CGNS Proposal for Extension 0050

DOF Storage and Interface Continuity

Version 1

August 4, 2026

CPEX#	0050
Scope	Record DOF storage layout (shared or independent) and, where DOFs are duplicated, whether the represented field is continuous across element interfaces
Champion	M. Scot Breitenfeld
Organization	The HDF Group
E-mail	brtnfld@hdfgroup.org
GitHub Issue	CGNS/CGNS#974
Reference Impl.	pending (see Section 12)
Target Release	CGNS 5.0, jointly with CPEX 0045 (see Section 12)
Date First Posted	April 21, 2026
Date of Last Revision	August 4, 2026
Prior Registry Entry	Listed while in draft as "Support Solution Continuity Type in CGNS" (`CPEX-0050-SolutionContinuity.pdf`); see Section 2.7
SIDS Status	accepted; awaiting implementation (adopted 2026-08-04)
Filemap Status	accepted; awaiting implementation (adopted 2026-08-04)
MLL Status	accepted; awaiting implementation (adopted 2026-08-04)

Contents

1 Abstract

This CPEX introduces two independent enumerations and their corresponding SIDS nodes: `DofStorage_t`, which records how degrees of freedom (DOFs) are stored relative to shared mesh entities, and `InterfaceContinuity_t`, which records what duplicated DOFs mean where they occur.

The two values of `DofStorage_t`—`SharedDofs` and `IndependentDofs`—answer a single question: for DOFs located at a geometric entity (vertex, edge, or face) that is shared between adjacent elements, does each adjacent element index the *same* buffer slot (shared storage), or does each element own its *own* independent slot (independent storage)? The question is always posed relative to a stated reference entity set—for a block indexed over volume elements those are the volume elements; for an interface block they are the volume elements adjacent to each listed face or edge (Section 4.2). This is a statement about data layout only; it makes no claim about the mathematical smoothness of the resulting field.

The gap being closed is specific. For every encoding legal today, the DOF layout is already fixed by `GridLocation_t` together with the SIDS `DataSize` rules, so this proposal is not a remedy for ambiguity. What cannot presently be expressed at all is an element-local layout at a shared-entity grid location: a solver storing per-element solution values at positions that coincide with shared mesh vertices has no conforming encoding, and the workaround observed in practice is to duplicate the grid coordinates, destroying the shared-vertex connectivity that mesh walking depends on (Section 2.2). Accordingly `IndependentDofs` at `GridLocation_t = Vertex`, together with the `PointList` ordering rule of Section 3.9, is the load-bearing content of this proposal; `SharedDofs` is retained for explicit self-description of the layout that the connectivity already implies. The node is defined for any polynomial order and any discretization family, and complements—without duplicating—the higher-order solution infrastructure of CPEX 0045 [4] and the integration-point infrastructure of CPEX 0047 [5], both of which are element-local by construction and require no such designator.

A second, independent attribute, `InterfaceContinuity_t`, records something the storage layout cannot: where DOFs are duplicated at element interfaces, whether the copies agree by construction. A continuous spectral-element field stored per element and a discontinuous Galerkin field stored per element are indistinguishable in CGNS today—same grid location, same sizing, same basis—yet only the former may be safely welded into a continuous reconstruction. This attribute is optional, is not verified by the library, and is orthogonal to `DofStorage_t`; it is the one item here that no other CGNS metadata supplies.

2 Motivation and Rationale

2.1 Problem Statement

Modern CFD and computational mechanics codes increasingly use discretization methods whose solution data cannot be fully described by the existing CGNS metadata [1]. The core issue is the treatment of degrees of freedom (DOFs) at shared geometric entities (vertices, edges, faces):

- In a **shared** layout (e.g., Continuous Galerkin FEM, C^0 spectral element methods), DOFs at shared interfaces are *shared*: a single value exists per geometric node, and the global solution field is at least C^0-continuous across element boundaries.
- In an **independent** layout (e.g., Discontinuous Galerkin [6, 7], finite volume, C^{-1} spectral element methods), DOFs are *element-local*: each element owns its own values at every point, including at shared interfaces. Multiple independent values may exist at the same geometric location.

This distinction affects:

1. **Data size and layout.** A DG solution on a mesh with shared vertices stores more values than a CG solution on the same mesh. Because the SIDS sizing rules for `Vertex` admit only the shared count, such a solution cannot be written at that grid location today; once a second sizing is admitted, a machine-readable discriminator is required so that readers can tell which of the two is in use.
2. **Visualization and interpolation.** Shared-DOF fields can be interpolated globally using the mesh connectivity; independent-DOF fields require element-local interpolation and may exhibit intentional discontinuities at element interfaces that should not be smoothed.
3. **Restart and data exchange.** Solvers reading restart files, and coupling frameworks exchanging data between codes, must know whether DOFs are shared or element-local to reconstruct the solution state and select the correct transfer operators. Where the grid location already settles that question this is a matter of convenience rather than necessity; where it does not, it is the difference between a usable file and an unreadable one.
4. **Error estimation and adaptation.** The jump in solution values across element interfaces in a DG solution is physically meaningful and is used for error estimation. A post-processor that incorrectly treats DG data as shared-DOF data would average out these jumps.

2.2 Evidence from Practice: Grid Duplication as a Workaround

The requirement addressed here was raised by implementors of third-party CGNS readers, whose difficulty is concrete rather than theoretical.[1] The substance of their report, paraphrased:

> At least one DG solver in production writes *duplicate grid coordinates* for nodes that are geometrically shared, so that the grid and the solution have matching extents and existing readers will accept the file. What is needed is a designator for the case of shared nodes in the grid but independent solution values at those nodes. Retaining shared grid nodes is what permits walking through the mesh—stream tracing, for example.

This is the expressiveness gap of Section 2.6 observed in the field, and it is the primary justification for this proposal. Because no designator exists, a solver wishing to emit element-local solution values at vertex positions must instead *explode the mesh*: every element receives its own copy of each vertex, the `GridCoordinates_t` array grows by the average vertex valence, and the shared-vertex connectivity that makes mesh walking possible is destroyed. The cost is borne twice—once in file size, and again by every downstream reader that can no longer traverse the mesh.

Two points about this evidence deserve emphasis, because they determine how narrow the proposal should be:

- The requirement is specifically for *independent* DOFs at `GridLocation_t = Vertex` while the grid retains shared vertices. It is a request for one designator value, not for a general classification of DOF storage.
- The constituency is third-party *readers*, not solver authors. This is what distinguishes this proposal from the alternative of requiring CPEX 0045's `InterpolationPoints`: a reader that must implement `Family_t`-level `SolutionInterpolation_t` lookup, basis evaluation, and control-point handling in order to interpret a first-order DG field bears a cost out of proportion to the data. Where the DOF positions simply *are* the mesh vertices, the difference is between reading one additional scalar node and implementing an interpolation framework.

  No claim of earlier availability is intended or implied. This proposal and CPEX 0045 are being developed together and target the same release, so neither is the faster route to the capability; the argument for this one rests on reader-side implementation cost alone. Joint development is itself the reason the two can be relied upon to interlock: shared vocabulary, a single pass through the MLL and `cgnscheck`, and no possibility of the two describing the same data in

[1] Raised in correspondence with the CPEX champion, April 2026. Specific parties and codes are intentionally not identified.

contradictory terms.

A second, independent instance of the same workaround. The pattern is not specific to CGNS. Reviewers of this proposal report that at least one widely used commercial post-processing format forces the identical workaround: although it acquired high-order support, that support remains restricted to nodal—and therefore continuous—values, so discontinuous fields must be emitted either by duplicating geometry or by writing each element as its own topologically independent region. The same limitation is reported to have been a principal motivation for CPEX 0045. Two unrelated formats exhibiting the same defect indicates that this is a general gap in how solution data is modelled, rather than a CGNS-specific oversight, and it means CPEX 0045 and this proposal are two approaches to one problem rather than competitors.

2.3 Inadequacy of Existing CGNS Constructs

No existing CGNS mechanism captures the shared/independent distinction:

`GridLocation_t` specifies *where* data is located (vertices, cell centers, faces, edges, interpolation points, or integration points), but not whether values at shared locations are shared or element-local. A `Vertex`-located solution could use either layout; the same applies to `InterpolationPoints` and `IntegrationPoint`.

`PointSet` (`PointList`, `PointRange`) can index subsets of data but does not encode DOF-sharing semantics.

`InterpolationType_t` (CPEX 0045 [4]) describes the polynomial basis (e.g., Lagrange, monomial), but not whether that basis yields a globally shared or element-local DOF set.

`SolutionInterpolation_t` (CPEX 0045, a child of `Family_t`) records the interpolation type and control-point layout for a given basic element type; the polynomial orders are recorded separately in the `InterpolationOrders` child of `FlowSolution_t`. Neither carries DOF-sharing information.

`ItgPointsStartOffset` (CPEX 0047 [5]) gives the per-element starting offsets of integration-point data, which implies an element-local layout for that particular grid location, but says nothing about vertex-, edge-, or face-located DOFs and does not generalize to other grid locations.

Data array size together with `GridLocation_t` *does* determine the layout for every encoding legal before this CPEX, and it should be stated plainly that this is so: the SIDS `DataSize` formula admits exactly one layout at each grid location (Section 6.2 tabulates them), and it does so irrespective of element mix. What the sizing rules cannot do is admit an element-local layout at a

shared-entity grid location at all—that layout is not merely ambiguous, it is inexpressible. The gap this CPEX addresses is therefore one of expressiveness, not of ambiguity; see Section 2.6.

2.4 Scope and Non-Goals

This proposal is deliberately narrow. Stating plainly what it does and does not close is the clearest way to show that it neither overlaps CPEX 0045 nor CPEX 0047.

What it closes. One gap: element-local solution DOFs at positions coinciding with shared mesh vertices, with the shared-vertex grid retained. This is the case of Section 2.2, it has no conforming encoding today, and it is not restricted to high-order methods—DG $p = 1$ on a linear mesh is the most common instance, and the predefined element types extend the same pattern through fourth order. Because the affected node is `FlowSolution_t` (and its `DiscreteData_t`, `ZoneSubRegion_t`, and `UserDefinedData_t` counterparts), the designator belongs there rather than inside the higher-order interpolation infrastructure.

What it does not close, by design.

- **Nothing that is already determined.** At `CellCenter`, under CPEX 0045's `InterpolationPoints`, and under CPEX 0047's `IntegrationPoint`, the layout follows from the grid location and the sizing rules. Recording it there is permitted, and recommended for uniformity, but it resolves nothing and this proposal does not claim otherwise.
- **Higher-order DOF identification.** Establishing which DOFs of two adjacent elements correspond across a shared face—the general dof-relation—requires per-element index mapping and orientation conventions that this proposal does not supply. For arbitrary interpolation points it cannot be done at all from the metadata CGNS currently defines (Section 4.2).
- **Smoothness order.** Whether duplicated DOF values agree by construction *is* recorded, by the separate `InterfaceContinuity` attribute of Section 3.3—it is a distinct property and not derivable from storage layout. What remains out of scope is the *order* of smoothness: C^k for $k \geq 1$ is a property of the basis and belongs in CPEX 0045's interpolation metadata.
- **Structured zones**, `NFACE_n` sections, and vector-conforming space orientation (Sections 3.9.1 and 3.2.1).

An acknowledged redundancy. This was the subject of the one question put to the Steering Committee, decided 2026-08-04; see Section 2.5. For discontinuous Galerkin data with $p \leq 4$ on a Lagrange basis there will be two con-

forming ways to express the same field: the `Vertex` encoding defined here, and CPEX 0045's `InterpolationPoints`. This is stated plainly because the cost is borne asymmetrically—writers gain a choice, while readers wishing to interpret all conforming files must support both. Tolerating alternative encodings of similar data is long-standing CGNS practice, and it is what makes the standard easy to write against, but it is a genuine burden on reader implementors and should not be presented as free.

Two things limit the cost. First, **CPEX 0045 is the recommended route wherever it is available**: it needs no redundant index list, it extends to any order, and it is the better mechanism where the DOF positions are not the mesh vertices. The encoding here is offered as the lower-adoption-cost path, not the preferred one. Second, a reader that implements only CPEX 0045 is never *wrong*—it is simply unable to read files that chose the other encoding, and can say so precisely, because the `DofStorage` node makes that choice explicit rather than leaving it to be inferred.

2.5 Decision by the Steering Committee

Everything else in this proposal was settled against the existing standard and the library source, and is presented as specification rather than as a question. One question could not be settled by evidence, because it concerned how the standard should be organised; it was put to the Committee.

Should the `Vertex` encoding of Section 3.9 live in this CPEX, or be folded into CPEX 0045?

Both encodings satisfy the requirement, and the grid is untouched either way. What the `Vertex` encoding offers is a smaller reader-side burden where the DOF positions simply *are* the mesh vertices: no `Family_t`-level `SolutionInterpolation_t` lookup, no basis evaluation, no control-point machinery. For first-order DG that is the difference between reading one additional scalar node and implementing an interpolation framework, and it was third-party reader implementors, not solver authors, who asked for it. The cost is the redundancy described above.

1. **Keep it here**, with CPEX 0045 recommended wherever available. The proposal as written takes this position.
2. **Fold the ordering rule into CPEX 0045** as an alternative encoding there. One proposal, one place to look, at the cost of enlarging CPEX 0045 late in its review and coupling two schedules that are currently only aligned.
3. **Decline the `Vertex` encoding** and require CPEX 0045. Removes the redundancy, and with it the low-cost path the requesters asked for.

Decided (2026-08-04 Steering Committee meeting): Option 1. The `Vertex` encoding stays in CPEX-0050, with CPEX-0045 recommended wherever available.

The normative content of Section 3.9.3 is exactly as specified below; the question was only ever which document it lives in. CPEX-0050 was adopted the same day for SIDS, Filemap and MLL, with the reference implementation of Section 12 still pending.

2.6 Prior Art: The DOF-Map Concept in ASCI-DMF / SAF

The concept encoded by `DofStorage_t` has well-established prior art in the scientific data management literature. The ASCI Data Model Framework (ASCI-DMF) and its implementation in the Sets and Fields (SAF) library [8] introduced the *dof-relation*: given a flat buffer of DOF values, the dof-relation enumerates for each element which buffer indices are assigned to it. When two adjacent elements refer to the *same* buffer index for a shared interface entity, that DOF is shared—this is `SharedDofs` in the present proposal. When each element refers to *different* buffer indices (each element owns its own slot), the DOFs are independent—corresponding to `IndependentDofs`.

Miller [8] notes that for piecewise-linear coordinate fields the dof-relation and the mesh connectivity (topo-relation) are one and the same array, and are consequently "hopelessly confused" in practice. CGNS today is in exactly this position: it has no dof-relation layer, so the only dof-map available to a solution at a shared-entity grid location is the one implied by the `Elements_t` connectivity. This does not make existing files ambiguous—a conforming file is unambiguous, because `GridLocation_t` together with the SIDS `DataSize` formula fixes the layout in every legal case (Section 6.2 tabulates them). The cost is expressiveness: because the implied dof-map *is* the connectivity, a solution whose DOFs are element-local cannot be written at `Vertex`, `EdgeCenter`, or `FaceCenter` at all. CPEX 0045 supplies an independent per-element dof-map for `InterpolationPoints`, and CPEX 0047 does so for `IntegrationPoint`; both are element-local by construction. What remains uncovered is element-local DOFs at `Vertex` on the predefined element types, where `DofStorage_t` serves as the discriminator that makes a second sizing both legal and self-describing.

It is important to note that `DofStorage_t` encodes only the *coarsest* level of DOF-map metadata—shared versus independent—which is sufficient for the vast majority of post-processing and data-exchange use cases. The full generality of a dof-relation (arbitrary per-element index remapping, as required for quadratic or higher-order elements whose mid-edge DOFs must be identified across elements) is the domain of CPEX 0045 [4] and CPEX 0047 [5]. These CPEXs are therefore complementary layers of the same concept introduced in ASCI-DMF.

2.7 Naming History

Two earlier working names left traces a reader may encounter. `SolutionRepresentation_t` is the source of the former filename `CPEX-0050-solution-representation.tex`.

`SolutionContinuity_t` is the name under which the CPEX registry currently lists this entry (scope line "Support Solution Continuity Type in CGNS," attachment `CPEX-0050-SolutionContinuity.pdf`); it was abandoned because readers consistently took "continuity" as a claim about mathematical smoothness, which `DofStorage_t` does not make. **The registry entry requires a corresponding update** to its scope line and attachment. No files have been written with either earlier name, so no migration path is required.

3 Detailed Description

3.1 Design Overview

The proposal introduces:

1. A new `DofStorage_t` enumeration with values covering the common DOF storage layouts (Section 3.2).
2. A new optional child node, `DofStorage`, of `FlowSolution_t`, `DiscreteData_t`, `ZoneSubRegion_t`, and `UserDefinedData_t`, recording the DOF storage layout (Section 3.4).
3. One pair of position-based Mid-Level Library [3] functions to read and write the DOF storage layout, following the pattern established by `cg_gridlocation_read` / `cg_gridlocation_write` (Sections 3.5 and 3.8).
4. A second, independent enumeration `InterfaceContinuity_t` and its optional child node `InterfaceContinuity`, on the same four parents, recording whether duplicated DOFs represent a field that is continuous across element interfaces (Section 3.3). This answers a different question from `DofStorage_t` and is the one piece of information here that no other CGNS metadata can supply.
5. A second pair of position-based functions for that attribute, built on the same address-helper mechanism.
6. Fortran bindings for both pairs (Example 7.4).

3.2 New Enumeration: `DofStorage_t`

```
typedef enum {
  CGNS_ENUMV( DofStorageNull )        = CG_Null,
  CGNS_ENUMV( DofStorageUserDefined ) = CG_UserDefined,
  CGNS_ENUMV( SharedDofs )            = 2,
  CGNS_ENUMV( IndependentDofs )       = 3
} CGNS_ENUMT( DofStorage_t );

#define NofValidDofStorage 4

extern CGNSDLL const char *DofStorageName[NofValidDofStorage];
```

The enumeration values are:

`DofStorageNull` No DOF storage layout specified. This is the default for backward compatibility: existing files and codes that do not set this field are unaffected. See Section 6.2 for the recommended interpretation of legacy data.

`DofStorageUserDefined` Application-defined DOF storage layout not covered by the standard values. When this value is used, writers *shall* attach a `Descriptor_t` child node named `"DofStorageDescription"` to the same parent node. The string value identifies the storage convention (e.g., `"HDG-trace"`, `"C1-spline"`). This allows readers to recognize the convention without requiring a separate side-channel; see Example 7.3 for the reader side.

`SharedDofs` DOFs at shared mesh entities (vertices, edges, faces) are *shared* between adjacent elements: each shared entity carries exactly one value. This is a statement about *storage and DOF-sharing semantics*, not about the mathematical smoothness of the field. Typical of: Continuous Galerkin FEM, C^0 spectral element methods, isogeometric analysis with C^0 junctions. See the caveat on vector-conforming spaces below.

`IndependentDofs` The solution field is element-local: each element owns independent DOFs, including at shared geometric locations, and no interface DOF is shared between adjacent elements. Typical of: Discontinuous Galerkin methods, finite volume methods, C^{-1} spectral element methods.

3.2.1 Caveat: Vector-Conforming Spaces

$\boldsymbol{H}(\mathrm{div})$-conforming spaces (e.g., Raviart–Thomas) and $\boldsymbol{H}(\mathrm{curl})$-conforming spaces (e.g., Nédélec) store one value per shared interface entity—a normal-flux or tangential-component DOF, respectively—and so satisfy the letter of the `SharedDofs` definition, even though the full vector field is not pointwise C^0-continuous.

However, such DOFs additionally carry a per-element **orientation convention**: the sign of the stored value depends on a choice of outward normal ($\boldsymbol{H}(\mathrm{div})$) or edge tangent direction ($\boldsymbol{H}(\mathrm{curl})$), and adjacent elements see the same DOF with opposite sign. `DofStorage_t` does not encode that orientation, and no existing CGNS construct does either.

Accordingly:

- A writer *may* label such a field `SharedDofs`, since the storage statement is accurate.
- A writer that does so *shall* also mark the field with `DofStorageUserDefined`-style provenance—either by using `DofStorageUserDefined` with a `DofStorageDescription` of `"Hdiv-RT"` or `"Hcurl-Nedelec"`, or by attaching a `Descriptor_t` identifying the space—so that readers are not misled.

- A reader *shall not* interpolate, average, or otherwise treat `SharedDofs` DOFs as pointwise nodal values of a scalar field without first establishing that the field is nodal. Applying nodal interpolation to Raviart–Thomas or Nédélec DOFs yields incorrect results.

It is tempting to reach instead for `SharedDofs` + `DiscontinuousInterface`, reading the pair as "one stored value per interface entity, but a vector field that is not continuous across that interface." As a description of $\boldsymbol{H}(\mathrm{div})$ or $\boldsymbol{H}(\mathrm{curl})$ data that reading is apt, but it is not the one `InterfaceContinuity_t` carries: that attribute compares *duplicated stored values* for numerical agreement (Section 3.3.2), and under `SharedDofs` nothing is duplicated, so the pair is rejected as contradictory (Section 3.3). Admitting it would also not be sufficient: a reader learning only "not fully continuous" still cannot act, because it does not know which component the shared DOF constrains or the sign convention relating the two adjacent elements. The label would advertise a self-describing file that is not one, which is the failure mode this section exists to prevent. Of the two routes above, therefore, a writer taking the first—`DofStorageUserDefined` with a `DofStorageDescription`—never encounters that cell at all, and a writer keeping `SharedDofs` should leave `InterfaceContinuity` unset and identify the space in the accompanying `Descriptor_t`.

Supplying the orientation metadata that would make vector-conforming spaces safely self-describing is deliberately out of scope for this CPEX and is a natural subject for a future, narrowly scoped proposal. Such a proposal is also the right place to introduce a positive encoding for conformity of a vector space, rather than overloading a cell of this one—which would fix that meaning permanently, on the strength of a reading that cannot yet be acted upon.

3.2.2 Classification Guidance

The two values are not of equal weight. `IndependentDofs` is load-bearing: at `GridLocation_t = Vertex` it admits a layout that has no other conforming encoding, and a reader must consult it to size and interpret the field correctly (Section 3.9). `SharedDofs` is confirmatory—wherever it is valid, the sharing it describes is already realized by the `Elements_t` connectivity—and is provided so that files are self-describing, not because readers require it. It is meaningless at $p = 0$ and invalid on a volume `InterpolationPoints` block (Section 4.2).

Writers should classify as follows.

HDG and mixed-layout methods store element-local volume DOFs and face-local trace DOFs in *separate* `FlowSolution_t` nodes, marked `IndependentDofs` and `SharedDofs` respectively. On a trace block, `SharedDofs` asserts only that the two volume elements adjacent to a given face index the same DOF block for that face; it asserts nothing about continuity across the skeleton, which the standard hybridizable trace space does not have.

Penalized / weakly-continuous DG methods use `IndependentDofs`. The data layout is element-local; "weak continuity" is a property of the variational formulation, not of the stored data.

Isogeometric analysis (C^k, $k \geq 1$) uses `SharedDofs`. DOFs are shared at patch interfaces; the smoothness order k belongs in CPEX 0045's interpolation metadata.

`DofStorageUserDefined` remains available as an escape hatch, consistent with CGNS convention for all enumeration types.

3.2.3 Enumeration Value Naming

For every CGNS enumeration the string written to the file is the enumerator's own name, so the choice of identifier is also a choice of file-format token. Only `Null` and `UserDefined` are stored abbreviated, as `"Null"` and `"UserDefined"`.

Where CGNS wants a value to be unmistakable it uses a *compound descriptive* name rather than a type-stem prefix: `CellCenter`, `FaceCenter`, `TimeAccurate`, `ConstantRate`, `NonDeformingGrid`. No CGNS enumeration prefixes its values with its own type stem. The values adopted here follow the compound pattern:

Enumeration	Values
`DofStorage_t`	`SharedDofs`, `IndependentDofs`
`InterfaceContinuity_t`	`ContinuousInterface`, `DiscontinuousInterface`

None collides with any identifier in `cgnslib.h`. The mild repetition in `DofStorage = SharedDofs` has direct precedent in `ArbitraryGridMotionType_t`, whose values are `NonDeformingGrid` and `DeformingGrid`. The `Null` and `UserDefined` members do carry the type-stem prefix in the C bindings, per the convention for those two members specifically.

3.3 New Enumeration: `InterfaceContinuity_t`

`DofStorage_t` records how DOFs are *stored*. It says nothing about what the stored values *mean* where storage is duplicated, and that turns out to be the one property in this problem space that no existing CGNS metadata can supply.

Consider two files that are identical in every respect CGNS can currently express—same `GridLocation_t`, same array length, same CPEX 0045 basis and order, same element set, and both with duplicated DOFs at element interfaces:

	Stored	Duplicated values at an interface
CG spectral element	per element	agree, by construction
Discontinuous Galerkin	per element	differ, meaningfully

Nothing distinguishes them. A post-processor must therefore assume the worst case— that every interface carries a genuine jump—and can never safely weld duplicated DOFs, average them, or reconstruct a continuous field, even when the discretization guarantees that doing so is exact. An error estimator cannot separate a physical jump from one that is zero by construction. The information is not derivable from the layout, because both files have the same layout.

```
typedef enum {
  CGNS_ENUMV( InterfaceContinuityNull )        = CG_Null,
  CGNS_ENUMV( InterfaceContinuityUserDefined ) = CG_UserDefined,
  CGNS_ENUMV( ContinuousInterface )            = 2,
  CGNS_ENUMV( DiscontinuousInterface )         = 3
} CGNS_ENUMT( InterfaceContinuity_t );

#define NofValidInterfaceContinuity 4

extern CGNSDLL const char
    *InterfaceContinuityName[NofValidInterfaceContinuity];
```

`InterfaceContinuityNull` Unspecified; the default, and the state of every file predating this CPEX. A reader *shall not* assume continuity from its absence: the safe interpretation is to preserve all duplicated values as distinct.

`ContinuousInterface` Wherever storage duplicates a DOF at a shared geometric entity, all copies hold the same value, by construction of the discretization. The duplication is redundancy, not information: a reader may weld, average, or interpolate across element interfaces without loss.

`DiscontinuousInterface` Duplicated values may differ, and the differences are meaningful. Readers must not average or conflate them. This is the expected value for DG, finite volume, and any method whose interface jumps carry physical or numerical content.

`InterfaceContinuityUserDefined` As with `DofStorageUserDefined`, writers *shall* attach a `Descriptor_t` child named `"InterfaceContinuity Description"` to the same parent.

3.3.1 Relationship to `DofStorage_t`

The two attributes are orthogonal, and each combination has a meaning:

	`ContinuousInterface`	`DiscontinuousInterface`
`SharedDofs`	CG; redundant†	inconsistent‡
`IndependentDofs`	CG stored per element	DG, FV, spectral C^{-1}

† Shared storage realizes continuity through the connectivity itself, so the annotation adds nothing derivable. It is permitted for uniformity.

‡ Contradictory: under `SharedDofs` no interface DOF is duplicated, so there are no differing copies for `DiscontinuousInterface` to assert. The asymmetry with the † cell is deliberate—there `ContinuousInterface` is trivially true and merely redundant, whereas `DiscontinuousInterface` asserts something the storage contradicts. `cgnscheck` should report an error (Section 11). This cell is *not* the encoding for a vector-conforming space whose single shared normal or tangential DOF leaves the full vector field discontinuous across the interface; see Section 3.2.1.

The load-bearing cell is `IndependentDofs` + `ContinuousInterface`: a continuous field stored redundantly, which is exactly the CG spectral-element case above and the case that motivates this attribute. It is also the combination a reader most benefits from knowing, since it is the only one in which welding is both safe and profitable.

3.3.2 Scope of the Assertion

Three limits are deliberate, and stating them is what keeps this attribute narrow enough to be useful.

It records agreement of stored values, not smoothness order. `ContinuousInterface` asserts that duplicated DOFs coincide, which for a nodal basis means the field is at least C^0 across the interface. It does not encode the smoothness order of the underlying basis: C^k for $k \geq 1$, as in isogeometric analysis, is a property of the basis and belongs in CPEX 0045's interpolation metadata, exactly as Section 2.4 states for the storage attribute. Nor is it a general statement about the continuity of the function space: the question it answers is the narrow, numerical one of whether two stored copies of one DOF hold the same number, because that is the question a post-processor must answer before welding them. A space can satisfy `ContinuousInterface` and still be discontinuous in some other component or derivative, and a space with no duplicated storage at all can be discontinuous without this attribute having anything to say (Section 3.2.1).

It is a writer's assertion, and the library does not verify it. The MLL shall not compare values on write; doing so would require reading the field arrays and resolving the DOF correspondence, which for arbitrary interpolation points is not generally possible (Section 4.2). `cgnscheck` *may* sample duplicated DOFs numerically and warn on disagreement; see Section 11. A writer that does not know shall write nothing, leaving `InterfaceContinuityNull`.

It is meaningful only where DOFs are duplicated. That is: under `IndependentDofs`, under `GridLocation_t = InterpolationPoints` and `IntegrationPoint`, and under the `Vertex` plus `PointList` encoding of Section 3.9. Where no duplication occurs—a `CellCenter` solution, or genuinely shared vertex storage—writers should omit it. The two values are not equally harmless there: `ContinuousInterface` is vacuous rather than wrong, since it is trivially satisfied by an empty set of duplicates, whereas `DiscontinuousInterface` asserts differing

copies that do not exist. Where the node also carries `SharedDofs` the two assertions contradict each other outright and `cgnscheck` should report an error; elsewhere—a `CellCenter` solution, say—the value is merely unsupportable and draws a warning, matching the severity this document gives the analogous `SharedDofs` misuse (Section 11).

3.4 SIDS Node Definition

The DOF storage layout is stored as a new optional child node of `FlowSolution_t`. The amended SIDS definition adds one line (shown with the `+` marker). The definition below also shows the `InterpolationOrders` and `CharacteristicLength` children introduced by CPEX 0045 [4], for context; note that CPEX 0045's `SolutionInterpolation_t` node is a child of `Family_t`, not of `FlowSolution_t`.

`FlowSolution_t` (amended, SIDS Section 7.7)

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

GridLocation_t GridLocation ;                                 (o/d)

IndexArray_t<1, 2, int> InterpolationOrders ;                 (o/d)
(CPEX 0045)
DataArray_t<R8, 1|2, ...> CharacteristicLength ;               (o)
(CPEX 0045)

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)

+   DofStorage_t DofStorage ;                                  (o)
+   InterfaceContinuity_t InterfaceContinuity ;                (o)

List( UserDefinedData_t UserDefinedData1 ...
```

```
UserDefinedDataN ) ;                                    (o)
}
```

Both new children are **optional** (o) and independent of one another. When `DofStorage` is absent the storage layout is unspecified (`DofStorageNull`); when `InterfaceContinuity` is absent the interface behaviour is unspecified (`InterfaceContinuityNull`). Either may be written without the other, and absence of both leaves a node indistinguishable from one written by a pre-CPEX-0050 library, preserving full backward compatibility.

The same optional child node is added to `DiscreteData_t` (SIDS Section 12.4), using identical structure and semantics (Example 7.6):

`DiscreteData_t` (amended, SIDS Section 12.4)

```
DiscreteData_t< int CellDimension, int IndexDimension,
int VertexSize[IndexDimension],
int CellSize[IndexDimension] > :=
{
...  (existing children unchanged) ...

+   DofStorage_t DofStorage ;                           (o)
+   InterfaceContinuity_t InterfaceContinuity ;         (o)
}
```

The optional child node is also added to `ZoneSubRegion_t` (SIDS Section 7.9). `ZoneSubRegion_t` holds `DataArray_t` nodes for localized solutions, boundary fluxes, or fields on sub-regions where the shared/independent distinction applies equally (Example 7.6):

`ZoneSubRegion_t` (amended, SIDS Section 7.9)

```
ZoneSubRegion_t< int RegionCellDimension > :=
{
...  (existing children unchanged) ...

+   DofStorage_t DofStorage ;                           (o)
+   InterfaceContinuity_t InterfaceContinuity ;         (o)
}
```

Finally, the child node is added to `UserDefinedData_t` (SIDS Section 12.10 today; see the note on renumbering below). `UserDefinedData_t` is the standard generic extension point and already accepts `GridLocation_t` and `PointSet` children; per-

mitting `DofStorage_t` there keeps the two consistent and costs nothing, because the position-based API of Section 3.5 covers all parents through a single code path:

`UserDefinedData_t` (amended, SIDS Section 12.10)

```
UserDefinedData_t :=
{
...  (existing children unchanged) ...

+   DofStorage_t DofStorage ;                          (o)
+   InterfaceContinuity_t InterfaceContinuity ;        (o)
}
```

3.4.1 Node Placement Rationale

The node is placed as a child of `FlowSolution_t` (rather than, for example, `Zone_t`) because:

- The DOF storage layout is a property of the *solution field*, not of the mesh or the zone.
- A single zone may contain multiple `FlowSolution_t` nodes with different layouts—for example, an HDG solver storing both volume (independent) and trace (shared) solutions, or a multi-physics code storing a shared-DOF displacement field alongside an independent-DOF stress field.
- This parallels the existing `GridLocation_t` child of `FlowSolution_t`, which is also a per-solution-node property.
- Placing it within CPEX 0045's `SolutionInterpolation_t` would be doubly wrong: that node is a child of `Family_t`, so a per-family setting could not distinguish two solutions on the same family, and it would incorrectly limit the scope to higher-order solutions when the distinction applies at any order.

Design Decision 5 records why a new node type was preferred to a new `GridLocation_t` value.

3.5 New MLL Functions

Each attribute is read and written by one position-based function pair operating on the node at the current position, as set by `cg_goto`, backed by an internal `cgi_`*x*`_address()` helper that resolves that position to the relevant struct field and enumerates the legal parent labels. This is the pattern CGNS already uses for every other optional scalar metadata child—`GridLocation`, `DataClass`, `Rind`, `DimensionalUnits`, `Descriptor`—so usage matches the sibling property exactly:

`cg_goto(...); cg_gridlocation_write(CellCenter);` Design Decision 4 records why this was preferred to a family of per-parent, explicit-index functions.

3.5.1 C API

```
/*
 * Write the DOF storage layout of the node at the current position,
 * which must have been set by cg_goto()/cg_gopath().
 *
 * Valid parent labels: FlowSolution_t, DiscreteData_t,
 *                      ZoneSubRegion_t, UserDefinedData_t.
 *
 * DofStorage - DofStorage_t value.  DofStorageNull is rejected with
 *              CG_ERROR: "unspecified" is expressed by not writing
 *              the node.  To remove an existing child, call
 *              cg_delete_node("DofStorage") at the same position in
 *              CG_MODE_MODIFY.
 *
 * Returns CG_OK, or CG_ERROR / CG_INCORRECT_PATH on failure.
 */
CGNSDLL int cg_dof_storage_write(
    CGNS_ENUMT(DofStorage_t) DofStorage);

/*
 * Read the DOF storage layout of the node at the current position.
 *
 * DofStorage - [out] DofStorage_t value.  Set to DofStorageNull,
 *              with a CG_OK return, when the node has no
 *              DofStorage child (legacy file or unset).
 *
 * Returns CG_OK, or CG_ERROR / CG_INCORRECT_PATH on failure.
 */
CGNSDLL int cg_dof_storage_read(
    CGNS_ENUMT(DofStorage_t) *DofStorage);

/*
 * Write / read the interface continuity of the node at the current
 * position.  Parent labels, overwrite behaviour, and the treatment of
 * the Null value are exactly as for cg_dof_storage_write/read:
 * InterfaceContinuityNull is rejected on write, and reading a node
 * that has none yields InterfaceContinuityNull with CG_OK.
 *
 * The library does not validate the assertion against the field data.
 */
CGNSDLL int cg_interface_continuity_write(
    CGNS_ENUMT(InterfaceContinuity_t) InterfaceContinuity);

CGNSDLL int cg_interface_continuity_read(
    CGNS_ENUMT(InterfaceContinuity_t) *InterfaceContinuity);

/*
 * Return the string name for a DofStorage_t value.
 * Matches the convention of cg_GridLocationName, cg_DataTypeName.
 * Returns NULL for out-of-range values.
 */
CGNSDLL const char *cg_DofStorageName(
    CGNS_ENUMT(DofStorage_t) type);
```

Behavior:

- `cg_dof_storage_write` creates the `DofStorage` child node under the node at the current position. In `CG_MODE_MODIFY`, an existing `DofStorage` child is deleted first. Writing `DofStorageNull` deletes an existing child without creating a replacement.
- `cg_dof_storage_read` returns the value cached in the internal structure by the tree parser. If the node does not exist (legacy file or unset), it returns `CG_OK` with `DofStorageNull`, never an error.
- Both functions return `CG_INCORRECT_PATH` if the current position is not one of the four permitted parent labels, matching `cg_gridlocation_read/_write` behavior.

3.5.2 Fortran API

```
subroutine cg_dof_storage_write_f(dofStorage, ier)
  integer, intent(in)  :: dofStorage
  integer, intent(out) :: ier

subroutine cg_dof_storage_read_f(dofStorage, ier)
  integer, intent(out) :: dofStorage
  integer, intent(out) :: ier
```

The interface-continuity pair follows identically:

```
call cg_interface_continuity_write_f(interfaceContinuity, ier)
call cg_interface_continuity_read_f(interfaceContinuity, ier)
```

3.6 Internal Data Structure Changes

Four structures in `cgns_header.h` each gain one field: `cgns_sol` (for `FlowSolution_t`), `cgns_discrete` (for `DiscreteData_t`), `cgns_subreg` (for `ZoneSubRegion_t`), and `cgns_user_data` (for `UserDefinedData_t`).

```
typedef struct {            /* FlowSolution_t node */
    /* ... existing fields ... */

    /* CPEX 0050 */
    CGNS_ENUMT(DofStorage_t)          dof_storage;
    CGNS_ENUMT(InterfaceContinuity_t) interface_continuity;
} cgns_sol;

/* cgns_discrete, cgns_subreg, and cgns_user_data gain the
 * identically named and typed fields. */
```

The two fields are independent: neither read nor write of one touches the other.

The `dof_storage` field defaults to `DofStorageNull` (value 0), which is both the zero-initialization value and the "not specified" state.

No node-identifier field is required. The write path locates an existing child by label at write time (Section 3.8.1), exactly as `cgi_location_address` does for `GridLocation_t`. This mirrors the existing structures, which carry `location` but no `location_id`.

3.7 Internal Lookup Table and String Mapping

The string table must be defined in `cgnslib.c` alongside `GridLocationName` and its siblings:

```
const char *DofStorageName[NofValidDofStorage] = {
    "Null",
    "UserDefined",
    "SharedDofs",
    "IndependentDofs"
};
```

A second table, `InterfaceContinuityName`, is defined the same way, holding `"Null"`, `"UserDefined"`, `"ContinuousInterface"`, and `"DiscontinuousInterface"`. Note the convention, verified against `GridLocationName` and `DataClassName` in `cgnslib.c`: the `Null` and `UserDefined` entries hold the bare strings `"Null"` and `"UserDefined"`, while the concrete values hold their full enumerator names.

A string-to-enum helper is added to `cgns_internals.c` for each enumeration, following `cgi_GridLocation` exactly. That includes its forward-compatibility behavior: an unrecognized string in a file written by a *newer* library version degrades to `UserDefined` with a warning rather than failing.

```
int cgi_DofStorage(char *Name, CGNS_ENUMT(DofStorage_t) *type)
{
    int i;
    for (i = 0; i < NofValidDofStorage; i++) {
        if (strcmp(Name, DofStorageName[i]) == 0) {
            (*type) = (CGNS_ENUMV(DofStorage_t))i;
            return CG_OK;
        }
    }
    if (cg->version > CGNSLibVersion) {
        (*type) = CGNS_ENUMV(DofStorageUserDefined);
        cgi_warning("Unrecognized DofStorage Type '%s' "
                    "replaced with 'UserDefined'", Name);
        return CG_OK;
    }
    cgi_error("Unrecognized DofStorage: %s", Name);
    return CG_ERROR;
}
```

3.8 MLL Implementation Requirements

Throughout this section, `cg` is the file-scope current-file pointer used internally by `cgnslib.c`, and `posit` is the file-scope current-position record maintained by `cg_goto`. Neither is declared in the listings, matching the surrounding library code.

3.8.1 Address Resolution Function (`cgns_internals.c`)

```c
/* A second, structurally identical helper,
 * cgi_interface_continuity_address(), resolves the
 * interface_continuity field and locates an existing
 * InterfaceContinuity_t child by label.  Only the type, the field
 * name, and the label string differ. */
CGNS_ENUMT(DofStorage_t) *cgi_dof_storage_address(
    int local_mode, int *ier)
{
    double *id, parent_id;
    CGNS_ENUMT(DofStorage_t) *dof_storage = 0;
    int nnod;

    if (posit == 0) {
        cgi_error("No current position set by cg_goto\n");
        (*ier) = CG_ERROR;
        return CG_OK;
    }

    /* Possible parents for DofStorage_t: FlowSolution_t,
     * DiscreteData_t, ZoneSubRegion_t, UserDefinedData_t */
    if (strcmp(posit->label, "FlowSolution_t") == 0)
        ADDRESS4SINGLE_ALLOC(cgns_sol, dof_storage)
    else if (strcmp(posit->label, "DiscreteData_t") == 0)
        ADDRESS4SINGLE_ALLOC(cgns_discrete, dof_storage)
    else if (strcmp(posit->label, "ZoneSubRegion_t") == 0)
        ADDRESS4SINGLE_ALLOC(cgns_subreg, dof_storage)
    else if (strcmp(posit->label, "UserDefinedData_t") == 0)
        ADDRESS4SINGLE_ALLOC(cgns_user_data, dof_storage)
    else {
        cgi_error("DofStorage_t node not supported under "
                  "'%s' type node", posit->label);
        (*ier) = CG_INCORRECT_PATH;
        return CG_OK;
    }

    /* On modify, delete any existing child, located by label */
    if (cg->mode == CG_MODE_MODIFY &&
        local_mode == CG_MODE_WRITE) {
        if (cgi_get_nodes(parent_id, "DofStorage_t", &nnod, &id))
            return CG_OK;
        if (nnod > 0) {
            if (cgi_delete_node(parent_id, id[0])) {
                (*ier) = CG_ERROR;
                return CG_OK;
            }
            CGNS_FREE(id);
        }
    }
    return dof_storage;
}
```

Locating the existing child by label is what makes a cached node identifier unnecessary, and is why the overwrite path cannot create a duplicate child.

3.8.2 Internal Tree Parser Update (`cgns_internals.c`)

When a CGNS file is opened, the MLL tree parser crawls the HDF5/CGIO tree and populates internal C structures. The parser must be updated to recognize child nodes with label `DofStorage_t` during the traversal of `FlowSolution_t` (`cgi_read_sol`), `DiscreteData_t` (`cgi_read_discrete`), `ZoneSubRegion_t` (`cgi_read_subregion`), and `UserDefinedData_t` (`cgi_read_user_data`).

The required behavior is:

1. During memory allocation of the parent structure, the `dof_storage` field is zero-initialized, which corresponds to `DofStorageNull`.
2. During the child-node traversal, if a child with label `"DofStorage_t"` is found, read its C1 string data and map it to the enum value via `cgi_DofStorage(string_data, &dof_storage)`.
3. If no such child node exists, the field retains its default `DofStorageNull` value.

This follows the existing pattern used for `GridLocation_t` parsing in `cgi_read_sol`. No node identifier is stored; see Section 3.8.1.

The same three steps apply to the `InterfaceContinuity_t` label and the `interface_continuity` field, in the same four parser functions. Because the two children are independent, a parent may carry either, both, or neither, and the traversal must not treat one as implying the other.

3.8.3 Read Function Implementation

```c
int cg_dof_storage_read(CGNS_ENUMT(DofStorage_t) *DofStorage)
{
    CGNS_ENUMT(DofStorage_t) *dof_storage;
    int ier = 0;

    CHECK_FILE_OPEN

    if (cgi_check_mode(cg->filename, cg->mode, CG_MODE_READ))
        return CG_ERROR;

    dof_storage = cgi_dof_storage_address(CG_MODE_READ, &ier);
    if (dof_storage == 0) return ier;

    (*DofStorage) = (*dof_storage);
    return CG_OK;
}
```

Because the tree parser already handles the absent-node case by leaving the field at `DofStorageNull`, the read function simply returns the cached value and `CG_OK`. This avoids any need to intercept `CG_NODE_NOT_FOUND` at read time: the absent-node case is handled once during file open, and all subsequent reads are served from the in-memory cache. Legacy code that iterates over solutions and queries the DOF storage layout will always receive `CG_OK` with `DofStorageNull` for files that predate this CPEX.

3.8.4 Write Function Implementation

```c
int cg_dof_storage_write(CGNS_ENUMT(DofStorage_t) DofStorage)
{
    CGNS_ENUMT(DofStorage_t) *dof_storage;
    int ier = 0;
    cgsize_t dim_vals;
    double posit_id, dummy_id;

    CHECK_FILE_OPEN

    if (cgi_check_mode(cg->filename, cg->mode, CG_MODE_WRITE))
        return CG_ERROR;

    if (INVALID_ENUM(DofStorage, NofValidDofStorage)) {
        cgi_error("Invalid DofStorage value: %d", (int)DofStorage);
        return CG_ERROR;
    }

    /* Null is not writable: absence of the node is what conveys
     * "unspecified".  This follows cg_gridlocation_write, whose
     * cgi_check_location rejects GridLocationNull.  Use
     * cg_delete_node("DofStorage") to remove an existing child. */
    if (DofStorage == CGNS_ENUMV(DofStorageNull)) {
        cgi_error("DofStorageNull cannot be written; omit the node "
                  "or use cg_delete_node");
        return CG_ERROR;
    }

    /* Resolves the parent, and on modify deletes any
     * existing DofStorage child (located by label). */
    dof_storage = cgi_dof_storage_address(CG_MODE_WRITE, &ier);
    if (dof_storage == 0) return ier;

    (*dof_storage) = DofStorage;

    if (cgi_posit_id(&posit_id)) return CG_ERROR;
    dim_vals = (cgsize_t)strlen(DofStorageName[DofStorage]);
    if (cgi_new_node(posit_id, "DofStorage", "DofStorage_t",
            &dummy_id, "C1", 1, &dim_vals,
            (void *)DofStorageName[DofStorage]))
        return CG_ERROR;

    return CG_OK;
}
```

Two details are worth noting. First, validation uses the library's own

`INVALID_ENUM(E,EMAX)` macro from `cgns_header.h`, as `cg_gridlocation_write` does, rather than an open-coded `switch`. Second, no node identifier is stored after `cgi_new_node` returns, because the address helper will find the node by label on any subsequent write.

3.9 Data Layout for Independent DOFs at Shared Mesh Vertices

A common and important case arises when a DG solver uses a mesh whose vertices are shared in the standard CGNS sense—enabling mesh walking, stream tracing, and connectivity queries via the standard `Elements_t` tables—but stores element-local, independent solution values at each element's copy of those vertices. At a shared vertex adjacent to k elements there are k distinct solution values, one per element; readers must not average or conflate them.

The correct CGNS encoding for this pattern is `GridLocation_t = Vertex` combined with `IndependentDofs` and a `PointList` that lists every element-vertex pair explicitly (repeated vertex indices are expected and meaningful). The `DofStorage_t` node is the machine-readable flag that tells readers these repeated entries represent independent DOFs, not duplicate data.

Without the `DofStorage_t` node, a reader cannot distinguish this layout from a malformed shared-DOF solution with duplicate entries, which is why the node is necessary. With it, a post-processor or coupling framework can correctly:

- walk the mesh using the shared-vertex connectivity for geometry operations (stream tracing, surface intersection, etc.);
- read solution values element-locally, preserving the jumps at shared interfaces.

3.9.1 Applicability and Restrictions

This encoding is subject to the following restrictions, which the SIDS amendment in Section 8 makes normative.

Unstructured zones only. The DOF ordering is defined by traversal of the zone's `Elements_t` sections, which structured zones do not have. The encoding is therefore valid only for `ZoneType_t = Unstructured`. Structured zones have no way to express independent DOFs at shared vertices, and this CPEX does not add one. A structured solver with element-local DOFs should either store cell-centered data (`GridLocation_t = CellCenter`, which is unambiguously independent) or convert the affected region to an unstructured zone. Note that CPEX 0045's `InterpolationOrders` child is likewise "recognized only for `Unstructured` zones," so the high-order route is not a structured fallback either.

The `PointList` must cover every element of the zone. The ordering rule below is a traversal of the zone's elements; a `PointList` covering only some elements has

no defined anchor into that traversal. A partial-coverage independent solution should use `GridLocation_t = InterpolationPoints` (CPEX 0045) or `IntegrationPoint` (CPEX 0047), both of which carry explicit per-element element-index lists.

Element type support. Fixed-size element types and `MIXED` and `NGON_n` sections are supported. `NFACE_n` is *not*: a polyhedral element's vertices are defined only indirectly, through its constituent faces, and CGNS defines no canonical vertex ordering for such an element, so "the element's vertices in order" is not well defined. Zones containing `NFACE_n` sections must use `GridLocation_t = InterpolationPoints` or `IntegrationPoint` for independent DOFs.

`Rind_t` must not be combined with this encoding. The `PointList` already enumerates every DOF explicitly, so there is no separate rind extent to declare, and the SIDS defines no ordering for rind entries within a point set. `Rind_t` remains fully usable with `IndependentDofs` when no point set is present; see Section 3.10.

Storage overhead. Because the `PointList` is, by construction, the concatenation of the zone's element connectivity, it duplicates data already present in the `Elements_t` sections at a cost of one `cgsize_t` per DOF. This is exactly the conflation of dof-relation and topo-relation that Miller [8] describes. The overhead should be weighed honestly against the two alternatives.

Writing the solution at `InterpolationPoints` instead does *not* destroy the shared-vertex connectivity: `GridCoordinates_t` and `Elements_t` are untouched, and mesh walking continues to work. On capability alone, therefore, the 0045 route also satisfies the requirement of Section 2.2, and this proposal should not claim otherwise. What it offers over that route is a far smaller implementation burden for readers—no `Family_t`-level `SolutionInterpolation_t` lookup, no basis evaluation, no control-point machinery—for the common case in which the DOF positions simply *are* the mesh vertices. For first-order DG that distinction is the difference between reading one extra scalar node and implementing an interpolation framework.

The alternative actually observed in production, by contrast, is strictly worse than either: duplicating the grid coordinates (Section 2.2) inflates `GridCoordinates_t` by the average vertex valence—far more than one `cgsize_t` per DOF—and destroys the connectivity outright. Measured against that, the `PointList` redundancy is a substantial improvement.

Where CPEX 0045 or CPEX 0047 has been adopted and the DOF positions are not the mesh vertices, `InterpolationPoints` or `IntegrationPoint` remains the preferred encoding: both size the array from per-element DOF counts with no redundant index list.

3.9.2 Other Grid Locations

For element-local DOFs at non-vertex positions (interior quadrature points, face quadrature points, etc.) under `FlowSolution_t`, use `GridLocation_t = InterpolationPoints` with CPEX 0045 metadata, or `GridLocation_t = IntegrationPoint` with CPEX 0047 metadata. Both are restricted by their respective proposals: CPEX 0045 states that `InterpolationPoints` "is valid only on `FlowSolution_t` nodes," and CPEX 0047 permits `IntegrationPoint` under `FlowSolution_t`, `ZoneSubRegion_t`, `BC_t`, and `BCDataSet_t`. Consequently `DiscreteData_t` has no standard encoding for element-local DOFs at non-vertex positions; `IndependentDofs` on a `DiscreteData_t` node is meaningful for `CellCenter` data, and for vertex data under the `PointList` encoding above, but not otherwise. Closing that gap is out of scope here.

`GridLocation_t = Vertex` with a `PointList` is specifically the correct choice when the DOF positions coincide with mesh vertices and the shared-vertex mesh connectivity is to be preserved.

`GridLocation_t = Vertex` with `IndependentDofs` and *no* `PointList` is not valid: the SIDS `DataSize` formula for `Vertex` yields `DataSize` = `VertexSize` (the shared-vertex count), which cannot represent the larger element-local DOF count. Similarly, `PointRange` is not suitable for this case because a contiguous range of vertex indices cannot express the element-local repetition of shared vertices; only `PointList` provides the necessary per-element DOF ordering. `GridLocation_t = CellCenter` with `IndependentDofs` remains valid since cell-centered data is inherently one value per element with no shared-entity ambiguity.

3.9.3 DOF Ordering Rule

SIDS Amendment — `PointList` semantics for independent DOFs. When `DofStorage_t` is `IndependentDofs` and a `PointList` is present, with `GridLocation_t = Vertex`, under any of `FlowSolution_t`, `DiscreteData_t`, `ZoneSubRegion_t`, or `UserDefinedData_t`, repeated vertex indices are permitted and represent element-local DOFs. Solution array position i corresponds to the i-th entry of the `PointList`, and the `PointList` shall be constructed as follows:

1. Traverse the zone's elements in ascending **element index** order. Element indices are the global, zone-wide indices defined by each `Elements_t` section's `ElementRange`; they are *not* the section storage order, and sections are not required to be stored in ascending `ElementRange` order.
2. Within each element, list the element's vertex indices in the canonical vertex order defined for its element type in SIDS §7.3.
3. For `MIXED` sections, the element-type code that precedes each element's vertex list in the connectivity array is not a DOF and is not listed.

4. For `NGON_n` sections, list the face's vertex indices in connectivity order; the per-element vertex count varies and the `PointList` length is the sum over all elements.

Repeated indices under `SharedDofs`, `DofStorageUserDefined`, or `DofStorageNull` remain undefined and invalid. This amendment does not affect `PointList` semantics under any other condition.

Because the `PointList` must equal the traversal above, a validator can check it exactly against the `Elements_t` connectivity; see Section 11.

See Example 7.5 for the complete C encoding of this pattern.

3.10 Interaction with `Rind_t` (Ghost Layers)

`Rind_t` remains usable with both `DofStorage_t` values, with one exception: as stated in Section 3.9.1, it must not be combined with the `Vertex` + `IndependentDofs` + `PointList` encoding of Section 3.9, because the SIDS defines no position for rind entries within a point set. The rest of this section therefore concerns solution nodes carrying no point set.

Where no point set is present, `DofStorage_t` applies uniformly to the entire solution field, including any rind (ghost) entries declared by `Rind_t`. An `IndependentDofs` label means that all data entries—interior and rind alike—are element-local: each rind entry belongs to one element and must not be interpreted as a shared vertex value. DG and finite-volume solvers that use ghost layers for MPI halo exchange may therefore store element-local rind DOFs in the standard way at the cell- and face-centred grid locations, where the rind extent counts entities and each entity carries one DOF; the `IndependentDofs` label covers them. Rind entries of a `SharedDofs` solution retain their usual shared-DOF semantics. No special treatment of the rind is required beyond what `Rind_t` already specifies; the DOF storage label is consistent across the full data array.

Two grid locations are deliberately absent from that statement. `Vertex` under `IndependentDofs` is unavailable in both directions: without a point set the combination is already invalid on sizing grounds (Section 3.9), and with one it excludes `Rind_t` by the restriction above. A solver needing both explicit element-local vertex DOFs and a declared ghost extent has no encoding in this CPEX, and should exchange its halo outside the file rather than declare it.

`InterpolationPoints` and `IntegrationPoint` are absent for a different reason. At both, the solution array is a concatenation of per-element DOF blocks, so the unit of the rind extent—elements or DOFs—is not fixed by CPEX 0045, by CPEX 0047, or by the SIDS. This proposal does not resolve that: the ambiguity belongs to the host encoding rather than to the storage label, and settling it is properly the business of

whichever of those two proposals owns the grid location. Writers should be aware that `Rind_t` is not portable at those locations until one of them does so.

4 Interaction with CPEX 0045 (Higher-Order Elements)

CPEX 0045 [4] introduced `GridLocation_t = InterpolationPoints`, the `InterpolationOrders` and `CharacteristicLength` children of `FlowSolution_t`, and the `SolutionInterpolation_t` and `ElementInterpolation_t` children of `Family_t`. The present CPEX complements rather than modifies that infrastructure:

- CPEX 0045 answers: "What polynomial basis and order describes the solution within each element?"
- This CPEX answers: "Are the resulting DOFs shared across element interfaces or element-local?"

Both pieces of metadata are needed for a complete description of a higher-order solution field. They are orthogonal:

	`SharedDofs`	`IndependentDofs`
Lagrange $p = 1$	CG FEM[‡]	DG $p = 1$
Lagrange $p = 3$	CG spectral[‡]	DG spectral
Hierarchical $p = 2$	n/a[§]	DG (modal)
$p = 0$ (constant)	n/a[†]	FV / DG $p = 0$

[†] A $p = 0$ basis has no interface DOFs to share: its single per-element DOF has no geometric support on the element boundary. `SharedDofs` is therefore not merely unusual at $p = 0$ but meaningless, and `cgnscheck` should warn on it (Section 11).

[‡] Realizable only at `GridLocation_t = Vertex` on the predefined element types, where the `Elements_t` connectivity supplies the DOF identification. It is *not* realizable under `InterpolationPoints`, where `SharedDofs` is invalid on a volume block (Section 4.2). Since predefined element types reach only fourth order, shared storage is expressible only for $p \leq 4$.

[§] Hierarchical bases admitting a vertex/edge/face/interior mode decomposition support continuous assembly mathematically, but CPEX 0045's monomial encodings (`ParametricMonomialsPascal`, `CartesianMonomialsPascal`) store modal coefficients that have no geometric support on any sub-entity, and 0045 fixes no sub-entity ordering for them. There is consequently no entity to share and no DOF correspondence to establish, so `SharedDofs` cannot be acted upon. The same holds for `ParametricLagrange` with user-supplied control points: 0045 guarantees a principal-vertices-first ordering only for `ElementInterpolation_t`, not for `SolutionInterpolation_t`, and defines no orientation-aware correspondence between the points of two adjacent elements.

4.1 Recommended Practice

For higher-order solutions using `GridLocation_t = InterpolationPoints`:

1. Use CPEX 0045 to define the element type, interpolation basis, and polynomial order via `SolutionInterpolation_t` on the `Family_t` and `InterpolationOrders` on the `FlowSolution_t`.
2. Use this CPEX to set `DofStorage_t` on the `FlowSolution_t` node.
3. Also set `InterfaceContinuity_t` on that node. This step matters most at precisely this grid location: the array is a concatenation of per-element DOF blocks, so every interface DOF is duplicated and the attribute is never vacuous here (Section 3.3.2). A continuous spectral-element field and a DG field remain indistinguishable without it no matter how completely steps 1 and 2 are carried out—same basis, same order, same array length (Example 7.7). A writer that does not know shall omit it rather than guess.

Example 7.2 shows all three steps.

For lower-order solutions using `GridLocation_t = Vertex` or `CellCenter`, this CPEX can be used independently of CPEX 0045; see Example 7.1. See Section 3.9 for the prescribed data layout when independent DOFs coincide with shared mesh vertices. There, `InterfaceContinuity_t` is worth setting under the `Vertex` + `IndependentDofs` + `PointList` encoding, which duplicates DOFs in the same way; at `CellCenter`, or at `Vertex` with genuinely shared storage, it should be omitted (Section 3.3.2).

4.2 `DofStorage_t` Under `InterpolationPoints`

Under `GridLocation_t = InterpolationPoints`, CPEX 0045 defines the field-array length as

$$L = \sum_{e \in \mathcal{E}} N_{\mathrm{DOFs}}(e),$$

where $\mathcal{E}$ is the set of elements covered by the block—all zone elements when no `PointRange`/`PointList` is present, or the listed elements otherwise—with each element's DOFs stored contiguously. Because the array is a concatenation of per-element DOF blocks, the following rules apply:

- `IndependentDofs` is the layout's natural reading and requires no additional metadata. It is the expected value for volume DG and spectral-element blocks.
- `SharedDofs` is permitted *only* when the block's element set consists of interface (face or edge) elements, listed explicitly by a `PointList` or `PointRange` over the element indices of the corresponding face or edge `Elements_t` section(s). The shared reading is then: each listed interface element's DOF block is shared between the volume elements adjacent to it. This is the HDG trace case.

- `SharedDofs` on a block whose element set is the zone's volume elements—in particular, any `InterpolationPoints` block with *no* point set—is **invalid**. Under that sizing each interface DOF would appear once per adjacent volume element, which contradicts the definition of `SharedDofs`. `cgnscheck` should report this as an error (Section 11).

5 Interaction with CPEX 0047 (Integration Points)

CPEX 0047 [5] introduces `GridLocation_t = IntegrationPoint`, quadrature-rule definitions, and an `ItgPointsStartOffset` array giving the starting position of each element's integration-point data within the solution array. It permits that grid location under `FlowSolution_t`, `ZoneSubRegion_t`, `BC_t`, and `BCDataSet_t`.

The two proposals are complementary. `ItgPointsStartOffset` is precisely the per-element offset array whose absence motivates Section 3.9, and where it applies it is the better mechanism—this CPEX recommends it over the redundant `PointList` encoding. Because integration points are interior to elements and the offset array assigns each element a disjoint slice, `IndependentDofs` is the only valid value under that grid location; recording it aids uniform reader dispatch but adds nothing derivable, and `SharedDofs` there should be a `cgnscheck` error. Neither proposal subsumes the other: 0047 says nothing about vertex-, edge-, or face-located DOFs, which is the only place a storage discriminator is load-bearing, and does not cover `DiscreteData_t`. Conversely, 0047 extends `BC_t` and `BCDataSet_t` where this CPEX deliberately does not (Design Decision 1), and its `IntegrationPoint` location already covers element-local boundary data, including the weakly-imposed boundary conditions of continuous methods.

One editorial concern spans the two proposals: CPEX 0045's `InterpolationPoints` and CPEX 0047's `IntegrationPoint` are similar names for different `GridLocation_t` values. That is a matter for the editors of those proposals, but readers of this one should note the distinction.

6 Backward Compatibility

- **Existing files:** Fully compatible. Both new nodes are optional; the absence of `DofStorage` is equivalent to `DofStorageNull` and the absence of `InterfaceContinuity` to `InterfaceContinuityNull`. No existing file is invalidated.
- **Existing readers:** Unaffected. Per the CGNS convention, readers silently ignore unknown child nodes. An older library reading a file with either new node will simply skip it. `cgnscheck` validates only the child labels it knows about

and does not enumerate or warn on unrecognized ones, so existing validation runs are unaffected as well.

- **Existing writers:** Unaffected. Codes that do not call the new API functions produce files without `DofStorage` nodes, identical to the current behavior.
- **Newer-to-older direction:** A file written by a future library with an additional `DofStorage_t` value will be read by a CPEX-0050-era library as `DofStorageUserDefined` with a warning, not an error, because `cgi_DofStorage` follows `cgi_GridLocation`'s forward-compatibility rule.
- **ABI:** The `cgns_sol`, `cgns_discrete`, `cgns_subreg`, and `cgns_user_data` structures each grow by one `int`-sized enum field. This is an ABI change requiring recompilation of code that directly accesses the internal structures. The public API is purely additive (new functions, new enum), so no existing function signatures change.

6.1 Parallel I/O

No new `cgp_*` functions are required, but the constraint under which the serial functions may be used in a parallel program must be stated explicitly.

The parallel MLL (`pcgnslib.c`) contains no handling of `GridLocation`, `DataClass`, or any other scalar metadata child; such nodes are written by the serial MLL even in parallel programs. `DofStorage` follows suit. Because writing it creates—and, on modify, deletes—an HDF5 group, and HDF5 structural modification under MPI-IO is a collective operation:

- `cg_dof_storage_write` **shall** be called by all ranks in the communicator, with identical arguments and from an identical current position.
- The same applies to the `cg_goto` call that establishes the position.
- `cg_dof_storage_read` may be called independently once the file is open, since it is served entirely from the in-memory structure populated at open time.

This is the same rule that already governs `cg_sol_write`, `cg_gridlocation_write`, and every other metadata-creating serial call used alongside `cgp_*` data transfers; it is stated here because the proposal adds a new call subject to it.

6.2 Interpreting Legacy Data (`DofStorageNull`)

`DofStorageNull` means *unspecified*, not "shared." A reader must not silently substitute a default. The following inference rules are recommended—non-normatively—for readers that must nevertheless act on files predating this CPEX:

Any element- or face-centred location That is, `CellCenter`, `FaceCenter`, `EdgeCenter`, or `[IJK]FaceCenter`. Treat as `IndependentDofs`: there is exactly one value per entity and no shared-entity ambiguity.

`Vertex` with `DataSize` = `VertexSize` Treat as `SharedDofs`. Before this CPEX, the SIDS `DataSize` formula admitted no other layout at this size.

`Vertex` with a `PointList` Unspecified. A legacy file gives no way to tell an independent-DOF layout from a genuine vertex subset. Readers should surface the ambiguity rather than guess.

`InterpolationPoints` (CPEX 0045) Treat as `IndependentDofs`, per Section 4.2.

`IntegrationPoint` (CPEX 0047) Treat as `IndependentDofs`.

Note that these rules make no "assume shared for backward compatibility" blanket assumption: since most CFD files are cell-centered finite volume, such an assumption would be wrong for the majority of existing data. Example 7.3 shows a reader applying this dispatch.

7 Examples

All examples use the position-based API: `cg_goto` to the target node, then `cg_dof_storage_write`. Error checking is omitted for brevity; production code must check every return value.

Two of the examples are load-bearing. Example 7.5 is the motivating case of Section 2.2: the encoding it shows has no alternative today, and it is the reason for the storage attribute. Example 7.7 is its counterpart for the continuity attribute—two files that CGNS cannot otherwise tell apart. The remainder are shorter and serve to fix the calling conventions: the shared/independent pair, the reader-side dispatch, the Fortran bindings, and the remaining permitted parents. Where those label a layout that the grid location and sizing rules already determine, it is noted in place rather than presented as a case requiring the node.

Examples using `GridLocation_t = InterpolationPoints` presuppose CPEX 0045, whose enumeration value does not exist in the current library; they are written against the joint CGNS 5.0 target of Section 12.

7.1 Example 1: Writing a Shared-DOF (CG) Solution

```c
/* Create a vertex-located solution node */
int S, F;
cg_sol_write(fn, B, Z, "FlowSolution", CGNS_ENUMV(Vertex), &S);

/* Mark DOFs as shared (CG FEM) */
cg_goto(fn, B, "Zone_t", Z, "FlowSolution_t", S, "end");
cg_dof_storage_write(CGNS_ENUMV(SharedDofs));

/* Write solution data (one value per mesh vertex, shared) */
cg_field_write(fn, B, Z, S, CGNS_ENUMV(RealDouble),
    "Pressure", pressure, &F);
```

7.2 Example 2: Writing an Independent-DOF (DG) Solution

```c
/* Create an interpolation-point-located solution node.
 * Per CPEX 0045, a block covering the whole zone at uniform
 * order needs no point set; DataSize is the sum of the
 * per-element DOF counts over all zone elements. */
int S, F;
cg_sol_write(fn, B, Z, "FlowSolution_DG",
    CGNS_ENUMV(InterpolationPoints), &S);

/* Set polynomial order (CPEX 0045):
 * spatialOrder = 3, temporalOrder = 0 */
cg_sol_interpolation_order_write(fn, B, Z, S, 3, 0);

/* Mark DOFs as element-local */
cg_goto(fn, B, "Zone_t", Z, "FlowSolution_t", S, "end");
cg_dof_storage_write(CGNS_ENUMV(IndependentDofs));

/* Duplicated interface DOFs carry meaningful jumps: this is DG,
 * not a continuous field stored per element (cf. Example 7). */
cg_interface_continuity_write(CGNS_ENUMV(DiscontinuousInterface));

/* Write element-local DOFs */
cg_field_write(fn, B, Z, S, CGNS_ENUMV(RealDouble),
    "Pressure", pressure_dg, &F);
```

7.3 Example 3: Reading and Dispatching on DOF Storage Layout

```c
CGNS_ENUMT(DofStorage_t) layout;

cg_goto(fn, B, "Zone_t", Z, "FlowSolution_t", S, "end");
cg_dof_storage_read(&layout);

switch (layout) {
case CGNS_ENUMV(SharedDofs):
    /* Shared DOFs: interpolate using global mesh connectivity */
    visualize_shared(fn, B, Z, S);
    break;

case CGNS_ENUMV(IndependentDofs):
    /* Element-local DOFs: render per-element, preserve jumps */
    visualize_independent(fn, B, Z, S);
```

```
    break;

case CGNS_ENUMV(DofStorageUserDefined): {
    /* Writer must have left a DofStorageDescription
     * descriptor identifying the convention. */
    char dname[CGIO_MAX_NAME_LENGTH+1];
    char *text = NULL;
    int nd, n;
    cg_ndescriptors(&nd);
    for (n = 1; n <= nd; n++) {
        cg_descriptor_read(n, dname, &text);
        if (text == NULL) continue;
        if (strcmp(dname, "DofStorageDescription") == 0)
            visualize_user_defined(fn, B, Z, S, text);
        cg_free(text);
        text = NULL;
    }
    break;
}

default:
    /* DofStorageNull: legacy file.  Apply the inference
     * rules of Section 6.2 -- do not assume SharedDofs. */
    visualize_legacy(fn, B, Z, S);
    break;
}
```

7.4 Example 4: Fortran Usage

```
integer :: S, ier, layout

! Write a shared-DOF solution
call cg_sol_write_f(fn, B, Z, 'FlowSolution', Vertex, S, ier)
call cg_goto_f(fn, B, ier, 'Zone_t', Z, 'FlowSolution_t', S, 'end')
call cg_dof_storage_write_f(SharedDofs, ier)

! Read the DOF storage layout back
call cg_goto_f(fn, B, ier, 'Zone_t', Z, 'FlowSolution_t', S, 'end')
call cg_dof_storage_read_f(layout, ier)
if (layout == IndependentDofs) then
    ! Element-local interpolation
end if
```

7.5 Example 5: Independent DOFs on a Shared-Vertex Mesh

This example addresses the case where the mesh uses shared vertices (enabling stream tracing and mesh walking via standard `Elements_t` connectivity), but each element stores its own independent solution values at those vertices. The `DofStorage_t` node is the flag that tells readers the repeated vertex indices in the `PointList` are independent DOFs, not duplicates. The zone must be unstructured, the `PointList` must cover every element, and the traversal order is fixed by Section 3.9.3.

```c
/*
 * Given, for a single-section unstructured zone:
 *   cgsize_t nVerts;          shared vertices in the zone
 *   cgsize_t nElems;          elements in the zone
 *   int      nVertsPerElem;   vertices per element (fixed type)
 *   cgsize_t *elem_conn;      connectivity, nElems*nVertsPerElem,
 *                             1-based, ascending element index
 *   double   *pressure_dg;    nElems*nVertsPerElem values
 *
 * Total DOFs = nElems * nVertsPerElem  (not nVerts).
 */
cgsize_t nDofs = nElems * (cgsize_t)nVertsPerElem;
cgsize_t *point_list = (cgsize_t *)malloc(nDofs * sizeof(cgsize_t));
if (point_list == NULL) return 1;

/* The PointList is the connectivity traversed in ascending
 * element index order, each element's vertices in canonical
 * order.  For this single fixed-size section that is exactly
 * the connectivity array; a multi-section, MIXED, or NGON_n
 * zone requires the traversal of Section 3.8.3. */
{
    cgsize_t i;
    for (i = 0; i < nDofs; i++)
        point_list[i] = elem_conn[i];
}

/* Create the solution node with a PointList.  Vertex location
 * preserves shared-mesh connectivity for geometry operations,
 * while DataSize becomes nDofs rather than nVerts. */
int S, F;
cg_sol_ptset_write(fn, B, Z, "FlowSolution_DG",
    CGNS_ENUMV(Vertex), CGNS_ENUMV(PointList),
    nDofs, point_list, &S);

/* Mark as independent: values at repeated vertex indices are
 * independent across elements, ordered by ascending element
 * index then canonical vertex order. */
cg_goto(fn, B, "Zone_t", Z, "FlowSolution_t", S, "end");
cg_dof_storage_write(CGNS_ENUMV(IndependentDofs));

/* Write element-local solution data */
cg_field_write(fn, B, Z, S, CGNS_ENUMV(RealDouble),
    "Pressure", pressure_dg, &F);

free(point_list);
```

A reader distinguishes this from a shared-DOF solution as follows:

```c
CGNS_ENUMT(DofStorage_t) layout;

cg_goto(fn, B, "Zone_t", Z, "FlowSolution_t", S, "end");
cg_dof_storage_read(&layout);

if (layout == CGNS_ENUMV(IndependentDofs)) {
    /* PointList entries are element-local DOF indices.
     * Values at the same vertex index from different elements
     * are independent -- do not average. */
}
```

7.6 Example 6: Other Permitted Parent Nodes

Usage on `DiscreteData_t` and `ZoneSubRegion_t` is identical to `FlowSolution_t`; only the `cg_goto` path differs.

```
/* DiscreteData_t: a cell-centered DG error indicator */
int D;
cg_discrete_write(fn, B, Z, "ErrorIndicator", &D);
cg_goto(fn, B, "Zone_t", Z, "DiscreteData_t", D, "end");
cg_gridlocation_write(CGNS_ENUMV(CellCenter));
cg_dof_storage_write(CGNS_ENUMV(IndependentDofs));
cg_array_write("ElementError", CGNS_ENUMV(RealDouble),
    1, &nElems, error_data);

/* ZoneSubRegion_t: per-face boundary fluxes.  The sub-region
 * dimension is one less than the zone's cell dimension, so query
 * the base rather than hard-coding it. */
char basename[33];   /* CGNS name limit is 32 chars */
int cellDim, physDim, SR;
cg_base_read(fn, B, basename, &cellDim, &physDim);
cg_subreg_bcname_write(fn, B, Z, "WallFluxes",
    cellDim - 1, "Wall_BC", &SR);
cg_goto(fn, B, "Zone_t", Z, "ZoneSubRegion_t", SR, "end");
cg_dof_storage_write(CGNS_ENUMV(IndependentDofs));
cg_array_write("HeatFlux", CGNS_ENUMV(RealDouble),
    1, &nBCFaces, heat_flux);
```

A single `cg_goto` serves every subsequent call: neither `cg_gridlocation_write`, `cg_dof_storage_write`, nor `cg_array_write` moves the current position. Both layouts above are already determined by the grid location and the sizing rules (Section 2.4); the node is written for uniformity, so that a reader can dispatch on one query across every solution node in a file.

7.7 Example 7: A Continuous Field Stored per Element

A continuous spectral-element solver writing at `InterpolationPoints` duplicates its interface DOFs, so the storage is `IndependentDofs`; but the duplicated values agree, and a post-processor may weld them. Only the second node communicates that.

```
int S;
cg_sol_write(fn, B, Z, "FlowSolution",
    CGNS_ENUMV(InterpolationPoints), &S);

cg_goto(fn, B, "Zone_t", Z, "FlowSolution_t", S, "end");

/* Storage duplicates DOFs at element interfaces ... */
cg_dof_storage_write(CGNS_ENUMV(IndependentDofs));

/* ... but the duplicates agree: the field is C0 across interfaces. */
cg_interface_continuity_write(CGNS_ENUMV(ContinuousInterface));
```

Replace the second call with `DiscontinuousInterface` and the file describes a DG solution instead. Every other item of metadata is identical, which is precisely why the attribute is needed. A reader dispatches on the pair:

```
CGNS_ENUMT(DofStorage_t) storage;
CGNS_ENUMT(InterfaceContinuity_t) cont;
cg_dof_storage_read(&storage);
cg_interface_continuity_read(&cont);

if (storage == CGNS_ENUMV(IndependentDofs) &&
    cont    == CGNS_ENUMV(ContinuousInterface)) {
    /* Safe to weld duplicated DOFs into a continuous field. */
} else {
    /* Preserve every value distinctly: either the jumps are
     * meaningful, or continuity was not asserted and must not
     * be assumed. */
}
```

8 SIDS Impact

1. **New enumeration type:** `DofStorage_t` is added to the SIDS Building-Block Structure Definitions (SIDS Section 4). That section is ordered alphabetically with one pre-existing exception—4.1 `DataClass_t`, 4.2 `Descriptor_t`, 4.3 `DimensionalUnits_t`, 4.4 `DimensionalExponents_t` (`Units` precedes `Exponents` in the published text), 4.5 `GridLocation_t`, 4.6 `IndexArray_t`, 4.7 `IndexRange_t`, 4.8 `Rind_t`. Alphabetically `DofStorage_t` falls after the two `Dimensional` entries and before `GridLocation_t`, so it is inserted as the new **Section 4.5** and the current 4.5–4.8 are renumbered 4.6–4.9. Correcting the existing inversion is out of scope here. All cross-references to those four sections require updating.

2. **Second new enumeration type:** `InterfaceContinuity_t` is likewise added to SIDS Section 4. Alphabetically it follows `IndexRange_t`, so with `DofStorage_t` inserted as 4.5 the two insertions together give `InterfaceContinuity_t` Section **4.9**, and the original 4.5–4.8 become 4.6–4.8 and 4.10. The renumbering should be applied once, for both additions together.

3. **`FlowSolution_t` amendment:** The `FlowSolution_t` node definition (SIDS Section 7.7) is amended to include optional `DofStorage_t` and `InterfaceContinuity_t` child nodes.

4. **`DiscreteData_t` amendment:** The `DiscreteData_t` node definition (SIDS Section 12.4) is similarly amended.

5. **`ZoneSubRegion_t` amendment:** The `ZoneSubRegion_t` node definition (SIDS Section 7.9) is similarly amended.

6. **`UserDefinedData_t` amendment:** The `UserDefinedData_t` node definition (SIDS Section 12.10, but see the renumbering note below) is similarly amended.
7. **`PointList` semantics amendment:** The description of `PointList` under `FlowSolution_t` in SIDS §7.7 is amended to permit repeated vertex indices when `IndependentDofs` and `GridLocation_t = Vertex`, in unstructured zones only. Repeated indices in that context represent element-local DOFs and are not malformed duplicates. The DOF-to-element mapping, the element traversal order, the treatment of `MIXED` type codes, the exclusion of `NFACE_n`, the whole-zone coverage requirement, and the prohibition on combining `Rind_t` with this encoding are specified in Sections 3.9.1 and 3.9.3 and shall be reproduced in the SIDS text. This amendment does not affect `PointList` semantics under any other condition.
8. **`InterpolationPoints` interaction:** If CPEX 0045 is adopted, its `GridLocation_t = InterpolationPoints` description is amended with the `DofStorage_t` compatibility rules of Section 4.2.
9. No existing SIDS node types, enumerations, or conventions are modified or removed. The only change to existing text is the Section 4 renumbering in item 1, which is editorial.

Renumbering coordinated with CPEX 0045. CPEX 0045 also amends SIDS Chapter 12: it inserts `ElementInterpolation_t` as Section 12.10 and `SolutionInterpolation_t` as Section 12.11, which pushes the present `UserDefinedData_t` from 12.10 to 12.12 and `Gravity_t` from 12.11 to 12.13. Because the two proposals target the same release (Section 12), the citations above are given against the *current* SIDS and must be applied after 0045's insertions: `UserDefinedData_t` is amended at whatever number it then carries. Chapter 4 and Chapter 7 are unaffected by 0045's renumbering—0045 generalises Sections 7.7 and 7.8 in place without inserting new ones—so the `FlowSolution_t`, `ZoneSubRegion_t`, `DiscreteData_t`, and Chapter 4 citations above are stable.

9 File Mapping Impact

The `DofStorage` child node is stored identically to `GridLocation`: as a character node containing the string name of the enumeration value. The node attributes are given below in the format prescribed by the CGNS File Mapping Manual's *Detailed CGNS Node Descriptions* [2]. `DofStorage_t` belongs in the "Location and Position Group" of basic nodes, immediately following `GridLocation_t`.

Node Attributes: `DofStorage_t`	
Name:	`DofStorage`
Label:	`DofStorage_t`
DataType:	`C1`
Dimension:	1
Dimension Values:	Length of the string value
Data:	`Null`, `UserDefined`, `SharedDofs`, or `IndependentDofs`
Children:	None
Cardinality:	0,1

Node Attributes: `InterfaceContinuity_t`	
Name:	`InterfaceContinuity`
Label:	`InterfaceContinuity_t`
DataType:	`C1`
Dimension:	1
Dimension Values:	Length of the string value
Data:	`Null`, `UserDefined`, `ContinuousInterface`, or `DiscontinuousInterface`
Children:	None
Cardinality:	0,1

Both are attribute-for-attribute the same as the `GridLocation_t` node description, apart from the name, label, and permitted data values. As for every CGNS enumeration, the stored strings are the enumerator names themselves; see Section 3.2.3 on why the values of this enumeration carry a prefix, following `BCType_t`.

9.1 Affected File Mapping Figures

The following File Mapping figures gain a `DofStorage` child box and an `InterfaceContinuity` child box, both drawn in the same style as the existing `GridLocation` child:

- `FlowSolution_t`
- `DiscreteData_t`
- `ZoneSubRegion_t`
- `UserDefinedData_t`

9.2 Writing Conventions

The node is written only when the DOF storage layout is known; `DofStorageNull` is represented by the *absence* of the node and is rejected on write.

Two existing mechanisms make this the conventional choice rather than a new one. `cg_gridlocation_write` likewise refuses its `Null` value where the location is meaningful: `cgi_check_location` rejects `GridLocationNull` under `FlowSolution_t`, `DiscreteData_t` and `BC_t`. And removal of an optional child is already a solved problem— `cg_delete_node("DofStorage")` at the same position in `CG_MODE_MODIFY` deletes it, and neither new node appears among the nodes `cg_delete_node` refuses to remove. No new convention is therefore proposed: "unspecified" is expressed by not writing the node, and un-saying a value is expressed by deleting it.

The nodes differ from `GridLocation` in what absence *means*. An absent `GridLocation` implies the concrete value `Vertex`; an absent `DofStorage` implies only "unspecified" (Section 6.2). This is why the default cannot simply be written out: there is no concrete default to write.

It follows that the string `"Null"` is never produced by this API. It remains in the `DofStorageName` table for positional correspondence with the enumeration, exactly as `GridLocationName[0]` does, and readers *shall* accept it—a hand-built file or a third-party writer may emit it—treating it as equivalent to an absent node. This is why Section 11 lists `"Null"` among the permitted strings and the File Mapping description of Section 9 lists `Null` among the permitted data values.

`InterfaceContinuity` follows the identical convention in every respect. In particular, absence must not be read as `ContinuousInterface`.

When `DofStorageUserDefined` is written, the accompanying description node is an ordinary `Descriptor_t` and follows the existing `Descriptor_t` file mapping unchanged, with name `DofStorageDescription`, label `Descriptor_t`, data type `C1`, dimension 1, and cardinality 0,1 within the parent node.

10 Design Decisions

Several design questions were evaluated during the development of this proposal. Each is addressed below with rationale grounded in existing CGNS conventions and SIDS precedent.

1. **Scope: `BCData_t` is excluded.** `DofStorage_t` is *not* added to `BCData_t`. Boundary condition data arrays in `BCData_t` are sized by the associated `PointList`/`PointRange` and `GridLocation_t`, which already determine the data layout unambiguously. Adding `DofStorage_t` to `BCData_t` would therefore be redundant, and would introduce a potential inconsistency if the BC layout differed from the solution it constrains.

   No inference in the other direction is intended or valid: the layout of boundary data does *not* follow from the DOF storage layout of the parent `FlowSolution_t`,

and weak imposition of boundary conditions is not a DG-specific technique. Nitsche and penalty methods in continuous FEM and in isogeometric analysis routinely pair a `SharedDofs` interior solution with element-local boundary data at face quadrature points. Such data is already expressible: CPEX 0047 permits `GridLocation_t = IntegrationPoint` under `BC_t` and `BCDataSet_t`, with `ItgPointsStartOffset` fixing the per-face layout, so no additional metadata is required for that case either. The `ZoneSubRegion_t` extension introduced here partially addresses such cases: interface fluxes or surface-local fields that require a DOF storage label can be stored in a `ZoneSubRegion_t` referencing the relevant boundary condition.

2. **Per-solution-node granularity (not per-field).** The DOF storage layout applies at the `FlowSolution_t` level, meaning all `DataArray_t` children share the same layout. This follows the precedent set by `GridLocation_t`, which is likewise a per-solution-node property: all fields within a `FlowSolution_t` node share the same grid location. Per-`DataArray_t` granularity was rejected because it would add significant API and metadata complexity with no demonstrated need. Solvers that genuinely mix layouts within a single zone (e.g., HDG methods with shared traces and independent volumes) should use separate `FlowSolution_t` nodes, exactly as they would use separate nodes for fields at different `GridLocation_t` values.

3. **Semantically inconsistent combinations are permitted but not enforced.** The MLL does *not* enforce consistency between `DofStorage_t` and `GridLocation_t`. Two combinations are defective, in different degrees:

   - A `CellCenter` solution is inherently element-local, so `SharedDofs` is *meaningless* there—the single per-element DOF has no geometric support on the element boundary. `cgnscheck` should warn.

   - `SharedDofs` on an `InterpolationPoints` block whose element set is the zone's volume elements is *invalid*, not merely odd: the sizing contradicts the definition of `SharedDofs`, and, independently, CPEX 0045 supplies no DOF-to-sub-entity map with which sharing could be realized (Section 4.2). `cgnscheck` should report an error.

   In neither case shall the library reject the combination on write. This follows the established CGNS convention of being permissive in what is accepted: the standard documents correct usage; validation is delegated to application-level tools such as `cgnscheck` (Section 11).

4. **Position-based API rather than per-parent functions.** The address-helper pattern used by every other optional scalar metadata child already solves parent dispatch, modify-mode overwrite, and absent-node reads, so reimplementing those three behaviors once per parent would add code without adding capability.

Two C functions and two Fortran bindings cover all four parents instead of six and six, and admitting a fifth parent later is one line in the address helper rather than four new public entry points.

5. **A new node type rather than a new `GridLocation_t` value.** Adding a value such as `ElementVertex` would conflate two orthogonal axes: `GridLocation_t` answers *where*, DOF storage answers *how many copies*. Because each existing spatial location can in principle carry either layout, folding the second axis into the first would eventually double the enumeration rather than add one value. It would also be worse for compatibility—every CGNS reader switches on `GridLocation_t`, whereas an unknown child node is silently ignored by both the MLL and `cgnscheck` (Section 6). CGNS already separates orthogonal per-solution properties into sibling child nodes (`GridLocation`, `Rind`, `DataClass`, `DimensionalUnits`). Note that CPEX 0045 and CPEX 0047 each *do* add a `GridLocation_t` value; those describe genuinely new locations, which is the correct use of that enumeration.

6. **Two separate attributes rather than one combined enumeration.** A single enumeration could in principle enumerate the meaningful pairs—`SharedContinuous`, `IndependentContinuous`, `IndependentDiscontinuous`. Rejected for three reasons. The two properties answer different questions, are supplied by different parts of a solver, and are known with different confidence: a writer almost always knows its storage layout, while continuity is an assertion about the discretization that it may not be in a position to make. Combining them would force a writer that knows only the layout to either overstate its knowledge or say nothing at all. A combined enumeration would also grow multiplicatively if either property later gains a value, and it would make the common case—recording storage alone, as every example before Example 7.7 does—depend on a decision about continuity that is irrelevant to it. Orthogonal properties are recorded in orthogonal nodes, which is also how CGNS treats `GridLocation_t` and `DataClass_t`.

10.1 Rejected Alternatives

A boolean `Discontinuous` flag. Rejected. A boolean cannot express "unspecified," which is exactly the state every pre-existing file is in, so a boolean would force writers to assert a layout they may not know. It also leaves no room for a `UserDefined` escape hatch, and CGNS has no precedent for boolean-valued metadata nodes.

A third `MixedStorage` value. Rejected. It would conflate two distinct layouts in one node, making post-processing harder rather than easier. Mixed-layout methods store distinguishable data and should store it in distinguishable nodes, exactly as they would for fields at different `GridLocation_t` values.

Encoding the layout in `UserDefinedData_t`. Rejected as the primary mechanism. It requires every reader to agree on a name and string convention by side-channel, which is precisely the interoperability failure this CPEX exists to fix. A standard enumeration with a standard label is machine-checkable; `UserDefinedData_t` is not. Note that this CPEX nevertheless *permits* `DofStorage_t` as a child of `UserDefinedData_t`, so applications storing fields there can label them properly.

Deriving the layout from array sizes. Not rejected, and worth stating accurately: for every encoding legal before this CPEX the layout *is* determined by `GridLocation_t` and the SIDS `DataSize` formula, irrespective of element mix (Section 6.2). A discriminator becomes necessary only once a second sizing is admitted at the same grid location, which is what the `Vertex` plus `PointList` encoding of Section 3.9 does. Outside that case the node is confirmatory, and is recommended for uniformity and self-description rather than required to resolve an ambiguity.

Placing the node on `Zone_t` or `Base_t`. Rejected. A single zone routinely carries solutions with different layouts—an HDG solver's volume and trace blocks, for instance—so a zone-level setting could not express them.

11 Recommended `cgnscheck` Validation Rules

Per Design Decision 3, the MLL is permissive and validation belongs in `cgnscheck`. The following rules are recommended. Note that the on-disk representation is a `C1` string, so there is no such thing as an out-of-range integer value in a file; the corresponding check is on the string.

Error: `DofStorage` node data is not one of `"Null"`, `"UserDefined"`, `"SharedDofs"`, or `"IndependentDofs"`. Note that the MLL itself reports this as an error via `cgi_DofStorage` for files at or below the library's own version, and as a warning (degrading to `UserDefined`) for files written by a newer library.

Error: `InterfaceContinuity` node data is not one of `"Null"`, `"UserDefined"`, `"ContinuousInterface"`, or `"DiscontinuousInterface"`, with the same newer-library degradation as above.

Error: `SharedDofs` together with `DiscontinuousInterface` — contradictory, since no interface DOF is duplicated under `SharedDofs` and a single shared storage slot cannot hold two differing values (Section 3.3). The diagnostic should name Section 3.2.1 as well: a writer of $\boldsymbol{H}(\mathrm{div})$ or $\boldsymbol{H}(\mathrm{curl})$ data is the likeliest author of this combination, and needs to be told what to write instead rather than only that this is wrong.

Warning: `InterfaceContinuity` present on a node where no DOF duplication

can occur—`GridLocation_t = CellCenter`, or `Vertex` with no `PointList` (Section 3.3.2). For `ContinuousInterface` the assertion is vacuous rather than wrong. For `DiscontinuousInterface` it is unsupportable, there being no differing copies to describe; that case escalates to the error above when the same node also carries `SharedDofs`, which contradicts it directly rather than merely failing to support it.

Warning, optional and off by default: where the field arrays and the DOF correspondence are both available, `cgnscheck` may sample duplicated DOFs and warn when `ContinuousInterface` but the sampled copies differ by more than a tolerance. This check is necessarily partial: for arbitrary interpolation points the correspondence cannot be established at all (Section 4.2), so a clean result is not a guarantee and a failure should be reported as a suspected inconsistency rather than a definite one.

Error: `IndependentDofs` with `GridLocation_t = Vertex` and *no* `PointList` — the SIDS `DataSize` formula yields `VertexSize` (the shared-vertex count), which cannot represent the larger element-local DOF count (Section 3.9).

Error: `IndependentDofs` with `GridLocation_t = Vertex` and a `PointList`, in a *structured* zone — the ordering rule is defined only for unstructured zones (Section 3.9.1).

Error: `IndependentDofs` with `GridLocation_t = Vertex` and a `PointList` whose contents do not equal the zone's element connectivity traversed per Section 3.9.3. Because the required contents are fully determined, this is an exact check, not a heuristic. It subsumes checks for wrong length, wrong ordering, and incomplete element coverage.

Error: `IndependentDofs` with `GridLocation_t = Vertex` and a `PointList` in a zone containing an `NFACE_n` section (Section 3.9.1).

Error: `Rind_t` present alongside `IndependentDofs`, `GridLocation_t = Vertex`, and a `PointList` (Section 3.9.1).

Error: `SharedDofs` with `GridLocation_t = InterpolationPoints` and no point set, or with a point set that lists volume rather than interface element indices (Section 4.2).

Error: `SharedDofs` with `GridLocation_t = IntegrationPoint` — integration-point data is necessarily element-local.

Warning: `SharedDofs` with `GridLocation_t = CellCenter` (or `FaceCenter`, `EdgeCenter`, `[IJK]FaceCenter`) — semantically inconsistent, since such data is inherently one value per entity with no shared-entity ambiguity (Design Decision 3). **Convention:** this combination should not be used.

Warning: A solution node marked `SharedDofs` whose CPEX-0045 `Interpolation-Orders` gives a spatial order of zero. A $p = 0$ basis has no interface DOFs to share.

Warning: A value of `DofStorageUserDefined` without a sibling `Descriptor_t` named `DofStorageDescription` — the convention is then unrecoverable by any reader.

Warning: A `DofStorage` node whose data is `"Null"`. This is legal but pointless: the absence of the node carries the same meaning, and writing `Null` through the MLL removes the node rather than creating one, so such a node indicates a file written outside the MLL.

12 Reference Implementation, Versioning, and Test Plan

12.1 Deliverables

The reference implementation comprises:

1. `cgnslib.h`: both enumerations, their `NofValid...` counts, both `...Name` extern declarations, and prototypes for `cg_dof_storage_read` / `_write`, `cg_interface_continuity_read` / `_write`, `cg_DofStorageName`, and `cg_InterfaceContinuityName`.
2. `cgnslib.c`: both string tables and the six public functions defined there—two read/write pairs and two name helpers. The four Fortran bindings live in `cgns_f.F90`.
3. `cgns_header.h`: the `dof_storage` and `interface_continuity` fields in `cgns_sol`, `cgns_discrete`, `cgns_subreg`, and `cgns_user_data`; prototypes for both address helpers and both string mappers.
4. `cgns_internals.c`: two address helpers, two string mappers, and the four parser updates, each handling both labels.
5. `cgns_f.F90` / `cgns_f.h`: the Fortran bindings and the parameter values for both enumerations.
6. `cgnscheck.c`: the validation rules of Section 11.
7. `cgnsview`: display of the new node (no code change expected; the generic C1 node display suffices).
8. Documentation updates to the SIDS, File Mapping Manual, and MLL reference per Sections 8 and 9.

12.2 Versioning

Target release: CGNS 5.0, jointly with CPEX 0045. The two proposals are being developed together for the same release. This is a deliberate choice rather than a scheduling coincidence: because the `Vertex` encoding defined here and CPEX 0045's `InterpolationPoints` can express overlapping cases (Section 2.4), implementing them in one pass is what guarantees they use consistent vocabulary, that `cgnscheck` validates both against the same model of DOF storage, and that the SIDS text cannot describe the same data in contradictory terms. Neither proposal should be understood as the faster route to the capability.

The library version constant is incremented, and the value at which `DofStorage_t` and `InterfaceContinuity_t` first appear shall be recorded in the SIDS text so that implementers can determine support from the file version alone. The forward-compatibility path described in Section 6 depends on this: `cgi_DofStorage` and `cgi_InterfaceContinuity` degrade unrecognized strings to `UserDefined` only when `cg->version > CGNSLibVersion`. Because both attributes are introduced in the same release, a single version threshold serves both.

12.3 Test Plan

1. **Round trip, all parents, both attributes.** For each of `FlowSolution_t`, `DiscreteData_t`, `ZoneSubRegion_t`, and `UserDefinedData_t`: write each of `SharedDofs`, `IndependentDofs`, and `DofStorageUserDefined`; close; reopen; read back and compare. Repeat for `ContinuousInterface`, `DiscontinuousInterface`, and `InterfaceContinuityUserDefined`.

2. **Independence of the two attributes.** Write each of the nine combinations of the two enumerations (including `Null` for each) on one node; verify that reading either is unaffected by the other, that writing one does not create or remove the other, and that writing `Null` to one leaves the other intact. This is the regression test for the shared address-helper mechanism.

3. **Absent node.** Read from a node with no `DofStorage` child and assert `CG_OK` with `DofStorageNull`. Repeat against a file written by a pre-CPEX-0050 library.

4. **Overwrite in `CG_MODE_MODIFY`.** Write `SharedDofs`, close, reopen in modify mode, write `IndependentDofs`, close, reopen: assert the value is `IndependentDofs` *and* that the parent has exactly one child with label `DofStorage_t`. This is the regression test for duplicate-child creation.

5. **Deletion.** Write `SharedDofs`, then write `DofStorageNull` in modify mode; assert the child node is gone and a subsequent read yields `DofStorageNull`.

6. **Invalid input.** Assert `CG_ERROR` from `cg_dof_storage_write` for out-of-range enum values.

7. **Wrong parent.** Assert `CG_INCORRECT_PATH` when the current position is, e.g., `GridCoordinates_t`.
8. **No current position.** Assert `CG_ERROR` when called without a preceding `cg_goto`.
9. **Forward compatibility.** Hand-construct a file (via CGIO) with `DofStorage` data `"Trace"` and a version greater than the library's; assert the read yields `DofStorageUserDefined` with a warning, not an error. With a version at or below the library's, assert `CG_ERROR`.
10. **Shared-vertex independent layout.** Build the Example 7.5 file; assert `cg_sol_ptset_info` reports `npnts` $= n_{\text{elem}} \times n_{\text{vpe}}$, that `cg_field_read` returns all values, and that `cgnscheck` passes. Then perturb the `PointList` ordering and assert `cgnscheck` reports an error.
11. **`cgnscheck` rules.** One file per rule in Section 11, asserting the expected error or warning.
12. **Fortran.** Mirror tests 1–4 through the Fortran bindings.
13. **Parallel.** Under `cgp_open`, call `cg_dof_storage_write` collectively from all ranks and assert the file is valid and single-valued; confirm on 1, 2, and 4 ranks.
14. **Backward compatibility of existing tests.** The full existing CGNS test suite shall pass unchanged.

13 Summary of API Additions

Symbol	Description
Public (`cgnslib.h`)	
`DofStorage_t`	New enumeration type: `DofStorageNull`, `DofStorageUserDefined`, `SharedDofs`, `IndependentDofs` (see Section 3.2.3 on the unprefixed value names).
`DofStorageName`	String table for the enumeration values.
`NofValidDofStorage`	Count of valid enumeration values (4).
`cg_DofStorageName`	C: return the string name for a `DofStorage_t` value (mirrors `cg_GridLocationName`).
`cg_dof_storage_write`	C: write the DOF storage layout of the node at the current position.
`cg_dof_storage_read`	C: read the DOF storage layout of the node at the current position.

Symbol	Description
`cg_dof_storage_write_f`	Fortran binding for `cg_dof_storage_write`.
`cg_dof_storage_read_f`	Fortran binding for `cg_dof_storage_read`.
`InterfaceContinuity_t`	New enumeration type: `InterfaceContinuityNull`, `InterfaceContinuityUserDefined`, `ContinuousInterface`, `DiscontinuousInterface`. Stored on disk under the same names; see Section 3.2.3.
`InterfaceContinuityName`	String table for the enumeration values.
`NofValidInterfaceContinuity`	Count of valid enumeration values (4).
`cg_InterfaceContinuityName`	C: return the string name for an `InterfaceContinuity_t` value.
`cg_interface_continuity_write`	C: write the interface continuity of the node at the current position.
`cg_interface_continuity_read`	C: read the interface continuity of the node at the current position.
`cg_interface_continuity_write_f`	Fortran binding.
`cg_interface_continuity_read_f`	Fortran binding.
Internal (`cgns_header.h`)	
`cgi_dof_storage_address`	Resolve the current position to the parent's `dof_storage` field; delete an existing child on modify.
`cgi_DofStorage`	Map a `DofStorage` string to the enumeration value.
`dof_storage`	New field in `cgns_sol`, `cgns_discrete`, `cgns_subreg`, `cgns_user_data`.
`cgi_interface_continuity_address`	As `cgi_dof_storage_address`, for the second attribute.
`cgi_InterfaceContinuity`	Map an `InterfaceContinuity` string to the enumeration value.
`interface_continuity`	Second new field in the same four structures.

1400

Eight accessor functions—a read/write pair per attribute, each with a Fortran binding—cover both attributes on all four parent node types, together with the two `cg_...Name` helpers, for ten new public entry points in total. No per-parent variants and no `cgp_*` variants are required; see Sections 3.5 and 6. The second attribute costs one additional address helper, one additional string mapper, and one additional field per structure, because it reuses the mechanism the first one establishes.

14 References

References

[1] CGNS Standard Interface Data Structures (SIDS), https://cgns.org/standard/SIDS/CGNS_SIDS.html

[2] CGNS File Mapping Manual—Detailed CGNS Node Descriptions, https://cgns.org/standard/FMM/nodes.html

[3] CGNS Mid-Level Library—Flow Solution, https://cgns.org/standard/MLL/api/c_api.html

[4] CPEX 0045, "Polynomial Data and Curved Grid Elements," https://github.com/CGNS/CGNS/issues/577

[5] CPEX 0047, "Quadrature Rules Definition and Data Storage" (Integration Point Definition), M. Philit and F. Huvelin, https://cgns.org/current/CPEX.html

[6] Hesthaven, J.S. and Warburton, T., *Nodal Discontinuous Galerkin Methods*, Springer, 2008.

[7] Cockburn, B., Karniadakis, G.E., and Shu, C.-W., eds., *Discontinuous Galerkin Methods: Theory, Computation, and Applications*, Springer, 2000.

[8] Miller, M.C., "On the Distinction Between Topological Space and Physical Space in SAF: In other words, what are topo-relations and dof-relations?" ASCI-DMF / SAF Internal Draft, 2003 (rev. 2014).