranteed to be unique across all cell types.

5.1.1.2. MaterialType (string). The material type of this cell type. Currently supported values are "solid" and "fluid", but this field is left as a string instead of an enum so it can easily be extended later without reopening the API.

5.1.1.3. MaterialId (int). A material id for this cell type. Although the actual material properties for this material id are stored elsewhere, this id can be used by solvers to merge cell types into common regions if they share the same material id.

5.1.1.4. PorosityId (int). A porosity id for this cell type. See note on MaterialId node.

5.1.1.5. SpinId (int). A spin id for this cell type for MRF cases. See note on MaterialId node.

5.1.1.6. GroupId (int). An additional integer flag associated with this cell type. If a solver is automatically merging cell types together to form regions based on common material types and Ids, a user may need to have a method to prevent two cell types from being merged if he wants them to stay as two separate regions. Thus, cell types with different group ids should not be merged even if all other properties are the same.

5.1.1.7. Radiation (int). 1 if the cell uses radiation, 0 otherwise.

5.1.2. **BoundaryRegion-#: M.** A node to describe each boundary region based on the id used in node 4.2.3.1.

5.1.2.1. Label (string). A string label describing this boundary region. Could be blank and is not guaranteed to be unique across all cell types.

5.1.2.2. BoundaryType (string). A string describing the type of this boundary region. Currently supported values are listed in Appendix A.3.

5.1.2.3. BoundaryName (string). The name of the boundary.

5.1.2.4. CyclicCoordinateSystemType (int). This is the same as used in pro-STAR. 1 (Cartesian), 2 (cylindrical), 3 (spherical), and 4 (toroidal).

5.1.2.5. CyclicTranslationMatrix (double array, 1, 3) Coordinate system translation matrix.

5.1.2.6. CyclicRotationMatrix (double array, 3, 3). Coordinate system rotation matrix

5.1.2.7. CyclicBoundaryOffset (double array, 1, 3).

5.1.3. **ModelConstants: S**. Values that are constant to the mesh. (e.g. gravity)

5.1.3.1.1. *Name*: (float).

6. **FieldSets: S**. A node to store any number of sets of field data, which could represent either initial field or solution data.

6.1. **FieldSet-#: M** (One node for each set of field data stored in the file). Each set of data should have a unique identifying number used in the nodes title.

6.1.1. SolutionTitle (string). A title describing this solution data set.

6.1.2. **Phase-#: M** (One for each phase in the problem)

6.1.2.1. **Fields: S.** A node to store all the fields. All fields are grouped into subnodes of this node to avoid name conflicts and simplify node name parsing.

6.1.2.1.1. **(name): M** (One for each data field in the problem). A list of data fields. The title of each field node must be a unique string describing the field. See Appendix A.2 for valid field names.

6.1.2.1.1.1. FieldType (enum). The type of the field. Valid enumerations are *Scalar, Vector, Tensor*.

6.1.2.1.1.2. Short name (string). 8 character or less string, unique among all fields in this FieldSet.

6.1.2.1.1.3. Units (string). String describing the units.

6.1.2.1.1.4. **Data-#: M** (One node for each chunk of data in this field).

6.1.2.1.1.4.1. DataType: (enum). A flag identifying which type of data this is. Valid enumerations are *Cell, Face, Vertex*.

6.1.2.1.1.4.2. Data: (int/float/double array, 1, # values). Data array. (Ints are only used by Lagrangian data). This node is only used if the type is *Scalar* and only if the data is not constant.

6.1.2.1.1.4.3. **Component-#: M** (string). Name of node includes parenthesis. Node only exists for vectors and tensors and contains the name of the scalar field containing the values for this component. # ranges from 0 – 2, inclusive for vectors, 0 – 8, inclusive for tensors. If one component is specified, all components should be specified.

6.1.2.1.1.4.4. ConstantData (int/float/double). Replaces **Data** node; specifies that all values are a particular constant value.

6.1.2.1.1.4.5. DataMapId: (int) An id from the **Maps** node used to associate the data with its proper objects (cells, faces, or vertices).

6.1.3. **RestartInfo: S**. Contains solver-specific information.

6.1.3.1. SolverName: (string). Name of the solver that created this restart info.

6.1.3.2. Iteration: (int). Iteration number.

6.1.3.3. Time: (float). Current time (in seconds, unless otherwise specified)

6.1.3.4. TimeUnits: (string). Units of time.

6.1.3.5. Angle: (float). Beginning angle (in radians).

6.1.3.6. **RestartData: S.** Contains the actual data. Actual contents varies by solver.

6.1.3.7. **ReferenceData: S**. Contains information about reference values.

6.1.3.7.1. Pressure: (float array, 2, # of prostar cell-types). An array of { prostar-cell-type, reference pressure }. The cell-type is really an int, but will be stored internally as a float (assuming cell-type count stays reasonably small).

6.1.3.7.2. Temperature: (float array, 2, # of prostar cell-types).
Reference temperature in Kelvin.
6.1.3.7.3. Density: (float array, 2, # of prostar cell-types).
6.1.3.7.4. *Name*: (float array, 2, # of prostar cell-types).

7. **ProstarSets: S**. A node to store any prostar set.
   7.1. **(name): S**. Contains the prostar set by this name. ADF limits the names to 32 characters.
      7.1.1. CellSet: (int array, 1, # of cells). Contains the cells in this set.
      7.1.2. VertexSet: (int array, 1, # of cells). Contains the vertices in this set.
      7.1.3. BoundarySet: (int array, 1, # of cells). Contains the boundaries in this set.
      7.1.4. BlockSet: (int array, 1, # of cells). Contains the blocks in this set.
      7.1.5. SplineSet: (int array, 1, # of cells). Contains the splines in this set.
      7.1.6. CoupleSet: (int array, 1, # of cells). Contains the couples in this set.

A.1 Cell Topologies

The following values are allowed for the cellTopologyType array in the **Topology->Cells** node.

1	Point
2	Line
28	Line with 1 extra node
29	Line with 2 extra nodes
3	Shell
4	Shell with midside vertices
11	Hexahedron
12	Prism
13	Tetrahedron
14	Pyramid
255	Polyhedron
21	Hexahedron with midside vertices
22	Prism with midside vertices
23	Tetrahedron with midside vertices
24	Pyramid with midside vertices

A.2 Field Names:

The following strings define the basic field names used in the **Fields** node. Non-standard fields can be referred to by any appropriate name, but the exact name below should be used for any standard field.

pressure
velocity
mass flux
displacement
molecular viscosity
turbulent viscosity
density
temperature
turbulence - ke

turbulence - epsilon
stress

A.3 Boundary Condition Names:

The following strings define the allowed boundary type names used in the **BoundaryRegion** node (5.1.2.2).

inlet
outlet
symmetry
wall
cyclic
stagnation
pressure
baffle
freestream

A.4 Discussion About Vertex Dependencies

This section describes the information needed by STAR-CCM to move vertices created by CP matches and cyclic boundaries that the creator of the mesh motion routine doesn't know about. This comes about because both routines create vertices when making each face set conformal. To move these vertices, STAR-CCM is going to recreate the intersection calculation to avoid (for as long as possible) generating bad cell faces. So enough information needs to be stored for this to happen. For CP matches it is relatively straight forward. Cyclic boundaries are a bit more problem since you need the coordinate system information they are defined in.

The master/slave pair designation is important so the STAR-CCM can do the same thing that pro-STAR does when generating the CCM file. This is what is needed:

1. Information needed by each vertex.
   1.1. Pair of vertices for describing line segment from master face. (2 int)
   1.2. Pair of vertices for describing line segment from slave face. (2 int) NOTE: For cyclic boundaries, there will be occasions when a vertex point on one side exactly matches an edge on the other side. If the vertex is on the master, you don't need to add any entry. If the vertex is on the slave, you will need to put the two master vertices out and the one slave vertex out in position 0 of this field. Position 1 MUST be -1.
   1.3. Flag indicating which pair of vertices get transformed. 0 => the master pair, 1 => the slave pair. This is for cyclic boundaries only. (1 int)
   1.4. Region Id indicates the region id for the side of the cyclic pair that gets transformed to the other cyclic boundary for processing. This is for cyclic boundaries only. (1 int)
2. Additional information needed for cyclic boundaries.
   2.1. Region Id corresponding to the Region Id above (this is encoded in the name of the BoundaryRegion-# node)
   2.2. Coordinate system type. This is the same as used in pro-STAR. (1 int) 1 => Cartesian, 2 => Cylindrical, 3 => Spherical, 4 => Toroidal
   2.3. Coordinate system translation matrix. (3 double)
   2.4. Coordinate system rotation matrix. (3x3 double)
   2.5. Cyclic boundary offset. (3 double)

First some terminology. "Master vertex" means a vertex that is part of the vertex set that makes up the master set of faces. Likewise, a "slave vertex" a similar meaning except on the slave side. Statements that master and slave vertices have the same location or are on each other applies when one of the face sets has been transformed onto the other face set using the defined transforms for the cyclic boundary. This does not mean that their final location is the same because of the transformation.

If a master vertex is on a line segment in the slave side, a vertex needs to be added to the slave side. This new slave vertex will depend on only one master vertex. In this case, m0 is the master vertex and m1, s0 and s1 are set to -1.

If a slave vertex is on a line segment in the master side, a vertex needs to be added to the master side. However, the master vertices do not move. So we have a new master vertex that the user doesn't know about and it needs to be between two master vertices. Therefore, we have a master vertex that is dependent on two other master vertices and a slave vertex and a slave vertex that is dependent on the new master vertex. Note that in this case, you are moving the location of a slave vertex that the user has already moved.

If a master vertex is on a slave face, a vertex needs to be added to the slave side. This vertex will have the same location as the master vertex.

If a slave vertex is on a master face, a vertex needs to be added to the master side. This vertex will have the same location as the slave vertex. I know this kind of violates the master doesn't move rule but the interior location of a master cell is not well defined especially if it is warped. The slave vertex seems like as good a place as any to put it.

This brings up another point. The order of these entries is important for the reason given above. Also, a face edge can be divided more then once. In this case, we do not refer back to the original vertices but the vertices that define the new segment, one or more of which would be moved by a previous calculation. Therefore, this list is not necessarily ordered by vertex number but by the order the new vertices were created.

If we look at the current storage scheme, we get this:

```
CP Match segments crossing              Vnew depends on m0,          m1, s0, s1
Cyclic segments crossing                Mnew depends on m0,          m1, s0, s1
                                        Snew depends on Mnew Above, -1, -1, -1
Cyclic master vertex on slave face      Snew depends on m0,          -1, -1, -1
Cyclic slave vertex on master face      Mnew depends on s0,          -1, -1, -1
Cyclic master vertex on slave segment   Snew depends on m0,          -1, -1, -1
Cyclic slave vertex on master segment   Mnew depends on m0,          m1, s0, -1
                                        Sold depends on Mnew Above, -1, -1, -1

Which give us these rules:
  m0, m1, s0, s1: segment intersection that is always on master segment.
  m0, m1, s0, -1: New vertex is on segment m0,m1 with location determined by s0.
  m0, -1, -1, -1: New vertex has same location as m0.
```

A.5. ExtendedSize and ExtendedData nodes

The ADF library limits the size of a node to 2 GB. The CCM format allows arbitrarily large nodes, which is accomplished by splitting the node into multiple nodes that are sized within the ADF limit. The CCM library presents this collection of nodes as one large node. The format for a large node is:

1. **Node**: Contains the first (approximately) 2 GB of data. The node dimension contains the size of the data that **Node** actually contains (not the virtual dimension).
   1.1. **ExtendedSize**: An array of the virtual size of **Node**. This is the same dimension information that would be stored in **Node** if ADF nodes could store dimensions that large.
   1.2. **ExtendedData-#**: The next 2 GB of data. As many **ExtendedData-#** nodes are written as necessary.

The actual size limit of the nodes is set inside the CCM library, and need not be 2 GB. Any size <= 2 GB is valid, and the CCM library will determine the actual size automatically.

Changes:

- **12/30/03 (Jester):**
  - Finalized the **Interfaces** node, changing it from red to blue.
  - Removed the **InterfaceData** node from the felds, since interface data can now be recorded on separate faces across the interface.
  - Removed red comments from the **OtherProcessor** node since nobody complained about them.
- **4/22/04 (Prewett):**
  - Added **Label** node to **State**.
  - Changed **MaterialType** (5.1.1.2) to blue.
  - Added **LagrangianData** node.
- **6/24/04 (Prewett):**
  - Added **Phase**, **Short name**, and **Units** nodes to post data.
  - Removed **DropletData** node because the **LangrangianData** renders it unnecessary.
- **8/2/04 (Prewett):**
  - Added restart node.
  - Changed vector post data to be stored as scalar components.
  - Added way to save a constant scalar field.
- **12/3/04 (Prewett):**
  - Added **BoundaryFaces-# -> ProstarRegionNumber** node (4.2.4.8).
  - Removed remaining red comments since no one has mentioned them in a year.
- **4/18/05 (Oaks)**
  - Added vertex dependency information
- **4/18/05 (Prewett)**

    - Added cyclic boundary nodes to the **ProblemDescription/BoundaryRegion**-# nodes. Altered the content of the **Vertices-#/VertexDependency** node.
- **6/16/05 (Prewett)**
    - Moved vertex dependency discussion to appendix and reformatted.
    - Added **BoundaryName** node.
- **6/27/05 (Prewett)**
    - Added the **ProstarSets** and the **ProblemDescription/MonitoringSets** nodes.
- **8/03/05 (Prewett)**
    - Added the interface type (4.2.6.2) and **ProstarBaffle** (4.2.6.3) nodes.
- **8/30/05 (Oaks)**
    - Removed the line: "No attempt to specify the type of interface is provided in the format, rather only the connectivity information is stored" in 4.2.6. It doesn't reflect the addition of Types to the interface data.
    - Added the following condition to the Interface/FaceIds (4.2.6.1). If the interface is baffle, the first item will be the positive baffle face and the second item will be the negative baffle face.
- **5/05/06 (Prewett)**
    - Added appendix A.5 documenting the format used to store nodes larger than 2 GB.