```
    TimeAccurate,
    NonTimeAccurate ) ;
```

`DataClass` describes the global default for the class of data contained in the CGNS database. If the CGNS database contains dimensional data (e.g., velocity with units of m/s), `DimensionalUnits` may be used to describe the system of units employed.

`FlowEquationSet` contains a description of the governing flow equations associated with the entire CGNS database. This structure contains information on the general class of governing equations (e.g., Euler or Navier-Stokes), equation sets required for closure, including turbulence modeling and equations of state, and constants associated with the equations.

`DataClass`, `DimensionalUnits`, `ReferenceState` and `FlowEquationSet` have special function in the CGNS hierarchy. They are globally applicable throughout the database, but their precedence may be superseded by local entities (e.g., within a given zone). The scope of these entities and the rules for determining precedence are treated in Section 6.4.

Globally relevant convergence history information is contained in `GlobalConvergenceHistory`. This convergence information includes total configuration forces, moments, and global residual and solution-change norms taken over all the zones.

Miscellaneous global data may be contained in the `IntegralData_t` list. Candidates for inclusion here are global forces and moments.

The `Family_t` data structure, defined in Section 12.6, is used to record geometry reference data. It may also include boundary conditions linked to geometry patches. For the purpose of defining material properties, families may also be defined for groups of elements. The family-mesh association is defined under the `Zone_t` and `BC_t` data structures by specifying the family name corresponding to a zone or a boundary patch.

The `UserDefinedData_t` data structure allows arbitrary user-defined data to be stored in `Descriptor_t` and `DataArray_t` children without the restrictions or implicit meanings imposed on these node types at other node locations.

6.3 Zone Structure Definition: `Zone_t`

The `Zone_t` structure contains all information pertinent to an individual zone. This information includes the zone type, the number of cells and vertices making up the grid in that zone, the physical coordinates of the grid vertices, grid motion information, the family, the flow solution, zone interface connectivity, boundary conditions, and zonal convergence history data. Zonal data may be recorded at multiple time steps or iterations. In addition, this structure contains a reference state, a set of flow equations and dimensional units that are all unique to the zone. For unstructured zones, the element connectivity may also be recorded.

```
  ZoneType_t := Enumeration(
    Null,
    Structured,
    Unstructured,
  UserDefined ) ;

Zone_t< int CellDimension, int PhysicalDimension > :=
  {
  List( Descriptor_t Descriptor1 ... DescriptorN ) ;                         (o)

  ZoneType_t ZoneType ;                                                      (r)

  int[IndexDimension] VertexSize ;                                           (r)
  int[IndexDimension] CellSize ;                                             (r)
  int[IndexDimension] VertexSizeBoundary ;                                   (o/d)

  List( GridCoordinates_t<IndexDimension, VertexSize>
        GridCoordinates, MovedGrid1 ... MovedGridN ) ;                        (o)

  List( Elements_t Elements1 ... ElementsN ) ;                               (o)

  List( RigidGridMotion_t RigidGridMotion1 ... RigidGridMotionN ) ;          (o)

  List( ArbitraryGridMotion_t
        ArbitraryGridMotion1 ... ArbitraryGridMotionN ) ;                    (o)

  FamilyName_t FamilyName ;                                                  (o)

  List( FlowSolution_t<CellDimension, IndexDimension, VertexSize, CellSize>
        FlowSolution1 ... FlowSolutionN ) ;                                  (o)

  List( DiscreteData_t<CellDimension, IndexDimension, VertexSize, CellSize>
        DiscreteData1 ... DiscreteDataN ) ;                                  (o)

  List( IntegralData_t IntegralData1 ... IntegralDataN ) ;                   (o)

  ZoneGridConnectivity_t<IndexDimension, CellDimension>
     ZoneGridConnectivity ;                                                  (o)

  ZoneBC_t<CellDimension, IndexDimension, PhysicalDimension> ZoneBC ;        (o)

  ZoneIterativeData_t<NumberOfSteps> ZoneIterativeData ;                     (o)

  ReferenceState_t ReferenceState ;                                          (o)

  RotatingCoordinates_t RotatingCoordinates ;                                (o)

  DataClass_t DataClass ;                                                    (o)
  DimensionalUnits_t DimensionalUnits ;                                  (o)

  FlowEquationSet_t<CellDimension> FlowEquationSet ;                     (o)

  ConvergenceHistory_t ZoneConvergenceHistory ;                          (o)

  List( UserDefinedData_t UserDefinedData1 ... UserDefinedDataN ) ;      (o)

  int Ordinal ;                                                          (o)
  } ;
```

Notes

1. Default names for the `Descriptor_t`, `Elements_t`, `RigidGridMotion_t`, `ArbitraryGridMotion_t`, `FlowSolution_t`, `DiscreteData_t`, `IntegralData_t`, and `UserDefinedData_t` lists are as shown; users may choose other legitimate names. Legitimate names must be unique within a given instance of `Zone_t` and shall not include the names `DataClass`, `DimensionalUnits`, `FamilyName`, `FlowEquationSet`, `GridCoordinates`, `Ordinal`, `ReferenceState`, `RotatingCoordinates`, `ZoneBC`, `ZoneConvergenceHistory`, `ZoneGridConnectivity`, `ZoneIterativeData`, or `ZoneType`.
2. The original grid coordinates should have the name `GridCoordinates`. Default names for the remaining entities in the `GridCoordinates_t` list are as shown; users may choose other legitimate names, subject to the restrictions listed in the previous note.
3. `ZoneType`, `VertexSize`, and `CellSize` are the only required fields within the `Zone_t` structure.

`Zone_t` requires the parameters `CellDimension` and `PhysicalDimension`. `CellDimension`, along with the type of zone, determines `IndexDimension`; if the zone type is `Unstructured`, `IndexDimension` = 1, and if the zone type is `Structured`, `IndexDimension` = `CellDimension`. These three structure parameters identify the dimensionality of the grid-size arrays. One or more of them are passed on to the grid coordinates, flow solution, interface connectivity, boundary condition and flow-equation description structures.

`VertexSize` is the number of vertices in each index direction, `and CellSize` is the number of cells in each direction. For example, for structured grids in 3-D, `CellSize = VertexSize - [1,1,1]`, and for unstructured grids in 3-D, `CellSize` is simply the total number of 3-D cells. `VertexSize` is the number of vertices defining "the grid" or the domain (i.e., without rind points); `CellSize` is the number of cells on the interior of the domain. These two grid-size arrays are passed onto the grid-coordinate, flow-solution and discrete-data substructures.

If the nodes are sorted between internal nodes and boundary nodes, then the optional parameter `VertexSizeBoundary` must be set equal to the number of boundary nodes. If the nodes are sorted, the grid coordinate vector must first include the boundary nodes, followed by the internal nodes. By default, `VertexSizeBoundary` equals zero, meaning that the nodes are unsorted. This option is only useful for unstructured zones. For structured zones, `VertexSizeBoundary` always equals 0 in all index directions.

```
    DataArray_t<real, 1, 15> CoordinateX =
      {{
      Data(real, 1, 15) = (x(i), i=1,15) ;
      }} ;

    DataArray_t<real, 1, 15> CoordinateY =
      {{
      Data(real, 1, 15) = (y(i), i=1,15) ;
      }} ;

    DataArray_t<real, 1, 15> CoordinateZ =
      {{
      Data(real, 1, 15) = (z(i), i=1,15) ;
      }} ;
    }} ;
```

7.3 Elements Structure Definition: `Elements_t`

The `Elements_t` data structure is required for unstructured zones, and contains the element connectivity data, the element type, the element range, the parent elements data, and the number of boundary elements.

```
  Elements_t :=
    {
    List( Descriptor_t Descriptor1 ... DescriptorN ) ;                   (o)

    Rind_t<IndexDimension> Rind ;                                        (o/d)

    IndexRange_t ElementRange ;                                          (r)

    int ElementSizeBoundary ;                                            (o/d)

    ElementType_t ElementType ;                                          (r)

    DataArray_t<int, 1, ElementDataSize> ElementConnectivity ;           (r)

    DataArray_t<int, 2, [ElementSize, 2]> ParentElements ;               (o)
    DataArray_t<int, 2, [ElementSize, 2]> ParentElementsPosition ;       (o)

    List( UserDefinedData_t UserDefinedData1 ... UserDefinedDataN ) ;    (o)
    } ;
```

Notes

1. Default names for the `Descriptor_t` and `UserDefinedData_t` lists are as shown; users may choose other legitimate names. Legitimate names must be unique within a given instance of `Elements_t` and shall not include the names `ElementConnectivity`, `ElementRange`, `ParentElements`, `ParentElementsPosition` or `Rind`.
2. `IndexRange_t`, `ElementType_t`, and `ElementConnectivity_t` are the required fields within the `Elements_t` structure. `Rind` has a default if absent; the default is equivalent to having a `Rind` structure whose `RindPlanes` array contains all zeros (see Section 4.8).

`Rind` is an optional field that indicates the number of rind elements included in the elements data. If `Rind` is absent, then the `DataArray_t` structure entities contain only core elements of a zone. If `Rind` is present, it will provide information on the number of rind elements, in addition to the core elements, that are contained in the `DataArray_t` structures.

Note that the usage of rind data with respect to the size of the `DataArray_t` structures is different under `Elements_t` than elsewhere. For example, when rind coordinate data is stored under `GridCoordinates_t`, the parameter `VertexSize` accounts for the core data only. The size of the `DataArray_t` structures containing the grid coordinates is determined by the `DataSize` function, which adds the number of rind planes or points to `VertexSize`. But for the element connectivity, the size of the `DataArray_t` structures containing the connectivity data is just `ElementDataSize`, which depends on `ElementSize`, and includes both the core and rind elements.

`ElementRange` contains the index of the first and last elements defined in `ElementConnectivity`. The elements are indexed with a global numbering system, starting at 1, for all element sections under a given `Zone_t` data structure. The global numbering insures that each element, whether it's a cell, a face, or an edge, is uniquely identified by its number. They are also listed as a continuous list of element numbers within any single element section. Therefore the number of elements in a section is:

```
ElementSize = ElementRange.end - ElementRange.start + 1
```

The element indices are used for the boundary condition and zone connectivity definition.

`ElementSizeBoundary` indicates if the elements are sorted, and how many boundary elements are recorded. By default, `ElementSizeBoundary` is set to zero, indicating that the elements are not sorted. If the elements are sorted, `ElementSizeBoundary` is set to the number of elements at the boundary. Consequently:

```
ElementSizeInterior = ElementSize - ElementSizeBoundary
```

`ElementType_t` is an enumeration of the supported element types:

```
ElementType_t := Enumeration(
   Null, NODE, BAR_2, BAR_3,
   TRI_3, TRI_6, QUAD_4, QUAD_8, QUAD_9,
   TETRA_4, TETRA_10, PYRA_5, PYRA_14,
   PENTA_6, PENTA_15, PENTA_18,
   HEXA_8, HEXA_20, HEXA_27, MIXED, NGON_n, UserDefined );
```

Section 3.3 illustrates the convention for element numbering.

For all element types except type `MIXED`, `ElementConnectivity` contains the list of nodes for each element. If the elements are sorted, then it must first list the connectivity of the boundary elements, then that of the interior elements.

$$
\texttt{ElementConnectivity} = \begin{array}{l}
\texttt{Node1}_1,\ \texttt{Node2}_1,\ \ldots\ \texttt{NodeN}_1, \\
\texttt{Node1}_2,\ \texttt{Node2}_2,\ \ldots\ \texttt{NodeN}_2, \\
\ldots \\
\texttt{Node1}_M,\ \texttt{Node2}_M,\ \ldots\ \texttt{NodeN}_M
\end{array}
$$

When the section `ElementType` is `MIXED`, the data array `ElementConnectivity` contains one extra integer per element, to hold each individual element type:

$$
\texttt{ElementConnectivity} = \begin{array}{l}
\texttt{Etype}_1,\ \texttt{Node1}_1,\ \texttt{Node2}_1,\ \ldots\ \texttt{NodeN}_1, \\
\texttt{Etype}_2,\ \texttt{Node1}_2,\ \texttt{Node2}_2,\ \ldots\ \texttt{NodeN}_2, \\
\ldots \\
\texttt{Etype}_M,\ \texttt{Node1}_M,\ \texttt{Node2}_M,\ \ldots\ \texttt{NodeN}_M
\end{array}
$$

`ElementDataSize` indicates the size (number of integers) of the array `ElementConnectivity`. For all element types except type `MIXED`, the `ElementDataSize` is given by:

```
ElementDataSize = ElementSize * NPE[ElementType]
```

In the case of `MIXED` element section, `ElementDataSize` is given by:

$$
\texttt{ElementDataSize} = \sum_{n=start}^{end} (\texttt{NPE[ElementType}_n\texttt{]} + 1)
$$

`NPE[ElementType]` is a function returning the number of nodes for the given `ElementType`. For example, `NPE[HEXA_8]=8`.

For face elements in 3–D, or bar element in 2–D, additonal data may be provided for each element in `ParentElements` and `ParentElementsPosition`. The element numbers of the two adjacent cells for each face are given in `ParentElements`. The corresponding canonical positions of the face in the two parent cells is given in `ParentElementsPosition`; these canonical face positions are defined in Section 3.3. For faces on the boundary of the domain, the second parent is set to zero.

`NGON_n` is used to express a polygon of n nodes. In order to record the number of nodes of any ngons, the `ElementType` must be set to `NGON_n + Nnodes`. For example, for an element type `NGON_n` composed of 25 nodes, one would set the `ElementType` to `NGON_n + 25`.

The `UserDefinedData_t` data structure allows arbitrary user-defined data to be stored in `Descriptor_t` and `DataArray_t` children without the restrictions or implicit meanings imposed on these node types at other node locations.

7.4 Elements Examples

This section contains two examples of elements definition in CGNS. In both cases, the unstructured zone contains 15 tetrahedral and 10 hexahedral elements.

Example 7-D: Unstructured Elements, Separate Element Types

In this first example, the elements are written in two separate sections, one for the tetrahedral elements and one for the hexahedral elements.

```
Zone_t UnstructuredZone =
  {{
  Elements_t TetraElements =
    {{
    IndexRange_t ElementRange = [1,15] ;

    int ElementSizeBoundary = 10 ;

    ElementType_t ElementType = TETRA_4 ;

    DataArray_t<int, 1, NPE[TETRA_4]×15> ElementConnectivity =
      {{
      Data(int, 1, NPE[TETRA_4]×15) = (node(i,j), i=1,NPE[TETRA_4], j=1,15) ;
      }} ;
    }} ;
  Elements_t HexaElements =
    {{
    IndexRange_t ElementRange = [16,25] ;

    int ElementSizeBoundary = 0 ;

    ElementType_t ElementType = HEXA_8 ;

    DataArray_t<int, 1, NPE[HEXA_8]×10> ElementConnectivity =
      {{
      Data(int, 1, NPE[HEXA_8]×10) = (node(i,j), i=1,NPE[HEXA_8], j=1,10) ;
      }} ;
    }} ;
  }} ;
```

Example 7-E: Unstructured Elements, Element Type `MIXED`

In this second example, the same unstructured zone described in Example 7-D is written in a single element section of type `MIXED` (i.e., an unstructured grid composed of mixed elements).

```
Zone_t UnstructuredZone =
{{
Elements_t MixedElementsSection =
  {{
  IndexRange_t ElementRange = [1,25] ;

  ElementType_t ElementType = MIXED ;

  DataArray_t<int, 1, ElementDataSize> ElementConnectivity =
    {{
    Data(int, 1, ElementDataSize) = (etype(j),(node(i,j),
         i=1,NPE[etype(j)]), j=1,25) ;
    }} ;
  }} ;
}} ;
```

7.5 Axisymmetry Structure Definition: `Axisymmetry_t`

The `Axisymmetry_t` data structure allows recording the axis of rotation and the angle of rotation around this axis for a two-dimensional dataset that represents an axisymmetric database.

```
Axisymmetry_t :=
  {
  List( Descriptor_t Descriptor1 ... DescriptorN ) ;                 (o)

  DataArray_t<real,1,2> AxisymmetryReferencePoint ;                  (r)
  DataArray_t<real,1,2> AxisymmetryAxisVector ;                      (r)
  DataArray_t<real,1,1> AxisymmetryAngle ;                           (o)
  DataArray_t<char,2,[32,2]> CoordinateNames ;                       (o)

  DataClass_t DataClass ;                                            (o)

  DimensionalUnits_t DimensionalUnits ;                              (o)

  List( UserDefinedData_t UserDefinedData1 ... UserDefinedDataN ) ;  (o)
  } ;
```

Notes

1. Default names for the `Descriptor_t` and `UserDefinedData_t` lists are as shown; users may choose other legitimate names. Legitimate names must be unique within a given instance of `Axisymmetry_t` and shall not include the names `AxisymmetryAngle`, `AxisymmetryAxisVector`, `AxisymmetryReferencePoint`, `CoordinateNames`, `DataClass`, or `DimensionalUnits`.
2. `AxisymmetryReferencePoint` and `AxisymmetryAxisVector` are the required fields within the `Axisymmetry_t` structure.

AxisymmetryReferencePoint specifies the origin used for defining the axis of rotation.

AxisymmetryAxisVector contains the direction cosines of the axis of rotation, through the AxisymmetryReferencePoint. For example, for a 2-D dataset defined in the (x, y) plane, if AxisymmetryReferencePoint contains $(0, 0)$ and AxisymmetryAxisVector contains $(1, 0)$, the x-axis is the axis of rotation.

AxisymmetryAngle allows specification of the circumferential extent about the axis of rotation. If this angle is undefined, it is assumed to be 360°.

CoordinateNames may be used to specify the first and second coordinates used in the definition of AxisymmetryReferencePoint and AxisymmetryAxisVector. If not found, it is assumed that the first coordinate is CoordinateX and the second is CoordinateY. The coordinates given under CoordinateNames, or implied by using the default, must correspond to those found under GridCoordinates_t.

DataClass defines the default class for numerical data contained in the DataArray_t entities. For dimensional data, DimensionalUnits may be used to describe the system of units employed. If present, these two entities take precedence over the corresponding entities at higher levels of the CGNS hierarchy, following the standard precedence rules.

The UserDefinedData_t data structure allows arbitrary user-defined data to be stored in Descriptor_t and DataArray_t children without the restrictions or implicit meanings imposed on these node types at other node locations.

7.6 Rotating Coordinates Structure Definition: RotatingCoordinates_t

The RotatingCoordinates_t data structure is used to record the rotation center and rotation rate vector of a rotating coordinate system.

```
RotatingCoordinates_t :=
  {
  List( Descriptor_t Descriptor1 ... DescriptorN ) ;                  (o)

  DataArray_t<real,1,PhysicalDimension> RotationCenter ;              (r)
  DataArray_t<real,1,PhysicalDimension> RotationRateVector ;          (r)

  DataClass_t DataClass ;                                             (o)

  DimensionalUnits_t DimensionalUnits ;                               (o)

  List( UserDefinedData_t UserDefinedData1 ... UserDefinedDataN ) ;   (o)
  } ;
```

Notes

1. Default names for the Descriptor_t and UserDefinedData_t lists are as shown; users may choose other legitimate names. Legitimate names must be unique within a given instance

of RotatingCoordinates_t and shall not include the names DataClass, DimensionalUnits, RotationCenter, or RotationRateVector.
2. RotationCenter and RotationRateVector are the required fields within the RotatingCoordinates_t structure.

RotationCenter specifies the coordinates of the center of rotation, and RotationRateVector specifies the components of the angular velocity of the grid about the center of rotation. Together, they define the angular velocity vector. The direction of the angular velocity vector specifies the axis of rotation, and its magnitude specifies the rate of rotation.

For example, for the common situation of rotation about the x-axis, RotationCenter would be specified as any point on the x-axis, like $(0, 0, 0)$. RotationRateVector would then be specified as (ω,0,0), where ω is the rotation rate. Using the right-hand rule, ω would be positive for clockwise rotation (looking in the $+x$ direction), and negative for counter-clockwise rotation.

Note that for a rotating coordinate system, the axis of rotation is defined in the inertial frame of reference, while the grid coordinates stored using the GridCoordinates_t data structure are relative to the rotating frame of reference.

DataClass defines the default class for data contained in the DataArray_t entities. For dimensional data, DimensionalUnits may be used to describe the system of units employed. If present, these two entities take precedence over the corresponding entities at higher levels of the CGNS hierarchy, following the standard precedence rules.

The UserDefinedData_t data structure allows arbitrary user-defined data to be stored in Descriptor_t and DataArray_t children without the restrictions or implicit meanings imposed on these node types at other node locations.

If rotating coordinates are used, it is useful to store variables relative to the rotating frame. Standardized data-name identifiers should be used for these variables, as defined for flow-solution quantities in Appendix A.

7.7 Flow Solution Structure Definition: FlowSolution_t

The flow solution within a given zone is described by the FlowSolution_t structure. This structure contains a list for the data arrays of the individual flow-solution variables, as well as identifying the grid location of the solution. It also provides a mechanism for identifying rind-point data included within the data arrays.

```
FlowSolution_t< int CellDimension, int IndexDimension,
                int VertexSize[IndexDimension],
                int CellSize[IndexDimension] > :=
  {
  List( Descriptor_t Descriptor1 ... DescriptorN ) ;                    (o)

  GridLocation_t GridLocation ;                                         (o/d)

  Rind_t<IndexDimension> Rind ;                                         (o/d)
  IndexRange_t<IndexDimension> PointRange ;                        (o)
  IndexArray_t<IndexDimension, ListLength[], int> PointList ;      (o)

  List( DataArray_t<DataType, IndexDimension, DataSize[]>
        DataArray1 ... DataArrayN ) ;                              (o)

  DataClass_t DataClass ;                                          (o)

  DimensionalUnits_t DimensionalUnits ;                            (o)

  List( UserDefinedData_t UserDefinedData1 ... UserDefinedDataN ) ;      (o)
  } ;
```

Notes

1. Default names for the `Descriptor_t`, `DataArray_t`, and `UserDefinedData_t` lists are as shown; users may choose other legitimate names. Legitimate names must be unique within a given instance of `FlowSolution_t` and shall not include the names `DataClass`, `DimensionalUnits`, `GridLocation`, `PointRange`, `PointList` or `Rind`.
2. There are no required fields for `FlowSolution_t`. `GridLocation` has a default of `Vertex` if absent. `Rind` also has a default if absent; the default is equivalent to having an instance of `Rind` whose `RindPlanes` array contains all zeros (see Section 4.8).
3. Both of the fields `PointRange` and `PointList` are optional. Only one of these two fields may be specified.
4. The structure parameter `DataType` must be consistent with the data stored in the `DataArray_t` structure entities (see Section 5.1); `DataType` is `real` for all flow-solution identifiers defined in Appendix A.
5. For unstructured zones `GridLocation` options are limited to `Vertex` or `CellCenter`, unless one of `PointRange` or `PointList` is present.
6. Indexing of data within the `DataArray_t` structures, must be consistent with the associated numbering of vertices or elements.

`FlowSolution_t` requires four structure parameters: `CellDimension` identifies the dimensionality of cells or elements, `IndexDimension` identifies the dimensionality of the grid-size arrays, and `VertexSize` and `CellSize` are the number of core vertices and cells, respectively, in each index direction. For unstructured zones, `IndexDimension` is always 1.

The flow solution data is stored in the list of `DataArray_t` entities; each `DataArray_t` structure entity may contain a single component of the solution vector. Standardized data-name identifiers for the flow-solution quantities are described in Appendix A. The field `GridLocation` specifies the location of the solution data with respect to the grid; if absent, the data is assumed to coincide with grid vertices (i.e., `GridLocation = Vertex`). All data within a given instance of `FlowSolution_t` must reside at the same grid location.

For structured grids, the value of `GridLocation` alone specifies the location and indexing of the flow solution data. Vertices are explicity indexed. Cell centers and face centers are indexed using the minimum of the connecting vertex indices, as described in the section Structured Grid Notation and Indexing Conventions (Section 3.2).

For unstructured grids, the value of `GridLocation` alone specifies location and indexing of flow solution data only for vertex and cell-centered data. The reason for this is that element-based grid connectivity provided in the `Elements_t` data structures explicitly indexes only vertices and cells. For data stored at alternate grid locations (e.g. edges), additional connectivity information is needed. This is provided by the optional fields `PointRange` and `PointList`; these refer to vertices, edges, faces or cell centers, depending on the values of `CellDimension` and `GridLocation`. The following table shows these relations.

`CellDimension`	`GridLocation`			
	`Vertex`	`EdgeCenter`	`*FaceCenter`	`CellCenter`
1	vertices	–	–	cells (line elements)
2	vertices	edges	–	cells (area elements)
3	vertices	edges	faces	cells (volume elements)

In the table, `*FaceCenter` stands for the possible types: `FaceCenter`, `IFaceCenter`, `JFaceCenter` or `KFaceCenter`.

Although intended for edge or face-based solution data for unstructured grids, the fields `PointRange/List` may also be used to (redundantly) index vertex and cell-centered data. In all cases, indexing of flow solution data corresponds to the element numbering as defined in the `Elements_t` data structures.

`Rind` is an optional field that indicates the number of rind planes (for structured grids) or rind points or elements (for unstructured grids) included in the data. Its purpose and function are identical to those described in Section 7.1. Note, however, that the `Rind` in this structure is independent of the `Rind` contained in `GridCoordinates_t`. They are not required to contain the same number of rind planes or elements. Also, the location of any flow-solution rind points is assumed to be consistent with the location of the core flow solution points (e.g., if `GridLocation = CellCenter`, rind points are assumed to be located at fictitious cell centers).

`DataClass` defines the default class for data contained in the `DataArray_t` entities. For dimensional flow solution data, `DimensionalUnits` may be used to describe the system of units employed. If present these two entities take precedence over the corresponding entities at higher levels of the CGNS hierarchy. The rules for determining precedence of entities of this type are discussed in Section 6.4.

The `UserDefinedData_t` data structure allows arbitrary user-defined data to be stored in `Descriptor_t` and `DataArray_t` children without the restrictions or implicit meanings imposed on these node types at other node locations.

FUNCTION `ListLength[]`:

return value: `int`
dependencies: `PointRange`, `PointList`

FlowSolution_t requires the structure function ListLength, which is used to specify the number of entities (e.g. vertices) corresponding to a given PointRange or PointList. If PointRange is specified, then ListLength is obtained from the number of points (inclusive) between the beginning and ending indices of PointRange. If PointList is specified, then ListLength is the number of indices in the list of points. In this situation, ListLength becomes a user input along with the indices of the list PointList. By "user" we mean the application code that is generating the CGNS database.

FUNCTION DataSize[]:

return value: one-dimensional int array of length IndexDimension
dependencies: IndexDimension, VertexSize, CellSize, GridLocation, Rind, ListLength[]

The function DataSize[] is the size of flow solution data arrays. If Rind is absent then DataSize represents only the core points; it will be the same as VertexSize or CellSize depending on GridLocation. The definition of the function DataSize[] is as follows:

```
if (PointRange/PointList is present) then
  {
  DataSize[] = ListLength[] ;
  }

else if (Rind is absent) then
  {
  if (GridLocation = Vertex) or (GridLocation is absent)
    {
    DataSize[] = VertexSize ;
    }
  else if (GridLocation = CellCenter) then
    {
    DataSize[] = CellSize ;
    }
  }
else if (Rind is present) then
  {
  if (GridLocation = Vertex) or (GridLocation is absent) then
    {
    DataSize[] = VertexSize + [a + b,...] ;
    }
  else if (GridLocation = CellCenter)
    {
    DataSize[] = CellSize + [a + b,...] ;
    }
  }
```

where `RindPlanes = [a,b,...]` (see Section 4.8 for the definition of `RindPlanes`).

7.8 Flow Solution Example

This section contains an example of the flow solution entity, including the designation of grid location and rind planes and data-normalization mechanisms.

Example 7-F: Flow Solution

Conservation-equation variables (ρ, ρu, ρv and ρe_0) for a 2-D grid of size 11×5. The flowfield is cell-centered with two planes of rind data. The density, momentum and stagnation energy (ρe_0) data is nondimensionalized with respect to a freestream reference state whose quantities are dimensional. The freestream density and pressure are used for normalization; these values are 1.226 kg/m^3 and 1.0132×10^5 N/m^2 (standard atmosphere conditions). The data-name identifier conventions for the conservation-equation variables are `Density`, `MomentumX`, `MomentumY` and `EnergyStagnationDensity`.

```
! CellDimension = 2
! IndexDimension = 2
! VertexSize = [11,5]
! CellSize = [10,4]
FlowSolution_t<2, 2, [11,5], [10,4]> FlowExample =
  {{
  GridLocation_t GridLocation = CellCenter ;

  Rind_t<2> Rind =
    {{
    int[4] RindPlanes = [2,2,2,2] ;
    }} ;

  DataClass_t DataClass = NormalizedByDimensional ;

  DimensionalUnits_t DimensionalUnits =
    {{
    MassUnits        = Kilogram ;
    LengthUnits      = Meter ;
    TimeUnits        = Second ;
    TemperatureUnits = Null ;
    AngleUnits       = Null ;
    }} ;

  ! DataType = real
  ! Dimension = 2
  ! DataSize = CellSize + [4,4] = [14,8]
  DataArray_t<real, 2, [14,8]> Density =
    {{
```

```
    Data(real, 2, [14,8]) = ((rho(i,j), i=-1,12), j=-1,6) ;

    DataConversion_t DataConversion =
      {{
      ConversionScale  = 1.226 ;
      ConversionOffset = 0 ;
      }} ;

    DimensionalExponents_t DimensionalExponents =
      {{
      MassExponent        = +1 ;
      LengthExponent      = -3 ;
      TimeExponent        =  0 ;
      TemperatureExponent =  0 ;
      AngleExponent       =  0 ;
      }} ;
    }} ;

  DataArray_t<real, 2, [14,8]> MomentumX =
    {{
    Data(real, 2, [14,8]) = ((rho_u(i,j), i=-1,12), j=-1,6) ;

    DataConversion_t DataConversion =
      {{
      ConversionScale  = 352.446 ;
      ConversionOffset = 0 ;
      }} ;
    }} ;

  DataArray_t<real, 2, [14,8]> MomentumY =
    {{
    Data(real, 2, [14,8]) = ((rho_v(i,j), i=-1,12), j=-1,6) ;

    DataConversion_t DataConversion =
      {{
      ConversionScale  = 352.446 ;
      ConversionOffset = 0 ;
      }} ;
    }} ;

  DataArray_t<real, 2, [14,8]> EnergyStagnationDensity =
    {{
    Data(real, 2, [14,8]) = ((rho_e0(i,j), i=-1,12), j=-1,6) ;

    DataConversion_t DataConversion =
      {{
      ConversionScale  = 1.0132e+05 ;
      ConversionOffset = 0 ;
      }} ;
    }} ;
  }} ;
```

The value of `GridLocation` indicates the data is at cell centers, and the value of `RindPlanes` specifies two rind planes on each face of the zone. The resulting value of the structure function `DataSize` is the number of cells plus four in each coordinate direction; this value is passed to each of the `DataArray_t` entities.

Since the data are all nondimensional and normalized by dimensional reference quantities, this information is stated in `DataClass` and `DimensionalUnits` at the `FlowSolution_t` level rather than attaching the appropriate `DataClass` and `DimensionalUnits` to each `DataArray_t` entity. It could possibly be at even higher levels in the heirarchy. The contents of `DataConversion` are in each case the denominator of the normalization; this is ρ_∞ for density, $\sqrt{p_\infty \rho_\infty}$ for momentum, and p_∞ for stagnation energy. The dimensional exponents are specified for density. For all the other data, the dimensional exponents are to be inferred from the data-name identifiers.

Note that no information is provided to identify the actual reference state or indicate that it is freestream. This information is not needed for data manipulations involving renormalization or changing the units of the converted raw data.

Note that there are no boundary-condition structures defined for abutting or overset interfaces, unless they involve cases of symmetry or degeneracy. In other words, it is a CGNS design intent that a given zone boundary segment or location should at most be defined (or covered) by either a boundary condition or a multizone interface connectivity, but not by both. There is also no separate boundary-condition structure for periodic boundary conditions (i.e., when a zone interfaces with itself). Both of these situations are addressed by the interface connectivity data structures described in Section 8.

In the sections to follow, the definitions of boundary-condition structures are first presented in Section 9.1 through Section 9.6. Boundary-condition types are then discussed in detail in Section 9.7, including a description of the boundary-condition equations to be enforced for each type; this section also describes the distinction between boundary-condition types that impose a set of equations regardless of local flow conditions and those that impose different sets of boundary-condition equations depending on the local flow solution. The rules for matching boundary-condition types and the appropriate sets of boundary-condition equations are next discussed in Section 9.8. Details of specifying data to be imposed in boundary-condition equations are provided in Section 9.9. Finally, Section 9.10 presents several examples of boundary conditions.

9.1 Boundary Condition Structures Overview

Prior to presenting the detailed boundary condition structures, we give a brief overview of the hierarchy used to describe boundary conditions.

Boundary conditions are classified as either fixed or flow-dependent. Fixed boundary conditions enforce a given set of boundary-condition equations regardless of flow conditions; whereas, flow-dependent boundary conditions enforce different sets of boundary-condition equations depending on local flow conditions. We incorporate both fixed and flow-dependent boundary conditions into a uniform framework. This allows all boundary conditions to be described in a similar manner. We consider this functionally superior to separately treating fixed and flow-dependent boundary conditions, even though the latter allows a simpler description mechanism for fixed boundary conditions. The current organization also makes sense considering the fact that flow-dependent boundary conditions are composed of multiple sets of fixed boundary conditions.

Figure 7 depicts the hierarchy used for prescribing a single boundary condition. Each boundary condition includes a type that describes the general equations to enforce, a patch specification, and a collection of data sets. The minimum required information for any boundary condition is the patch specification and the boundary-condition type (indicated by "BC type (compound)" in the figure). This minimum information is similar to that used in many existing flow solvers.

Generality in prescribing equations to enforce and their associated boundary-condition data is provided in the optional data sets. Each data set contains all boundary condition data required for a given fixed or simple boundary condition. Each data set is also tagged with a boundary-condition type. For fixed boundary conditions, the hierarchical tree contains a single data set, and the two boundary-condition types shown in Figure 7 are identical. Flow-dependent or compound boundary conditions contain multiple data sets, each to be applied separately depending on local flow conditions. The compound boundary-condition type describes the general flow-dependent boundary conditions, and each data set contains associated simple boundary-condition types. For

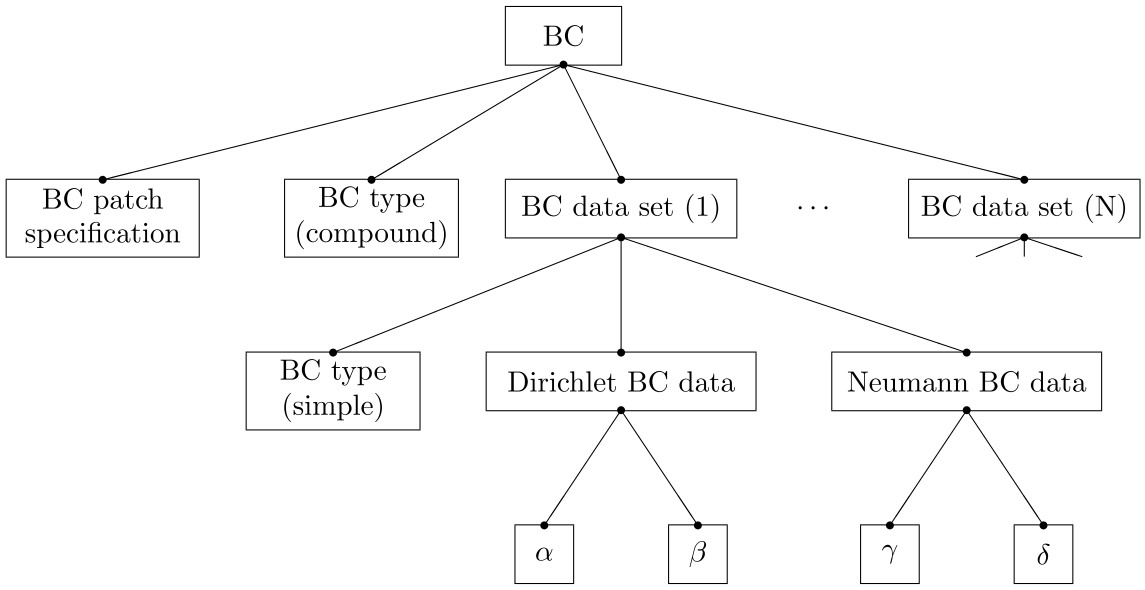

Figure 7: Hierarchy for Boundary Condition Structures

example, a farfield boundary condition would contain four data sets, where each applies to the different combinations of subsonic and supersonic inflow and outflow. Boundary-condition types are described in Section 9.7 and Section 9.8.

Within a single data set, boundary condition data is grouped by equation type into Dirichlet and Neumann data. The lower leaves of Figure 7 show data for generic flow-solution quantities α and β to be applied in Dirichlet conditions, and data for γ and δ to be applied in Neumann boundary conditions. `DataArray_t` entities are employed to store these data and to identify the specific flow variables they are associated with.

In situations where the data sets (or any information contained therein) are absent from a given boundary-condition hierarchy, flow solvers are free to impose any appropriate boundary conditions. Although not pictured in Figure 7, it is also possible to specify the reference state from which the flow solver should extract the boundary-condition data.

9.2 Zonal Boundary Condition Structure Definition: `ZoneBC_t`

All boundary-condition information pertaining to a given zone is contained in the `ZoneBC_t` structure.

```
ZoneBC_t< int CellDimension, int IndexDimension, int PhysicalDimension > :=
  {
  List( Descriptor_t Descriptor1 ... DescriptorN ) ;                    (o)

  List( BC_t<CellDimension, IndexDimension, int PhysicalDimension>
        BC1 ... BCN ) ;                                                 (o)
  ReferenceState_t ReferenceState ;                                    (o)

  DataClass_t DataClass ;                                              (o)

  DimensionalUnits_t DimensionalUnits ;                                (o)

  List( UserDefinedData_t UserDefinedData1 ... UserDefinedDataN ) ;    (o)
  } ;
```

Notes

1. Default names for the `Descriptor_t`, `BC_t`, and `UserDefinedData_t` lists are as shown; users may choose other legitimate names. Legitimate names must be unique within a given instance of `ZoneBC_t` and shall not include the names `DataClass`, `DimensionalUnits`, or `ReferenceState`.
2. All lists within a `ZoneBC_t` structure entity may be empty.

`ZoneBC_t` requires three structure parameters, `CellDimension`, `IndexDimension` and `PhysicalDimension`, which are passed onto all `BC_t` substructures.

Boundary-condition information for a single patch is contained in the `BC_t` structure. All boundary-condition information pertaining to a given zone is contained in the list of `BC_t` structure entities. If a zone contains N boundary-condition patches, then N (and only N) separate instances of `BC_t` must be provided in the `ZoneBC_t` entity for the zone. That is, each boundary-condition patch must be represented by a single `BC_t` entity.

Reference data applicable to all boundary conditions of a zone is contained in the `ReferenceState` structure. `DataClass` defines the zonal default for the class of data contained in the boundary conditions of a zone. If the boundary conditions contain dimensional data, `DimensionalUnits` may be used to describe the system of dimensional units employed. If present, these three entities take precedence of all corresponding entities at higher levels of the hierarchy. These precedence rules are further discussed in Section 6.4.

Reference-state data is useful for situations where boundary-condition data is not provided, and flow solvers are free to enforce any appropriate boundary condition equations. The presense of `ReferenceState` at this level or below specifies the appropriate flow conditions from which the flow solver should extract its boundary-condition data. For example, when computing an external flowfield around an airplane, an engine nozzle exit is often simulated by imposing a stagnation pressure boundary condition (or some other stagnation quantity) different from freestream. The nozzle-exit stagnation quantities could be specified in an instance of `ReferenceState` at this level or below in lieu of providing explicit Dirichlet or Neumann data (see Section 9.9).

The `UserDefinedData_t` data structure allows arbitrary user-defined data to be stored in `Descriptor_t` and `DataArray_t` children without the restrictions or implicit meanings imposed on these node types at other node locations.

9.3 Boundary Condition Structure Definition: BC_t

BC_t contains boundary-condition information for a single BC surface patch of a zone. A BC patch is the subrange of the face of a zone where a given boundary condition is applied.

The structure contains a boundary-condition type, as well as one or more sets of boundary-condition data that are used to define the boundary-condition equations to be enforced on the BC patch. For most boundary conditions, a single data set is all that is needed. The structure also contains information describing the normal vector to the BC surface patch.

```
BC_t< int CellDimension, int IndexDimension, int PhysicalDimension > :=
  {
  List( Descriptor_t Descriptor1 ... DescriptorN ) ;                    (o)

  BCType_t BCType ;                                                     (r)

  GridLocation_t GridLocation ;                                         (o/d)

  IndexRange_t<IndexDimension> PointRange ;                             (r:o)
  IndexArray_t<IndexDimension, ListLength[], int> PointList ;           (o:r)

  int[IndexDimension] InwardNormalIndex ;                               (o)

  IndexArray_t<PhysicalDimension, ListLength[], real> InwardNormalList ;  (o)

  List( BCDataSet_t<CellDimension, IndexDimension, ListLength[], GridLocation>
        BCDataSet1 ... BCDataSetN ) ;                                   (o)

  BCProperty_t BCProperty ;                                             (o)

  FamilyName_t FamilyName ;                                             (o)

  ReferenceState_t ReferenceState ;                                     (o)

  DataClass_t DataClass ;                                               (o)

  DimensionalUnits_t DimensionalUnits ;                                 (o)

  List( UserDefinedData_t UserDefinedData1 ... UserDefinedDataN ) ;     (o)

  int Ordinal ;                                                         (o)
  } ;
```

Notes

1. Default names for the Descriptor_t, BCDataSet_t, and UserDefinedData_t lists are as shown; users may choose other legitimate names. Legitimate names must be unique within

a given instance of BC_t and shall not include the names BCProperty, BCType, DataClass, DimensionalUnits, FamilyName, GridLocation, InwardNormalIndex, InwardNormalList, Ordinal, PointList, PointRange or ReferenceState.

2. GridLocation is optional; if absent its default value is Vertex. For 2–D grids (CellDimension = 2), GridLocation may take the additional value of EdgeCenter. For 3–D grids (CellDimension = 3), GridLocation may take the additional values of EdgeCenter, FaceCenter, IFaceCenter, JFaceCenter or KFaceCenter.
3. One of PointRange or PointList must be specified but not both. They must define a face subrange of the zone.
4. InwardNormalIndex is only an option for structured grids. For unstructured grid boundaries, it should not be used. InwardNormalIndex may have only one nonzero element, whose sign indicates the computational-coordinate direction of the BC patch normal; this normal points into the interior of the zone.
5. InwardNormalList contains a list of vectors normal to the BC patch pointing into the interior of the zone. It is a function of PhysicalDimension and ListLength[]. The vectors are located at the vertices of the BC patch when GridLocation is set to Vertex. Otherwise, they are located at edge/face midpoints. The vectors are not required to have unit magnitude.
6. If PointRange and InwardNormalList are specified, an ordering convention is needed for indices on the BC patch. An ordering convention is also needed if a range is specified and local data is present in the BCDataSet_t substructures. FORTRAN multidimensional array ordering is used.

BCType specifies the boundary-condition type, which gives general information on the boundary-condition equations to be enforced. BCType_t is defined in Section 9.7 along with the meanings of all the BCType values.

The BC patch may be specified by PointRange if it constitutes a logically rectangular region. In all other cases, PointList should be used to list the vertices or cell edges/faces making up the BC patch. When GridLocation is set to Vertex, then PointList or PointRange refer to vertex indices, for both structured and unstructured grids. When GridLocation is set to EdgeCenter, then PointRange/List refer to edge elements. For 3–D grids, when GridLocation is set to FaceCenter, IFaceCenter, etc., then PointRange/List refer to face elements. The interpretation of PointRange/List is summarized in the table below:

CellDimension	GridLocation		
	Vertex	EdgeCenter	*FaceCenter
1	vertices	–	–
2	vertices	edges	–
3	vertices	edges	faces

In the table, *FaceCenter stands for the possible types: FaceCenter, IFaceCenter, JFaceCenter or KFaceCenter.

For structured grids, face centers are indexed using the minimum of the connecting vertex indices, as described in Section 3.2. For unstructured grids, edge and face elements are indexed using their element numbering as defined in the Elements_t data structures.

The BC patch defined by `PointRange/List` is a surface region over which the particular set of boundary conditions is applied. However, in the current standard there is no mechanism to specify whether boundary conditions are enforced in the weak or strong form. If boundary conditions are imposed using collocation (i.e., strong form), there is also no requirement that they be imposed at the same locations used to define the BC patch (via `PointRange/List`). In the case when BC patches are defined in terms of vertices (or edges in 3–D), then the bounding vertices will be located on multiple BC patches. If boundary conditions are imposed using collocation at vertices, then for this case there is no mechanism to determine which BC patch takes precedence for any of these bounding vertices.

Some boundary conditions require a normal direction to be specified in order to be properly imposed. For structured zones a computational-coordinate normal can be derived from the BC patch specification by examining redundant index components. Alternatively, for structured zones this information can be provided directly by `InwardNormalIndex`. From Note 4, this vector points into the zone and can have only one non-zero element. For exterior faces of a zone in 3-D, `InwardNormalIndex` should take the following values:

Face	`InwardNormalIndex`	**Face**	`InwardNormalIndex`
i-min	$[+1, 0, 0]$	i-max	$[-1, 0, 0]$
j-min	$[0, +1, 0]$	j-max	$[0, -1, 0]$
k-min	$[0, 0, +1]$	k-max	$[0, 0, -1]$

The physical-space normal vectors of the BC patch may be described by `InwardNormalList`; these are located at vertices or cell faces, consistent with the BC patch specification. `InwardNormalList` is listed as an optional field because it is not always needed to enforce boundary conditions, and the physical-space normals of a BC patch can usually be constructed from the grid. However, there are some situations, such as grid-coordinate singularity lines, where `InwardNormalList` becomes a required field, because it cannot be generated from other information.

The `BC_t` structure provides for a list of boundary-condition data sets, described in the next section. In general, the proper `BCDataSet_t` instance to impose on the BC patch is determined by the `BCType` association table (Table 4 on p. 117). The mechanics of determining the proper data set to impose is described in Section 9.8.

For a few boundary conditions, such as a symmetry plane or polar singularity, the value of `BCType` completely describes the equations to impose, and no instances of `BCDataSet_t` are needed. For "simple" boundary conditions, where a single set of Dirichlet and/or Neumann data is applied, a single `BCDataSet_t` will likely appear (although this is not a requirement). For "compound" boundary conditions, where the equations to impose are dependent on local flow conditions, several instances of `BCDataSet_t` will likely appear; the procedure for choosing the proper data set is more complex as described in Section 9.8.

A `BCProperty_t` data structure, described in Section 9.6, may be used to record special properties associated with particular boundary condition patches, such as wall functions or bleed regions.

`FamilyName` identifies the family to which the boundary belongs. Family names link the mesh boundaries to the CAD surfaces. (See Section 12.6.) Boundary conditions may also be defined directly on families. In this case, the BCType must be FamilySpecified. If, under a BC_t structure, both FamilyName_t and BCType_t are present, and the BCType is *not* FamilySpecified, then the BCType which *is* specified takes precedence over any BCType which might be stored in a FamilyBC_t structure under the specified Family_t.

Reference data applicable to the boundary conditions of a BC patch is contained in the ReferenceState structure. DataClass defines the default for the class of data contained in the boundary conditions. If the boundary conditions contain dimensional data, DimensionalUnits may be used to describe the system of dimensional units employed. If present, these three entities take precedence of all corresponding entities at higher levels of the hierarchy. These precedence rules are further discussed in Section 6.4.

The UserDefinedData_t data structure allows arbitrary user-defined data to be stored in Descriptor_t and DataArray_t children without the restrictions or implicit meanings imposed on these node types at other node locations.

Ordinal is user-defined and has no restrictions on the values that it can contain. It is included for backward compatibility to assist implementation of the CGNS system into applications whose I/O depends heavily on the numbering of BC patches. Since there are no restrictions on the values contained in Ordinal (or that Ordinal is even provided), there is no guarantee that the BC patches for a given zone in an existing CGNS database will have sequential values from 1 to N without holes or repetitions. Use of Ordinal is discouraged and is on a user-beware basis.

FUNCTION ListLength[]:

return value: int
dependencies: PointRange, PointList

BC_t requires the structure function ListLength, which is used to specify the number of vertices or edge/face elements making up the BC patch. If PointRange is specified, then ListLength is obtained from the number of points (inclusive) between the beginning and ending indices of PointRange. If PointList is specified, then ListLength is the number of indices in the list of points. In this situation, ListLength becomes a user input along with the indices of the list PointList. By “user” we mean the application code that is generating the CGNS database.

ListLength is also the number of elements in the list InwardNormalList. Note that syntactically PointList and InwardNormalList must have the same number of elements.

If neither PointRange or PointList is specified in a particular BCDataSet_t substructure, ListLength must be passed into it to determine the length of BC data arrays.

9.4 Boundary Condition Data Set Structure Definition: BCDataSet_t

BCDataSet_t contains Dirichlet and Neumann data for a single set of boundary-condition equations. Its intended use is for simple boundary-condition types, where the equations imposed do not depend on local flow conditions.

```
  BCDataSet_t< int CellDimension, int IndexDimension, int ListLengthParameter,
GridLocation_t GridLocationParameter > :=
{
List( Descriptor_t Descriptor1 ... DescriptorN ) ;                    (o)

BCTypeSimple_t BCTypeSimple ;                                         (r)

BCData_t<ListLengthBCData[]> DirichletData ;                          (o)
BCData_t<ListLengthBCData[]> NeumannData ;                            (o)

GridLocation_t GridLocation ;                                         (o/d)

IndexRange_t<IndexDimension> PointRange ;                             (o)
IndexArray_t<IndexDimension, ListLength[], int> PointList ;           (o)

ReferenceState_t ReferenceState ;                                     (o)

DataClass_t DataClass ;                                               (o)

DimensionalUnits_t DimensionalUnits ;                                 (o)

List( UserDefinedData_t UserDefinedData1 ... UserDefinedDataN ) ;     (o)
} ;
```

Notes

1. Default names for the `Descriptor_t` and `UserDefinedData_t` lists are as shown; users may choose other legitimate names. Legitimate names must be unique within a given instance of `BCDataSet_t` and shall not include the names `BCTypeSimple`, `DataClass`, `DimensionalUnits`, `DirichletData`, `GridLocation`, `NeumannData`, `PointList`, `PointRange`, or `ReferenceState`.
2. `BCTypeSimple` is the only required field. All other fields are optional and the `Descriptor_t` list may be empty.
3. `GridLocation` is optional; if absent its default value is `GridLocationParameter`. For 2–D grids (`CellDimension` = 2), `GridLocation` may take the values of `Vertex` or `EdgeCenter`. For 3–D grids (`CellDimension` = 3), `GridLocation` may take the values of `Vertex`, `EdgeCenter`, `FaceCenter`, `IFaceCenter`, `JFaceCenter` or `KFaceCenter`.
4. `PointRange` and `PointList` are both optional; only one of them may be specified. They must define a face subrange of the zone.

`BCDataSet_t` requires the structure parameters `CellDimension`, `IndexDimension`, `ListLengthParameter` and `GridLocationParameter`. These are all used to control the grid location and length of data arrays in the `Dirichlet` and `Neumann` substructures. They are inputs for the structure functions `ListLength[]` and `ListLengthBCData[]` defined below.

`BCTypeSimple` specifies the boundary-condition type, which gives general information on the boundary-condition equations to be enforced. `BCTypeSimple_t` is defined in Section 9.7 along with the meanings of all the BCTypeSimple values. BCTypeSimple is also used for matching boundary condition data sets as discussed in Section 9.8.

Boundary-condition data is separated by equation type into Dirichlet and Neumann conditions. Dirichlet boundary conditions impose the value of the given variables, whereas Neumann boundary conditions impose the normal derivative of the given variables. The mechanics of specifying Dirichlet and Neumann data for boundary conditions is covered in Section 9.9.

The substructures DirichletData and NeumannData contain boundary-condition data which may be constant over the BC patch or defined locally at each vertex or edge/face of the patch. Locally defined data can be specified in one of two ways. If GridLocation, PointRange and PointList are all absent, then the data is defined consistent with the BC patch specification of the parent BC_t structure. In this case, the location of the locally defined data is given by GridLocationParameter and the length of the data arrays are given by ListLengthParameter. If GridLocation and one of PointRange or PointList is present, then the same rules provided in Section Section 9.3 apply. In this case, the length of the data arrays is given by ListLength[].

Reference quantities applicable to the set of boundary-condition data are contained in the ReferenceState structure. DataClass defines the default for the class of data contained in the boundary-condition data. If the boundary conditions contain dimensional data, DimensionalUnits may be used to describe the system of dimensional units employed. If present, these three entities take precedence of all corresponding entities at higher levels of the hierarchy. These precedence rules are further discussed in Section 6.4.

The UserDefinedData_t data structure allows arbitrary user-defined data to be stored in Descriptor_t and DataArray_t children without the restrictions or implicit meanings imposed on these node types at other node locations.

FUNCTION ListLength[]:

return value: int
dependencies: PointRange, PointList

BCDataSet_t requires the structure function ListLength, which is used to specify the length of locally defined Dirichlet and Neumann data arrays when the grid location of these quatities differs from that of the BC patch definition. The definition of ListLength is identical to that provided in BC_t (Section 9.3).

FUNCTION ListLengthBCData[]:

return value: int
dependencies: ListLengthParameter, PointRange, PointList

BCDataSet_t also requires the structure function ListLengthBCData. If PointRange or PointList is present, then ListLengthBCData takes the value of ListLength. If both are absent, then it takes the value ListLengthParameter.

This example raises the question of whether unused structure parameters are required in structure entities. The answer is no. We included them here for completeness. The purpose of structure parameters is to mimic the need to define elements of a entity based on information contained elsewhere (at a higher level) in the CGNS database. When this need is not present in a given instance of a structure entity, the structure parameters are superfluous. In some of the following examples, structure parameters that are superfluous or otherwise not needed are denoted by "?".

Example 9-C: Subsonic Inflow

Subsonic inflow for a 2-D structured zone: The BC patch is on the i-min face and includes $j \in [2, 7]$. As prescribed by the boundary-condition type, three quantities must be specified. Uniform entropy and stagnation enthalpy are specified with values of 0.94 and 2.85, respectively. A velocity profile is specified at face midpoints, given by the array `v_inflow(j)`. No dimensional or nondimensional information is provided.

```
! CellDimension = 2, IndexDimension = 2
BC_t<2,2,?> BC3 =
  {{
  BCType_t BCType = BCInflowSubsonic ;

  GridLocation_t GridLocation = FaceCenter ;

  IndexRange_t<2> PointRange =
    {{
    int[2] Begin = [1,2] ;
    int[2] End   = [1,6] ;
    }} ;

  ! ListLength = 5
  BCDataSet_t<5> BCDataSet1 =
    {{
    BCTypeSimple_t BCTypeSimple = BCInflowSubsonic ;

    ! Data array length = ListLength = 5
    BCData_t<5> DirichletData =
      {{
      DataArray_t<real, 1, 1> EntropyApprox =
        {{
        Data(real, 1, 1) = 0.94 ;
        }} ;

      DataArray_t<real, 1, 1> EnthalpyStagnation =
        {{
        Data(real, 1, 1) = 2.85 ;
        }} ;
      DataArray_t<real, 1, 5> VelocityY =
        {{
        Data(real, 1, 5) = (v_inflow(j), j=3,7) ;
        }} ;
      }} ;
    }} ;
  }} ;
```

This is another example of a simple boundary-condition type. The primary additional complexity included in this example is multiple Dirichlet conditions with one containing local data. DirichletData contains three DataArray_t entities named EntropyApprox, EnthalpyStagnation and VelocityY. This specifies three Dirichlet boundary conditions to be enforced, and the names identify the solution quantities to set. Since both EntropyApprox and EnthalpyStagnation have an array-length structure parameter of one, they identify global data, and the values are provided. VelocityY is an array of data values and contains the values in v_inflow(). The length of the array is given by ListLength, which represents the number of cell faces because BC3 is specified using the value of FaceCenter for GridLocation. Note that the beginning and ending indices on the array v_inflow() are unimportant (they are user inputs); there just needs to be five values provided.

Example 9-D: Outflow

Outflow boundary condition with unspecified normal Mach number for an *i*-max face of a 3-D structured zone: for subsonic outflow, a uniform pressure is specified; for supersonic outflow, no boundary-condition equations are specified.

```
  !  CellDimension = 3, IndexDimension = 3
  BC_t<3,3,3> BC4 =
    {{
    BCType_t BCType = BCOutflow ;

    IndexRange_t<3> PointRange = {{ }} ;

    BCDataSet_t<?> BCDataSetSubsonic =
      {{
      BCTypeSimple_t BCTypeSimple = BCOutflowSubsonic ;

      BCData_t<?> DirichletData =
        {{
        DataArray_t<real, 1, 1> Pressure = {{ }} ;
        }} ;
      }} ;

    BCDataSet_t<?> BCDataSetSupersonic =
      {{
      BCTypeSimple_t BCTypeSimple = BCOutflowSupersonic ;
      }} ;
    }} ;
```

This is an example of a complex boundary-condition type; the equation set to be enforced depends on the local flow conditions, namely the Mach number normal to the boundary. Two data sets are provided, `BCDataSetSubsonic` and `BCDataSetSupersonic`; recall the names are unimportant and are user defined. The first data set has a boundary-condition type of `BCOutflowSubsonic` and prescribes a global Dirichlet condition on static pressure. Any additional boundary conditions needed may be applied by a flow solver. The second data set has a boundary-condition type of `BCOutflowSupersonic` with no additional boundary-condition equation specification. Typically, all solution quantities are extrapolated from the interior for supersonic outflow. From the boundary-condition type association table (Table 4), `BCOutflow` requires two data sets with boundary-condition types `BCOutflowSubsonic` and `BCOutflowSupersonic`. The accompanying usage rule states that the data set for `BCOutflowSubsonic` should be used for a subsonic normal Mach number; otherwise, the data set for `BCOutflowSupersonic` should be enforced.

Any additional data sets with boundary-condition types other than `BCOutflowSubsonic` or `BCOutflowSupersonic` could be provided (the definition of `BC_t` allows an arbitrary list of `BCDataSet_t` entities); however, they should be ignored by any code processing the boundary-condition information. Another caveat is that providing two data sets with the same simple boundary-condition type would cause indeterminate results — which one is the correct data set to apply?

The actual global data value for static pressure is not provided; an abbreviated form of the `Pressure` entity is shown. This example also uses the “`?`” notation for unused data-array-length structure parameters.

Example 9-E: Farfield

Farfield boundary condition with arbitrary flow conditions for a *j*-max face of a 2-D structured zone: If subsonic inflow, specify entropy, vorticity and incoming acoustic characteristics; if supersonic inflow specify entire flow state; if subsonic outflow, specify incoming acoustic characteristic; and if supersonic outflow, extrapolate all flow quantities. None of the extrapolated quantities for the different boundary condition possibilities need be stated.

```
!  CellDimension = 2, IndexDimension = 2
BC_t<2,2,2> BC5 =
  {{
  BCType_t BCType = BCFarfield ;

  IndexRange_t<2> PointRange = {{ }} ;

  int[2] InwardNormalIndex = [0,-1] ;

  BCDataSet_t<?> BCDataSetInflowSupersonic =
    {{
    BCTypeSimple_t BCTypeSimple = BCInflowSupersonic ;
    }} ;
  BCDataSet_t<?> BCDataSetInflowSubsonic =
    {{
    BCTypeSimple_t BCTypeSimple = BCInflowSubsonic ;

    BCData<?> DirichletData =
      {{
      DataArray_t<real, 1, 1> CharacteristicEntropy      = {{ }} ;
      DataArray_t<real, 1, 1> CharacteristicVorticity1   = {{ }} ;
      DataArray_t<real, 1, 1> CharacteristicAcousticPlus = {{ }} ;
      }} ;
    }} ;

  BCDataSet_t<?> BCDataSetOutflowSupersonic =
    {{
    BCTypeSimple_t BCTypeSimple = BCOutflowSupersonic ;
    }} ;

  BCDataSet_t<?> BCDataSetOutflowSubsonic =
    {{
    BCTypeSimple_t BCTypeSimple = BCOutflowSubsonic ;

    BCData<?> DirichletData =
      {{
      DataArray_t<real, 1, 1> CharacteristicAcousticMinus = {{ }} ;
      }} ;
    }} ;
  }} ;
```

The farfield boundary-condition type is the most complex of the compound boundary-condition types. `BCFarfield` requires four data sets; these data sets must contain the simple boundary-condition types `BCInflowSupersonic`, `BCInflowSubsonic`, `BCOutflowSupersonic` and `BCOutflow-Subsonic`. This example provides four appropriate data sets. The usage rule given for `BCFarfield` in Table 4 states which set of boundary-condition equations to be enforced based on the normal velocity and normal Mach number.

The data set for supersonic-inflow provides no information other than the boundary-condition type. A flow solver is free to apply any conditions that are appropriate; typically all solution quantities are set to freestream reference state values. The data set for subsonic-inflow states that three Dirichlet conditions should be enforced; the three data identifiers provided are among the list of conventions given in Appendix A.5. The data set for supersonic-outflow only provides the boundary-condition type, and the data set for subsonic-outflow provides one Dirichlet condition on the incoming acoustic characteristic, `CharacteristicAcousticMinus`.

Also provided in the example is the inward-pointing computational-coordinate normal; the normal points in the $-j$ direction, meaning the BC patch is a j-max face. This information could also be obtained from the BC patch description given in `IndexRange`.

Note that this example shows only the overall layout of the boundary-condition entity. `IndexRange` and all `DataArray_t` entities are abbreviated, and all unused structure functions are not evaluated.

Example 9-F: Viscous Solid Wall II

There are circumstances when a user may wish to define a BC patch using vertices (under `BC_t`), but store the BC data at face centers (under `BCDataSet_t`). The following example is similar to Example 9-B, with the exception that the Dirichlet data for temperature is stored at face centers rather than at vertices.

As before, the example is a viscous solid wall in a 3-D structured zone, where a Dirichlet condition is enforced for temperature; the wall temperature for the entire wall is specified to be 273 K. The BC patch is on the *j*-min face and is bounded by the indices (1,1,1) and (33,1,9).

```
!  CellDimension = 3, IndexDimension = 3
BC_t<3,3,3> BC2 =
  {{
  BCType_t BCType = BCWallViscousIsothermal ;

  !  Grid location is Vertex by default
  IndexRange_t<3> PointRange =
    {{
    int[3] Begin = [1 ,1,1] ;
    int[3] End   = [33,1,9] ;
    }} ;

  !  ListLength = 33*9 = 297
  BCDataSet_t<297> BCDataSet1 =
    {{
    BCTypeSimple_t BCTypeSimple = BCWallViscousIsothermal ;

    GridLocation_t GridLocation = FaceCenter ;
    IndexRange_t<3> PointRange =

      int[3] Begin = [1 ,1,1] ;
      int[3] End   = [32,1,8] ;
       ;

    !  ListLength = 32*8 = 256
    BCData_t<256> DirichletData =
      {{
      DataArray_t<real, 1, 1> Temperature =
        {{
        Data(real, 1, 1) = 273. ;

        DataClass_t DataClass = Dimensional ;

        DimensionalUnits_t DimensionalUnits =
          {{
          MassUnits        = Null ;
          LengthUnits      = Null ;
          TimeUnits        = Null ;
          TemperatureUnits = Kelvin ;
          AngleUnits       = Null ;
          }} ;
        }} ;
      }} ;
    }} ;
  }} ;
```

As in Example 9-B, although the boundary-condition data is global, we include in this example structure parameters that are the lengths of potential local-data arrays. In `BC_t`, `GridLocation` is not specified, and thus is `Vertex` by default. The structure function `ListLength` is 297, based on the specification of `PointRange`, and that value is passed to `BCDataSet_t`.

In this example `PointRange` is specified in `BCDataSet_t`, so the `ListLength` passed into it from `BC_t` is not used. In `BCDataSet_t`, `GridLocation` is specified as `FaceCenter`, and `PointRange` is set accordingly. The corresponding value of `ListLength` is 256, which is passed into `BCData_t`.

As before, in `BCData_t` the entity `Temperature` contains global data, so the value of `ListLength` is unused.

NormDefinitions be utilized to describe the convergence information recorded in the data arrays. The format used to describe the convergence norms in NormDefinitions is currently unregulated.

12.4 Discrete Data Structure Definition: DiscreteData_t

DiscreteData_t provides a description of generic discrete data (i.e., data defined on a computational grid); it is identical to FlowSolution_t except for its type name. This structure can be used to store field data, such as fluxes or equation residuals, that is not typically considered part of the flow solution. DiscreteData_t contains a list for data arrays, identification of grid location, and a mechanism for identifying rind-point data included in the data arrays. All data contained within this structure must be defined at the same grid location and have the same amount of rind-point data.

```
DiscreteData_t< int CellDimension, int IndexDimension, int VertexSize[IndexDimension],
                 int CellSize[IndexDimension] > :=
  {
  List( Descriptor_t Descriptor1 ... DescriptorN ) ;                        (o)

  GridLocation_t GridLocation ;                                             (o/d)

  Rind_t<IndexDimension> Rind ;                                             (o/d)

  IndexRange_t<IndexDimension> PointRange ;                                 (o)
  IndexArray_t<IndexDimension, ListLength[], int> PointList ;               (o)

  List( DataArray_t<DataType, IndexDimension, DataSize[]>
        DataArray1 ... DataArrayN ) ;                                       (o)

  DataClass_t DataClass ;                                                   (o)

  DimensionalUnits_t DimensionalUnits ;                                     (o)

  List( UserDefinedData_t UserDefinedData1 ... UserDefinedDataN ) ;         (o)
  } ;
```

Notes

1. Default names for the Descriptor_t, DataArray_t, and UserDefinedData_t lists are as shown; users may choose other legitimate names. Legitimate names must be unique within a given instance of DiscreteData_t and shall not include the names DataClass, DimensionalUnits, GridLocation, PointRange, PointList or Rind.
2. There are no required fields for DiscreteData_t. GridLocation has a default of Vertex if absent. Rind also has a default if absent; the default is equivalent to having an instance of Rind whose RindPlanes array contains all zeros (see Section 4.8).

3. Both of the fields `PointRange` and `PointList` are optional. Only one of these two fields may be specified.
4. The structure parameter `DataType` must be consistent with the data stored in the `DataArray_t` entities (see Section 5.1).
5. For unstructured zones `GridLocation` options are limited to `Vertex` or `CellCenter`, unless one of `PointRange` or `PointList` is present.
6. Indexing of data within the `DataArray_t` structures, must be consistent with the associated numbering of vertices or elements.

`DiscreteData_t` requires four structure parameters: `CellDimension` identifies the dimensionality of cells or elements, `IndexDimension` identifies the dimensionality of the grid size arrays, and `VertexSize` and `CellSize` are the number of core vertices and cells, respectively, in each index direction. For unstructured zones, `IndexDimension` is always 1.

The arrays of discrete data are stored in the list of `DataArray_t` entities. The field `GridLocation` specifies the location of the data with respect to the grid; if absent, the data is assumed to coincide with grid vertices (i.e., `GridLocation = Vertex`). All data within a given instance of `DiscreteData_t` must reside at the same grid location.

For structured grids, the value of `GridLocation` alone specifies the location and indexing of the discrete data. Vertices are explicity indexed. Cell centers and face centers are indexed using the minimum of the connecting vertex indices, as described in the section Structured Grid Notation and Indexing Conventions (Section 3.2).

For unstructured grids, the value of `GridLocation` alone specifies location and indexing of discrete data only for vertex and cell-centered data. The reason for this is that element-based grid connectivity provided in the `Elements_t` data structures explicitly indexes only vertices and cells. For data stored at alternate grid locations (e.g. edges), additional connectivity information is needed. This is provided by the optional fields `PointRange` and `PointList`; these refer to vertices, edges, faces or cell centers, depending on the values of `CellDimension` and `GridLocation`. The following table shows these relations.

`CellDimension`	`GridLocation`			
	`Vertex`	`EdgeCenter`	`*FaceCenter`	`CellCenter`
1	vertices	–	–	cells (line elements)
2	vertices	edges	–	cells (area elements)
3	vertices	edges	faces	cells (volume elements)

In the table, `*FaceCenter` stands for the possible types: `FaceCenter`, `IFaceCenter`, `JFaceCenter` or `KFaceCenter`.

Although intended for edge or face-based discrete data for unstructured grids, the fields `PointRange/List` may also be used to (redundantly) index vertex and cell-centered data. In all cases, indexing of discrete data corresponds to the element numbering as defined in the `Elements_t` data structures.

`Rind` is an optional field that indicates the number of rind planes (for structured grids) or rind points or elements (for unstructured grids) included in the data. Its purpose and function are identical to those described in Section 7.1. Note, however, that the `Rind` in this structure is independent of the `Rind` contained in `GridCoordinates_t`. They are not required to contain the same number of rind planes or elements. Also, the location of any discrete-data rind points is assumed to be consistent with the location of the core points (e.g., if `GridLocation = CellCenter`, rind points are assumed to be located at fictitious cell centers).

`DataClass` defines the default class for data contained in the `DataArray_t` entities. For dimensional data, `DimensionalUnits` may be used to describe the system of units employed. If present these two entities take precedence over the corresponding entities at higher levels of the CGNS hierarchy. The rules for determining precedence of entities of this type are discussed in Section 6.4.

The `UserDefinedData_t` data structure allows arbitrary user-defined data to be stored in `Descriptor_t` and `DataArray_t` children without the restrictions or implicit meanings imposed on these node types at other node locations.

FUNCTION `ListLength[]`**:**

return value: `int`
dependencies: `PointRange`, `PointList`

`DiscreteData_t` requires the structure function `ListLength`, which is used to specify the number of entities (e.g. vertices) corresponding to a given `PointRange` or `PointList`. If `PointRange` is specified, then `ListLength` is obtained from the number of points (inclusive) between the beginning and ending indices of `PointRange`. If `PointList` is specified, then `ListLength` is the number of indices in the list of points. In this situation, `ListLength` becomes a user input along with the indices of the list `PointList`. By "user" we mean the application code that is generating the CGNS database.

FUNCTION `DataSize[]`**:**

return value: one-dimensional `int` array of length `IndexDimension`
dependencies: `IndexDimension`, `VertexSize[]`, `CellSize[]`, `GridLocation`, `Rind`, `ListLength[]`

The function `DataSize[]` is the size of discrete-data arrays. It is identical to the function `DataSize[]` defined for `FlowSolution_t` (see Section 7.7).

12.5 Integral Data Structure Definition: `IntegralData_t`

`IntegralData_t` provides a description of generic global or integral data that may be associated with a particular zone or an entire database. In contrast to `DiscreteData_t`, integral data is not associated with any specific field location.

```
IntegralData_t :=
  {
  List( Descriptor_t Descriptor1 ... DescriptorN ) ;                    (o)
```

```
    List( DataArray_t<DataType, 1, 1> DataArray1 ... DataArrayN ) ;         (o)

    DataClass_t DataClass ;                                                  (o)

    DimensionalUnits_t DimensionalUnits ;                                    (o)

    List( UserDefinedData_t UserDefinedData1 ... UserDefinedDataN ) ;        (o)
    } ;
```

Notes

1. Default names for the Descriptor_t, DataArray_t, and UserDefinedData_t lists are as shown; users may choose other legitimate names. Legitimate names must be unique within a given instance of DiscreteData_t and shall not include the names DataClass or DimensionalUnits.
2. There are no required fields for IntegralData_t.
3. The structure parameter DataType must be consistent with the data stored in the DataArray_t entities (see Section 5.1).

DataClass defines the default class for data contained in the DataArray_t entities. For dimensional data, DimensionalUnits may be used to describe the system of units employed. If present these two entities take precedence over the corresponding entities at higher levels of the CGNS hierarchy. The rules for determining precedence of entities of this type are discussed in Section 6.4.

The UserDefinedData_t data structure allows arbitrary user-defined data to be stored in Descriptor_t and DataArray_t children without the restrictions or implicit meanings imposed on these node types at other node locations.

12.6 Family Data Structure Definition: Family_t

Geometric associations need to be set through one layer of indirection. That is, rather than setting the geometry data for each mesh entity (nodes, edges, and faces), they are associated to intermediate objects. The intermediate objects are in turn linked to nodal regions of the computational mesh. We define a CFD *family* as this intermediate object. This layer of indirection is necessary since there is rarely a 1-to-1 connection between mesh regions and geometric entities.

The Family_t data structure holds the CFD family data. Each mesh surface is linked to the geometric entities of the CAD databases by a name attribute. This attribute corresponds to a family of CAD geometric entities on which the mesh face is projected. Each one of these geometric entities is described in a CAD file and is not redefined within the CGNS file. A Family_t data structure may be included in the CGNSBase_t structure for each CFD family of the model.

The Family_t structure contains all information pertinent to a CFD family. This information includes the name attribute or family name, the boundary conditions applicable to these mesh regions, and the referencing to the CAD databases.

```
  Family_t :=
{
List( Descriptor_t Descriptor1 ... DescriptorN ) ;                     (o)

FamilyBC_t FamilyBC ;                                                  (o)

List( GeometryReference_t GeometryReference1 ... GeometryReferenceN ) ; (o)

RotatingCoordinates_t RotatingCoordinates ;                            (o)

List( UserDefinedData_t UserDefinedData1 ... UserDefinedDataN ) ;      (o)

int Ordinal ;                                                          (o)
} ;
```

Notes

1. All data structures contained in `Family_t` are optional.
2. Default names for the `Descriptor_t`, `GeometryReference_t`, and `UserDefinedData_t` lists are as shown; users may choose other legitimate names. Legitimate names must be unique at this level and must not include the names `FamilyBC`, `Ordinal`, or `RotatingCoordinates`.
3. The CAD referencing data are written in the `GeometryReference_t` data structures. They identify the CAD systems and databases where the geometric definition of the family is stored.
4. The boundary condition type pertaining to a family is contained in the data structure `FamilyBC_t`. If this boundary condition type is to be used, the `BCType` specified under `BC_t` must be `FamilySpecified`.
5. For the purpose of defining zone properties, families are extended to a volume of cells. In such case, the `GeometryReference_t` structures are not used.
6. The mesh is linked to the family by attributing a family name to a BC patch or a zone in the data structure `BC_t` or `Zone_t`, respectively.
7. `Ordinal` is defined in the SIDS as a user-defined integer with no restrictions on the values that it can contain. It may be used here to attribute a number to the family.

Rotation of the CFD family may be defined using the `RotatingCoordinates_t` data structure.

The `UserDefinedData_t` data structure allows arbitrary user-defined data to be stored in `Descriptor_t` and `DataArray_t` children without the restrictions or implicit meanings imposed on these node types at other node locations.

12.7 Geometry Reference Structure Definition: `GeometryReference_t`

The standard interface data structure identifies the CAD systems used to generate the geometry, the CAD files where the geometry is stored, and the geometric entities corresponding to the family. The `GeometryReference_t` structures contain all the information necessary to associate a CFD family to the CAD databases. For each `GeometryReference_t` structure, the CAD format is recorded in `GeometryFormat`, and the CAD file in `GeometryFile`. The geometry entity or entities within this CAD file that correspond to the family are recorded under the `GeometryEntity_t` nodes.

```
GeometryReference_t :=
  {
  List( Descriptor_t Descriptor1 ... DescriptorN ) ;                   (o)

  GeometryFormat_t GeometryFormat ;                                    (r)

  GeometryFile_t GeometryFile ;                                        (r)

  List (GeometryEntity_t GeometryEntity1 ... GeometryEntityN) ;        (o/d)

  List( UserDefinedData_t UserDefinedData1 ... UserDefinedDataN ) ;    (o)
  } ;
```

The `GeometryFormat` is an enumeration type that identifies the CAD system used to generate the geometry.

```
GeometryFormat_t := Enumeration(
  Null,
  NASA-IGES,
  SDRC,
  Unigraphics,
  ProEngineer,
  ICEM-CFD,
  UserDefined ) ;
```

Notes

1. Default names for the `Descriptor_t`, `GeometryEntity_t`, and `UserDefinedData_t` lists are as shown; users may choose other legitimate names. Legitimate names must be unique at this level and must not include the names `GeometryFile` or `GeometryFormat`.
2. By default, there is only one `GeometryEntity` and its name is the family name.
3. There is no limit to the number of CAD files or CAD systems referenced in a CGNS file. Different parts of the same model may be described with different CAD files of different CAD systems.
4. Other CAD geometry formats may be added to this list as needed.

12.8 Family Boundary Condition Structure Definition: `FamilyBC_t`

One of the main advantages of the concept of a layer of indirection (called a family here) is that the mesh density and the geometric entities may be modified without altering the association between nodes and intermediate objects, or between intermediate objects and geometric entities. This is very beneficial when handling boundary conditions and properties. Instead of setting boundary conditions directly on mesh entities, or on CAD entities, they may be associated to the intermediate objects. Since these intermediate objects are stable in the sense that they are not subject to mesh or geometric variations, the boundary conditions do not need to be redefined each time the model is modified. Using the concept of indirection, the boundary conditions and property settings are made independent of operations such as geometric changes, modification of mesh topology (i.e., splitting into zones), mesh refinement and coarsening, etc.

The FamilyBC_t data structure contains the boundary condition type. It is envisioned that it will be extended to hold both material and volume properties as well.

```
FamilyBC_t :=
  {
  BCType_t BCType;                                                   (r)

  List( FamilyBCDataSet_t BCDataSet1 ... BCDataSetN ) ;             (o)
  } ;
```

Notes

1. Default names for the FamilyBCDataSet_t list are as shown; users may choose other legitimate names. Legitimate names must be unique within a given instance of FamilyBC_t and shall not include the name BCType.

BCType specifies the boundary-condition type, which gives general information on the boundary-condition equations to be enforced. Boundary conditions are to be applied at the locations specified by the BC_t structure(s) associated with the CFD family.

The FamilyBC_t structure provides for a list of boundary-condition data sets. In general, the proper FamilyBCDataSet_t instance to impose on the CFD family is determined by the BCType association table (Table 4 on p. 117). The mechanics of determining the proper data set to impose is described in Section 9.8.

For a few boundary conditions, such as a symmetry plane or polar singularity, the value of BCType completely describes the equations to impose, and no instances of FamilyBCDataSet_t are needed. For "simple" boundary conditions, where a single set of Dirichlet and/or Neumann data is applied, a single FamilyBCDataSet_t will likely appear (although this is not a requirement). For "compound" boundary conditions, where the equations to impose are dependent on local flow conditions, several instances of FamilyBCDataSet_t will likely appear; the procedure for choosing the proper data set is more complex as described in Section 9.8.

12.9 Family Boundary Condition Data Set Structure Definition: FamilyBC-DataSet_t

FamilyBCDataSet_t contains Dirichlet and Neumann data for a single set of boundary-condition equations. Its intended use is for simple boundary-condition types, where the equations imposed do not depend on local flow conditions.

```
FamilyBCDataSet_t :=
{
List( Descriptor_t Descriptor1 ... DescriptorN ) ;                     (o)

BCTypeSimple_t BCTypeSimple ;                                          (r)

BCData_t<1> DirichletData ;                                            (o)
BCData_t<1> NeumannData ;                                              (o)

ReferenceState_t ReferenceState ;                                      (o)

DataClass_t DataClass ;                                                (o)

DimensionalUnits_t DimensionalUnits ;                                  (o)

List( UserDefinedData_t UserDefinedData1 ... UserDefinedDataN ) ;      (o)
} ;
```

Notes

1. Default names for the `Descriptor_t` and `UserDefinedData_t` lists are as shown; users may choose other legitimate names. Legitimate names must be unique within a given instance of `FamilyBCDataSet_t` and shall not include the names `BCTypeSimple`, `DataClass`, `DimensionalUnits`, `DirichletData`, `NeumannData` or `ReferenceState`.
2. `BCTypeSimple` is the only required field. All other fields are optional and the `Descriptor_t` list may be empty.

`BCTypeSimple` specifies the boundary-condition type, which gives general information on the boundary-condition equations to be enforced. `BCTypeSimple_t` is defined in Section 9.7 along with the meanings of all the `BCTypeSimple` values. `BCTypeSimple` is also used for matching boundary condition data sets as discussed in Section 9.8.

Boundary-condition data is separated by equation type into Dirichlet and Neumann conditions. Dirichlet boundary conditions impose the value of the given variables, whereas Neumann boundary conditions impose the normal derivative of the given variables. The mechanics of specifying Dirichlet and Neumann data for boundary conditions is covered in Section 9.9.

The substructures `DirichletData` and `NeumannData` contain boundary-condition data defined as globally constant over the family.

Reference quantities applicable to the set of boundary-condition data are contained in the `ReferenceState` structure. `DataClass` defines the default for the class of data contained in the boundary-condition data. If the boundary conditions contain dimensional data, `DimensionalUnits` may be used to describe the system of dimensional units employed. If present, these three entities take precedence of all corresponding entities at higher levels of the hierarchy. These precedence rules are further discussed in Section 6.4.

The `UserDefinedData_t` data structure allows arbitrary user-defined data to be stored in `Descriptor_t` and `DataArray_t` children without the restrictions or implicit meanings imposed on these node types at other node locations.

Note that `FamilyBCDataSet_t` is similar to the data structure `BCDataSet_t` (Section 9.4). The primary difference is that `FamilyBCDataSet_t` only allows for globally constant Dirichlet and Neumann data.

12.10 User-Defined Data Structure Definition: `UserDefinedData_t`

Since the needs of all CGNS users cannot be anticipated, `UserDefinedData_t` provides a means of storing arbitrary user-defined data in `Descriptor_t` and `DataArray_t` children without the restrictions or implicit meanings imposed on these node types at other node locations.

```
UserDefinedData_t :=
  {
  List( Descriptor_t Descriptor1 ... DescriptorN ) ;                    (o)

  GridLocation_t GridLocation ;                                         (o/d)

  IndexRange_t<IndexDimension> PointRange ;                             (o)
  IndexArray_t<IndexDimension, ListLength, int> PointList ;             (o)

  List( DataArray_t<> DataArray1 ... DataArrayN ) ;                     (o)

  DataClass_t DataClass ;                                               (o)

  DimensionalUnits_t DimensionalUnits ;                                 (o)

  FamilyName_t FamilyName ;                                             (o)

  List( UserDefinedData_t UserDefinedData1 ... UserDefinedDataN ) ;     (o)

  int Ordinal ;                                                         (o)
  } ;
```

Notes

1. Default names for the `Descriptor_t`, `DataArray_t`, and `UserDefinedData_t` lists are as shown; users may choose other legitimate names. Legitimate names must be unique within a given instance of `UserDefinedData_t` and shall not include the names `DataClass`, `DimensionalUnits`, `FamilyName`, `GridLocation`, `Ordinal`, `PointList`, or `PointRange`.
2. `GridLocation` may be set to `Vertex`, `IFaceCenter`, `JFaceCenter`, `KFaceCenter`, `FaceCenter`, `CellCenter`, or `EdgeCenter`. If `GridLocation` is absent, then its default value is `Vertex`. When `GridLocation` is set to `Vertex`, then `PointList` or `PointRange` refer to node indices, for both structured and unstructured grids. When `GridLocation` is set to `FaceCenter`, then `PointList` or `PointRange` refer to face elements.
3. `GridLocation`, `PointRange`, and `PointList` may only be used when `UserDefinedData_t` is located below a `Zone_t` structure in the database hierarchy.