CGNS Proposal for Extension 0049

CGNS Compatibility Version Control

Version 1.0

March 28, 2026

CPEX#	0049
Scope	Compatibility version control for improved interoperability
Champion	M. Scot Breitenfeld
Organization	The HDF Group
E-mail	brtnfld@hdfgroup.org
GitHub Issue	`https://github.com/CGNS/CGNS/issues/915`
Reference Impl.	`https://github.com/brtnfld/CGNS/tree/915`
Target Release	CGNS v5.0.0
Date First Posted	March 28, 2026
SIDS Status	proposed
Filemap Status	proposed
MLL Status	proposed

Contents

1 Abstract

This CPEX proposes a compatibility version control mechanism for the CGNS Mid-Level Library (MLL), inspired by the HDF5 library's `H5Pset_libver_bounds` facility. The current CGNS library rejects files whose recorded `CGNSLibraryVersion_t` exceeds the running library version, even when the file contains no version-specific features that would prevent correct interpretation. This unnecessarily breaks interoperability between tools compiled against different CGNS library releases.

The extension introduces API functions and configuration options that allow applications to specify version constraints for reading and writing, and shifts read-time version enforcement from a simple numeric comparison to a feature-aware compatibility check. A accompanying *automatic write-version* mode selects the minimum necessary version based on the features actually written, maximizing compatibility without requiring applications to hard-code version numbers.

2 Motivation and Rationale

2.1 Problem Statement

When opening a CGNS file, the library currently enforces the following logic in `cg_open`:

```
if (cg->version > CGNSLibVersion) {
    /* Reject if major version of file > major version of library */
    if ((cg->version / 1000) > (CGNSLibVersion / 1000)) {
        cgi_error("A more recent version of the CGNS library "
                  "created the file. Therefore, the CGNS library "
                  "needs updating before reading the file '%s'.",
                  filename);
        return CG_ERROR;
    }
    /* Warn only if different in second digit */
    if ((cg->version / 100) > (CGNSLibVersion / 100)) {
        cgi_warning("The file being read is more recent "
                    "that the CGNS library used");
    }
}
```

This version-number-only check creates several practical problems:

1. **False rejections.** A file written by CGNS 5.0 that uses only features present since CGNS 4.x will be rejected by a CGNS 4.x library, even though it could be read correctly.
2. **Blocked tool chains.** Widely-used post-processing tools (Tecplot, ParaView, VisIt) may ship with an older embedded CGNS library. Users who generate files with a newer library version cannot visualize their own data without recompiling or downgrading.
3. **Hindrance to incremental adoption.** Organizations running mixed CGNS library versions across production codes, mesh generators, and post-processors are forced into synchronized upgrades—an unrealistic constraint in large multi-team environments.

2.2 Precedent: HDF5 Library Version Bounds

HDF5 solved an analogous problem with the `H5Pset_libver_bounds(fapl, low, high)` API, which lets applications declare the range of HDF5 library versions whose on-disk representations they are willing to produce or consume. This approach:

- Decouples the *library version* from the *SIDS version*.
- Allows older libraries to read newer files when no incompatible features are present.
- Gives applications explicit control over the compatibility vs. feature trade-off.

The present CPEX adapts this design for CGNS.

3 Detailed Description

3.1 Design Overview

The implementation introduces three accompanying changes to the CGNS Mid-Level Library:

1. **Version milestone constants** (`CG_LIBVER_*`) encoding recognized CGNS version milestones as plain integer macros.
2. **New API functions** for setting and querying compatibility version bounds (lower and upper) on both a global and per-file basis, plus a parameter-object API for thread-safe per-open configuration.
3. **Feature-aware open and write logic** in `cg_open` that replaces the current numeric-only version check with a feature-compatibility assessment, and an automatic write-version mode that records the minimum version required by the features actually written.

3.2 File Format Alternatives

This CPEX must record two pieces of information that do not currently exist in the file: (a) the minimum CGNS version required to read the file, and (b) a diagnostic bitmask indicating which version-specific features are present. Two file-format approaches are presented below. **Alternative B is strongly recommended** because it preserves data provenance—a fundamental requirement for CFD archival formats. Alternative A is included for completeness so the committee can evaluate both approaches, but its semantic overloading of `CGNSLibraryVersion_t` makes it unsuitable for production use.

3.2.1 Alternative A: Overload `CGNSLibraryVersion_t`

Under this approach, no new nodes are introduced. Instead, the existing `CGNSLibraryVersion_t` node is redefined:

- When the library automatically determines the minimum version needed (see Section 3.6), it stamps `CGNSLibraryVersion_t` with the *minimum required version* instead of the writing library's version. A file written by CGNS 5.0 that uses only V3.2 features will carry `CGNSLibraryVersion_t = 3200`.
- A hidden diagnostic bitmask attribute (`_CGNS_FeatureMask`, described in Section 3.8) is stored on `CGNSLibraryVersion_t`.
- The SIDS must redefine the node's semantics from "the library that created this file" to "the minimum library required to read it."

Advantages: No new nodes; no file-mapping additions beyond one hidden attribute; maximum structural simplicity.

Disadvantages: Provenance information (which library wrote the file) is lost when automatic versioning is used; requires a semantic redefinition of a long-established SIDS node; tools or workflows that depend on `CGNSLibraryVersion_t` for debugging, auditing, or compliance will break. Additionally, when an older file is opened and the AUTO write mode determines a minimum version lower than what `CGNSLibraryVersion_t` currently records, the node value would be silently overwritten—modifying the file's metadata without any user action or awareness.

3.2.2 Alternative B: New `CGNSMinRequiredVersion_t` node (recommended)

Under this approach, `CGNSLibraryVersion_t` retains its original meaning and a new sibling node is introduced:

`CGNSMinRequiredVersion_t`

```
CGNSMinRequiredVersion_t := ‘‘CGNSMinRequiredVersion’’
Label = ‘‘CGNSMinRequiredVersion_t’’
Data-Type = ‘‘R4’’
Dimension = 1, Dimension-Value = 1
Data = CGNSMinRequiredVersion
```

The node is a child of the file root node, at the same level as `CGNSLibraryVersion_t` and `CGNSBase_t`. Its value is an `R4` real number using the same decimal convention as `CGNSLibraryVersion_t` (e.g., `3.200` for CGNS 3.2, `5.000` for CGNS 5.0). The diagnostic bitmask (`_CGNS_FeatureMask`, Section 3.8) is placed on this new node.

- `CGNSLibraryVersion_t` is **always** written as the current library version (existing behavior, unchanged).
- `CGNSMinRequiredVersion_t` receives the minimum version required to read the file (determined automatically or set explicitly by the application).
- Old libraries silently ignore the new node (standard CGNS convention for unknown nodes).

Advantages:

- No semantic redefinition of any existing SIDS node; `CGNSLibraryVersion_t` is entirely unchanged in both meaning and structure.
- Provenance preserved: `CGNSLibraryVersion_t` always identifies the writing library.
- The MODIFY-mode logic is simplified: `CGNSLibraryVersion_t` is always upgraded to the current library version (existing behavior); only `CGNSMinRequiredVersion_t` uses the automatic version mechanism.
- Debugging and compliance workflows continue to work—the question "which library wrote this?" has a definitive answer.

Disadvantages:

- Requires a SIDS addition (new node type) and a corresponding file-mapping update; in particular, the root-node schema in SIDS Section 2 must be formally amended.
- Introduces one additional small node per file.

3.2.3 Comparison

	Alt. A (overload)	Alt. B (new node)
New SIDS nodes	0	1
SIDS semantic changes	1 (redefinition)	0
Provenance preserved	×	✓
MODIFY-mode logic	Special case	Existing behavior
Old library reads new file	Works	Works (ignores new node)
New library reads old file	Falls back to scan	Falls back to scan
Hidden attributes	1	1 (on new node)

Alternative B is recommended. Adding a new SIDS node is less disruptive than redefining the semantics of an established one, and preserving provenance is valuable for the long-lived CFD files that are the primary motivation for this CPEX.

3.2.4 Alternative B: Node Placement—Root Sibling vs. Child of `CGNSLibraryVersion_t`

Within Alternative B, a secondary design question arises: should `CGNSMinRequiredVersion_t` be placed as a *sibling* of `CGNSLibraryVersion_t` at the file root, or as a *child* of `CGNSLibraryVersion_t`?

Child of `CGNSLibraryVersion_t`

Advantages:

- All version metadata is co-located under one existing node; readers that need both values traverse a single subtree.
- The root namespace does not grow.
- The SIDS amendment is scoped to an existing node type rather than the root-node schema.

Disadvantages:

- `CGNSLibraryVersion_t` is no longer structurally unchanged—its SIDS definition must be amended to allow an optional child, negating one of Alt. B's stated advantages over Alt. A.
- `CGNSLibraryVersion_t` is currently defined as a scalar leaf node; promoting it to a container changes its character in the SIDS type system.

Root sibling (recommended)

Advantages:

- Clean semantic separation: "written by version X" (`CGNSLibraryVersion_t`) and "requires at least version Y" (`CGNSMinRequiredVersion_t`) are independent facts that need not be physically nested.
- `CGNSLibraryVersion_t` is truly unchanged in both semantics and structure, preserving Alt. B's core advantage.
- Consistent with all existing root-level CGNS metadata nodes, which are siblings rather than children of one another.

- The root-schema amendment (SIDS Section 2) is no more invasive than a `CGNSLibraryVersion_t` structural amendment would be.

Disadvantages:

- Root namespace grows by one node type; strict validators that enumerate expected root children may warn until updated.
- Readers that need both version values must visit two sibling nodes rather than one parent.

Root sibling placement is adopted in this proposal.

3.3 Version Constants: `CG_LIBVER_*`

Version milestone constants are defined in `cgnslib.h` as plain integer macros using the encoding MAJOR $\times$ 1000 + MINOR $\times$ 100 (matching the existing `CGNSLibraryVersion_t` node format):

```
/* Named constants are defined only for versions where a SIDS change
 * was introduced that requires version tracking in the write path
 * (cgi_require_version) or a feature detector in the registry.
 * Intermediate releases with no SIDS changes (e.g., 3.3, 4.1-4.4)
 * are omitted; use the raw integer encoding for those if needed:
 *   MAJOR * 1000 + MINOR * 100  (e.g., 4400 for CGNS 4.4)
 */
#define CG_LIBVER_EARLIEST  1050  /* CGNS 1.05 (oldest library version) */
#define CG_LIBVER_V12       1200  /* CGNS 1.2  - CGNSBase_t gained CellDimension
                                   *             and PhysicalDimension fields;
                                   *             used in write-path version bump */
#define CG_LIBVER_V30       3000  /* CGNS 3.0  - Extended element types */
#define CG_LIBVER_V31       3100  /* CGNS 3.1  - Reordered element types */
#define CG_LIBVER_V32       3200  /* CGNS 3.2  - NGON/NFACE v3.2 format */
#define CG_LIBVER_V40       4000  /* CGNS 4.0  - ElementStartOffset */
#define CG_LIBVER_V45       4500  /* CGNS 4.5  - Particles (CPEX 0046) */
#define CG_LIBVER_V50       5000  /* CGNS 5.0  - High-order elements (CPEX 0045) */
#define CG_LIBVER_LATEST    5000  /* Current library version */
#define CG_LIBVER_AUTO        -1  /* Automatic write-version selection */
```

`CG_LIBVER_EARLIEST` denotes the oldest CGNS library version. `CG_LIBVER_LATEST` is an alias for the current library version at compile time. `CG_LIBVER_AUTO` is a special sentinel directing the library to choose the minimum write version automatically (see Section 3.6). The internal feature name and detector associated with each constant are catalogued in the registry table in Section 3.7.

3.4 New MLL Functions

3.4.1 Global Bounds via `cg_configure`

The global version bounds are set and queried through the existing `cg_configure` interface using the following new tokens:

```
#define CG_CONFIG_LIBVER_LOW       10  /* Set lower bound */
#define CG_CONFIG_LIBVER_HIGH      11  /* Set upper bound */
#define CG_CONFIG_GET_LIBVER_LOW   12  /* Query lower bound */
#define CG_CONFIG_GET_LIBVER_HIGH  13  /* Query upper bound */
```

These values (10–13) are the next available after the existing `cg_configure` tokens (1–9 are currently assigned).

Defaults: `low = CG_LIBVER_AUTO`, `high = CG_LIBVER_LATEST`.

```c
/* Set lower bound (write-version floor) */
cg_configure(CG_CONFIG_LIBVER_LOW,
             (void *)(intptr_t)CG_LIBVER_AUTO);

/* Set upper bound (read/write ceiling) */
cg_configure(CG_CONFIG_LIBVER_HIGH,
             (void *)(intptr_t)CG_LIBVER_V40);

/* Query current bounds */
int lo, hi;
cg_configure(CG_CONFIG_GET_LIBVER_LOW,  &lo);
cg_configure(CG_CONFIG_GET_LIBVER_HIGH, &hi);
```

Scope: Global—applies to all subsequent `cg_open` calls until changed. Not thread-safe; use the parameter-object API (Section 3.4.3) for thread-safe per-file configuration.

3.4.2 Library Version Bounds Functions

The following functions provide per-file control—including post-open adjustment and thread-safe multi-file scenarios:

```c
int cg_set_libver_bounds(int fn,
                         int low, int high);

int cg_get_libver_bounds(int fn,
                         int *low, int *high,
                         int *min_version);
```

`cg_get_libver_bounds` retrieves version information for an open file. Any output pointer may be `NULL` to skip that value:

`low` Current configured lower bound ($O(1)$). Pass `NULL` to skip.

`high` Current configured upper bound ($O(1)$). Pass `NULL` to skip.

`min_version` Minimum CGNS version required by the features actually present in the file. **Only computed when non-`NULL`**; incurs an $O(N_{\text{zones}} \times N_{\text{sections}})$ feature scan on the first call (cached for subsequent calls). Pass `NULL` to avoid the scan entirely.

Terminology note. The three output parameters serve distinct purposes and should not be confused:

- `low` is the *configured lower bound*—the minimum SIDS version produced on write. When set to `CG_LIBVER_AUTO`, the library determines this automatically from the features written. It is a policy setting, not a file property.
- `high` is the *configured upper bound*—the maximum feature level the application is willing to encounter. It is also a policy setting.
- `min_version` is the *floor the file requires*—the oldest library version capable of reading the features actually present. It is derived from the file's content, not from any user setting.

This NULL-gated design keeps the API surface small while preserving full cost control: applications that only need the configured bounds pass `NULL` for `min_version` and pay $O(1)$; applications that need the computed minimum version pay the scan cost exactly once.

Behavior of `min_version` in write mode. When called on a file opened with `CG_MODE_WRITE` or `CG_MODE_MODIFY` before any nodes have been written (or after a node deletion), `min_version` returns the current *effective write version*: the library version that will be recorded in `CGNSLibraryVersion_t` when the file is closed. When `low` is `CG_LIBVER_AUTO`, this begins at `CG_LIBVER_EARLIEST` and increments as features are written; when `low` is an explicit constant, it returns at least that value. It does *not* return `CG_LIBVER_AUTO` regardless of the `low` setting.

By contrast, HDF5's `H5Pget_libver_bounds` retrieves only the configured policy (`low`, `high`); HDF5 provides no equivalent of the `min_version` output. The ability to query the minimum version actually required by file content is a capability unique to this CGNS extension.

3.4.3 Parameter Object API

For per-open configuration—including version bounds, file type, and compression—a parameter object type `cg_parameters_t` is provided. This is an opaque pointer; all operations go through the API.

Thread-safety note. The parameter object makes *configuration* per-call and race-free, but the MLL itself is not fully thread-safe: `cg_open` still accesses the global file table (`cg_files[]` array). Applications that open different files from concurrent threads must serialize their `cg_open`/`cg_close` calls or use external locking.

```c
/* Opaque handle (NULL == use global configuration) */
#define CG_PARAMS_DEFAULT ((cg_parameters_t)NULL)

/* Parameter keys */
typedef enum {
    CG_PARAM_FILE_TYPE    = 100, /* CG_FILE_HDF5, CG_FILE_ADF, etc. */
    CG_PARAM_COMPRESS     = 101, /* Compression level: 0 (none) to 9 */
    CG_PARAM_LIBVER_LOW   = 102, /* Lower bound: CG_LIBVER_AUTO or
                                    * explicit CG_LIBVER_* constant */
    CG_PARAM_LIBVER_HIGH  = 103  /* Upper bound: CG_LIBVER_* ceiling */
} CG_PARAM_KEY;

/* Lifecycle */
int cg_params_create(cg_parameters_t *params);
int cg_params_destroy(cg_parameters_t  params);

/* Generic setter (follows cg_configure pattern) */
int cg_params_set(cg_parameters_t params, int key, void *value);
```

Default values when a parameter object is created with `cg_params_create`:

- `CG_PARAM_COMPAT_LOW`: `CG_LIBVER_AUTO`
- `CG_PARAM_COMPAT_HIGH`: `CG_LIBVER_LATEST`
- `CG_PARAM_FILE_TYPE`: current `cgns_filetype` global
- `CG_PARAM_COMPRESS`: current `cgns_compress` global

A parameter object is passed to the 4-argument form of `cg_open_with_params`:

```
int cg_open_with_params(const char      *filename,
                        int              mode,
                        cg_parameters_t  params,
                        int             *fn);
```

For source-level backward compatibility `cg_open` is defined as a polymorphic macro dispatching on argument count:

```
/* 3-argument (legacy): uses global configuration */
cg_open(filename, mode, &fn);

/* 4-argument: uses supplied parameter object */
cg_open(filename, mode, params, &fn);
```

Passing `CG_PARAMS_DEFAULT` as the `params` argument is equivalent to the 3-argument call.

Design rationale: parameter object vs. extra arguments. Four API patterns were considered for passing per-open configuration to `cg_open`:

1. **Extra positional arguments.**

   ```
   /* Hypothetical: direct extra args (low, high) */
   cg_open(file, mode, &fn,
           CG_LIBVER_V40, CG_LIBVER_LATEST);
   ```

   Simple for a fixed parameter set, but every new configuration option requires a new function signature or a new overloaded variant. The polymorphic `cg_open` macro already uses the 4-argument slot for the params object; adding further positional arguments would exhaust this approach quickly.

2. **Options struct with named fields.**

   ```
   /* Hypothetical: named-field struct */
   cg_open_opts opts = {
       .low       = CG_LIBVER_V40,
       .high      = CG_LIBVER_LATEST,
       .file_type = CG_FILE_HDF5,
   };
   cg_open_ex(file, mode, &opts, &fn);
   ```

   More discoverable—fields are visible in the struct definition—but not ABI-stable. Adding a new field changes the struct layout for all callers, requiring either a versioned struct (with a `size` sentinel as used in POSIX's `struct addrinfo`) or a full API break.

3. **Variadic key–value pairs.**

   ```
   /* Hypothetical: key-value varargs (as used by cg_goto) */
   cg_open_ex(file, mode, &fn,
              CG_PARAM_LIBVER_LOW,  CG_LIBVER_V40,
              CG_PARAM_LIBVER_HIGH, CG_LIBVER_LATEST,
              CG_PARAM_END);
   ```

   CGNS already uses this pattern in `cg_goto`, which takes variadic `(label, index)` pairs terminated by a sentinel. The approach is familiar within the library, requires no heap

allocation, and avoids a separate create/destroy lifecycle. However, it is not type-safe—all values must be passed as `int` or cast through a common type—and it cannot be passed as a single opaque handle to a downstream function without reconstructing the argument list. It also cannot hold non-integer parameters (e.g., future string-valued options) without a type-tag convention.

4. **Property object / opaque handle (chosen design).**

```
cg_params_create(&params);
cg_params_set(params, CG_PARAM_LIBVER_LOW,
              (void *)(intptr_t)CG_LIBVER_V40);
cg_params_set(params, CG_PARAM_LIBVER_HIGH,
              (void *)(intptr_t)CG_LIBVER_LATEST);
cg_open(file, mode, params, &fn);
cg_params_destroy(params);
```

   More verbose for simple cases, but ABI-stable: new parameters are added as new `CG_PARAM_*` keys with no change to any function signature or struct layout. It also directly mirrors HDF5's File Access Property List pattern (`H5Pcreate`/`H5Pset_*`/`H5Fopen`), which is already familiar to the scientific computing community.

The property object was chosen because CGNS library releases are infrequent, ABI compatibility across releases is important to users, and the `cg_open` macro preserves the original 3-argument call for the common case so that existing code requires no changes.

Known limitation: macro overloading. The polymorphic `cg_open` macro is implemented via C99 variadic macro argument counting. Some vendor preprocessors do not support the `__VA_ARGS__` counting trick reliably.

The reference implementation provides `cg_open_with_params` as the explicit 4-argument entry point, alongside the original `cg_open`:

```
/* Existing API: strict 3-argument function */
CGNSDLL int cg_open(const char *filename, int mode, int *fn);

/* Explicit 4-argument API for per-open configuration */
CGNSDLL int cg_open_with_params(const char *filename, int mode,
                                cg_parameters_t params, int *fn);
```

Both symbols are exported from the shared library. Language bindings that use `dlsym` or FFI (Python `ctypes`, Java JNI, etc.) resolve these symbols directly; the macro is not involved.

The polymorphic convenience macro is active only for compiled C code and can be suppressed by defining `CGNS_NO_MACROS`:

```c
#ifndef CGNS_NO_MACROS
  /* Macro overload: 3-arg -> cg_open_with_params(f,m,DEFAULT,fn)
     4-arg -> cg_open_with_params(f,m,p,fn) */
  #define cg_open(...) /* variadic dispatch */
#endif
```

3.5 Modified cg_open Behavior

When a file is opened for reading (`CG_MODE_READ` or `CG_MODE_MODIFY`), the revised `cg_open` performs the following checks in order:

1. **Read** the `CGNSLibraryVersion_t` node, which records the library version that wrote the file, denoted V_{file}.
2. **Major version safety check.** If the major version of V_{file} exceeds the major version of the running library ($\lfloor V_{\text{file}}/1000 \rfloor > \lfloor V_{\text{lib}}/1000 \rfloor$), return `CG_ERROR` immediately. This is a *crash-prevention guard*, not a compatibility check: a major version increment may indicate fundamentally different internal data layouts that would corrupt memory if the library attempted to parse them. It must therefore run *before* Step 3 reads the full file structure. It is distinct from the feature compatibility check in Step 4, which operates on already-parsed in-memory structures and cannot substitute for this guard.
3. **Read file structure** via `cgi_read()`.
4. **Enforce upper bound.** If the user has set $V_{\text{high}} <$ `CG_LIBVER_LATEST` and $V_{\text{file}} > V_{\text{high}}$, return `CG_ERROR` immediately. When the user explicitly requests a ceiling, that ceiling is *strict*: the library cannot guarantee the absence of unknown features beyond registry coverage, so it must reject rather than silently accept. This step is skipped when $V_{\text{high}} =$ `CG_LIBVER_LATEST` (the default), allowing feature-aware acceptance in Step 5.
5. **Check feature compatibility.** If $V_{\text{file}} > V_{\text{lib}}$, check the file's metadata for features introduced after the running library's version. If an incompatible feature is detected, return `CG_ERROR` with a diagnostic naming the feature and the required minimum library version.
6. **Accept.** If no incompatibility is found, open the file successfully.
7. **Warn if appropriate.** If $V_{\text{file}} > V_{\text{lib}}$ but the file was accepted, issue a `cgi_warning` so that the application can log the condition.

Note on the lower bound and reads. The lower bound (`low`) controls write behavior only; it does not cause rejection of files on read. Because CGNS maintains full backward compatibility, a newer library can always read files written by older versions. Older files are always readable regardless of the `low` setting.

Performance note. For files written by a CPEX-0049-aware library, Step 5 is resolved in $O(1)$ using the `CGNSMinRequiredVersion_t` node or `_CGNS_FeatureMask` attribute (see Sections 3.2 and 3.8). The $O(N_{\text{zones}} \times N_{\text{sections}})$ feature scan is only required as a fallback for legacy files that lack these metadata. Even then, the scan is skipped when $V_{\text{high}} \geq$ `CG_LIBVER_LATEST` and $V_{\text{file}} \leq V_{\text{lib}}$—the common case for default-configured applications.

3.6 Write-Mode Version Selection

The write version is the value recorded in the file's `CGNSLibraryVersion_t` node. It is governed by the lower bound (`low`). Two modes are supported:

`low = CG_LIBVER_AUTO` **(default).** For new files (`CG_MODE_WRITE`), the library begins with an effective version of `CG_LIBVER_EARLIEST` and upgrades incrementally via an internal `cgi_require_version()` call each time an API function writes a version-specific feature. For existing files (`CG_MODE_MODIFY`), the effective version is initialized to max(existing `CGNSMinRequiredVersion_t`, `CG_LIBVER_EARLIEST`) and the in-memory feature mask is loaded from the existing `_CGNS_FeatureMask` (or populated via fallback scan for legacy files). In both cases, the file is stamped with the minimum version required by the features actually present, maximizing compatibility with older readers.

Maintenance requirement. The AUTO mechanism is only correct if *every* write path that introduces a version-specific feature calls `cgi_require_version()` before writing. A maintainer who implements a new CPEX and omits this call will silently produce a file whose version header is too low, breaking readers that rely on the header to detect the new feature. To guard against this, a CI test must write every known node type to a fresh file in AUTO mode and assert that the resulting `CGNSLibraryVersion_t` equals the expected minimum version for that node type.

`low = explicit version constant.` Setting `low` (via `CG_PARAM_COMPAT_LOW` or `CG_CONFIG_COMPAT_LOW`) to a specific `CG_LIBVER_*` constant forces at least that version to be written. Any attempt to write a feature whose required version exceeds `high` returns `CG_ERROR`.

`CG_MODE_MODIFY` **handling.** When opening an existing file for modification, the two version nodes are treated differently:

- `CGNSLibraryVersion_t` is **always** upgraded to the current library version V_{lib}, regardless of the `low` setting. This preserves its SIDS semantics: "the version of the library that last wrote the file."
- `CGNSMinRequiredVersion_t` (Alternative B) uses the AUTO incremental-upgrade mechanism: the effective version is initialized to the existing node's value (or the result of the fallback feature scan for legacy files) and only increases as new features are written. This ensures monotonicity without recomputing from scratch.

Parallel safety. In parallel I/O mode, each rank may write different features and therefore arrive at a different effective version. Because the `CGNSLibraryVersion` node is a single shared object, all ranks must agree on its final value.

The recommended approach is *sync-at-close*: during `cgp_close`, the library performs an `MPI_Allreduce(MPI_MAX)` on each rank's locally tracked effective version before writing the version node and closing the file.

Note: this `MPI_Allreduce` is new. In the current implementation, `cgp_close` is a one-line wrapper that simply calls `cg_close`; it performs no MPI collective of its own. The proposed reduction is therefore a genuine addition to the close path. It is feasible because `cgp_close` is already required to be called collectively—`cg_close` calls `H5Fclose`, which is a collective operation in parallel HDF5. The `MPI_Allreduce` does not change the collectivity contract of `cgp_close`, but it does add a new synchronisation point that implementations must account for.

The sequence for parallel AUTO mode is:

1. Each rank opens the file via `cgp_open` with `low = CG_LIBVER_AUTO`. The effective version starts at `CG_LIBVER_EARLIEST`.

2. As each rank writes features, `cgi_require_version()` bumps that rank's effective version locally—no file metadata is written and no error is returned.

3. At `cgp_close` time, before flushing metadata, the library performs a single collective reduction using a custom `MPI_Op`. Both values are packed into a two-element `int64_t` buffer and reduced in one call:

```
int64_t buf[2] = { effective_version, current_feature_mask };
MPI_Allreduce(MPI_IN_PLACE, buf, 2, MPI_INT64_T,
              cgp_version_mask_op, comm);
effective_version    = buf[0];
current_feature_mask = buf[1];
```

   The custom operator `cgp_version_mask_op` applies `MAX` to element 0 (version) and bitwise-`OR` to element 1 (mask):

```
static void cgp_version_mask_reduce(
        void *in, void *inout, int *len, MPI_Datatype *dt)
{
    int64_t *a = (int64_t *)in;
    int64_t *b = (int64_t *)inout;
    for (int i = 0; i < *len; i += 2) {
        b[i]   = a[i]   > b[i] ? a[i] : b[i]; /* MAX: version */
        b[i+1] = a[i+1] | b[i+1];              /* OR:  mask    */
    }
}
```

   This operator is registered once at library initialisation via `MPI_Op_create`.

4. The collective maximum version and merged bitmask are written to `CGNSLibraryVersion_t`, `CGNSMinRequiredVersion_t` (Alternative B), and `_CGNS_FeatureMask`, and the file is closed normally.

If an application sets an explicit `low` bound instead of AUTO, that version applies identically on all ranks and no reduction is needed. If a rank attempts to write a feature whose minimum version exceeds the configured `high` bound, the library returns `CG_ERROR` immediately on that rank.

3.7 Feature-Version Registry

The library maintains an internal, statically-compiled registry mapping CGNS features to the version in which they were introduced. Each entry has the form:

```c
typedef struct {
    const char *feature_name;     /* e.g. "ParticleZone_t" */
    int         min_version;      /* CG_LIBVER_* constant  */
    int (*detector)(cgns_base *); /* returns 1 if feature present */
} cgns_feature_version;
```

The current registry (in `src/cgns_internals.c`):

Feature	Min Version	Constant (§3.3)
`Extended_ElementTypes`	3000	`CG_LIBVER_V30`
`Reordered_ElementTypes`	3100	`CG_LIBVER_V31`
`NGON_NFACE_V32`	3200	`CG_LIBVER_V32`
`ElementStartOffset`	4000	`CG_LIBVER_V40`
`ParticleZone_t`	4500	`CG_LIBVER_V45` (CPEX 0046)
`ParticleCoordinates_t`	4500	`CG_LIBVER_V45` (CPEX 0046)
`ParticleSolution_t`	4500	`CG_LIBVER_V45` (CPEX 0046)
`ElementInterpolation_t`	5000	`CG_LIBVER_V50` (CPEX 0045)
`SolutionInterpolation_t`	5000	`CG_LIBVER_V50` (CPEX 0045)
`HighOrder_ElementTypes`	5000	`CG_LIBVER_V50` (CPEX 0045)
`Unknown_Modern_Features`	`CG_LIBVER_LATEST`	Fail-safe

Note on `CG_LIBVER_V12`. The constant `CG_LIBVER_V12` (1200) is used in the write-path version bump (`cgi_require_version`) when writing `CGNSBase_t` nodes, but has no corresponding registry entry or detector. This is intentional: every modern CGNS file contains the V1.2 base structure (with `CellDimension` and `PhysicalDimension`), so a detector would always return true and serve no diagnostic purpose.

The fail-safe entry `Unknown_Modern_Features` catches any features not recognized by the library (currently, element types beyond `HEXA_125`), preventing silent data corruption when reading files written with future CGNS versions.

Hard guard: modifying files that trigger the fail-safe. When a file containing unknown element types is opened in `CG_MODE_MODIFY`, the library cannot safely *preserve* unknown data structures during a partial rewrite. For example, if a CGNS 5.0 library opens a CGNS 6.0 file containing `HEXA_216` elements and then modifies an unrelated zone, the write path has no knowledge of how to serialize the unknown element data back to disk. The result would be a corrupted or truncated file. Therefore, `cg_open` **must return** `CG_ERROR` when `Unknown_Modern_Features` is detected and the mode is `CG_MODE_MODIFY`. The error message must identify the unrecognized features so the user can decide whether to upgrade the library or open the file in `CG_MODE_READ` instead. Opening the same file in `CG_MODE_READ` remains permitted; only write-capable modes are rejected.

During the feature-compatibility scan, each detector function examines the CGNS tree for the relevant node types. Detectors that operate on nodes not yet parsed into MLL data structures (e.g. `ElementInterpolation_t`) use `cgi_get_nodes()` to scan the raw file tree directly.

Known limitation: low-level write bypasses. The AUTO version-tracking mechanism relies on `cgi_require_version()` being called from the MLL write path. Applications that write version-specific nodes via low-level escapes—such as `cg_user_data_write`, direct `cgio_*` calls, or external HDF5/ADF tools—bypass the MLL entirely and will not trigger `cgi_require_version()`. The

library version node and `_CGNS_FeatureMask` will then under-report the features present. Users who write SIDS nodes through low-level interfaces are responsible for calling `cg_configure(CG_CONFIG_WRITE_VERSION, ...)` to set the version explicitly, or for accepting that the resulting file may be misidentified as compatible by compatibility-checking readers.

3.8 Eliminating the $O(N)$ Feature Scan

The $O(N_{\text{zones}} \times N_{\text{sections}})$ feature scan is unacceptable for production HPC files, which routinely contain 100 000 or more unstructured blocks.

Under Alternative B (Section 3.2), the `CGNSMinRequiredVersion_t` node already eliminates this cost for the primary use case: the open-time compatibility check reads a single integer from the new node and compares it against the configured compatibility version in $O(1)$. **No feature scan is required** when the node is present. The $O(N)$ scan is only needed as a fallback for legacy files that lack the node.

The `_CGNS_FeatureMask` attribute provides an additional **diagnostic** benefit: when a file is rejected, the bitmask identifies *which* features caused the incompatibility, enabling precise error messages (e.g., "file requires CGNS $\geq$ 4.5 for `ParticleZone_t`") without performing a full scan. It is therefore specified as a recommended attribute, not a strict requirement for the $O(1)$ check.

Under Alternative A, where there is no `CGNSMinRequiredVersion_t` node, the feature mask is the *only* $O(1)$ mechanism and is therefore a core requirement.

The reference implementation does not yet include this attribute; it should be added before the CPEX is finalized.

3.8.1 Specification

When a CPEX-0049-aware library closes a writable file, it writes a hidden attribute named `_CGNS_FeatureMask`. Under Alternative A (Section 3.2), the attribute is placed on the existing `CGNSLibraryVersion_t` node; under Alternative B (recommended), it is placed on the new `CGNSMinRequiredVersion_t` node. The attribute is stored as a **signed 64-bit integer** (`I8` in SIDS `DataType_t` terms). In the HDF5 backend this maps to `H5T_NATIVE_INT64`; in the legacy ADF backend, to `I8`. Using the SIDS-compliant `I8` type ensures that generic tree-walking utilities (`cgnscheck`, `cg_array_info`, custom scripts) can query the attribute without encountering an undefined data type. Although `I8` is signed, bit-level operations are well-defined in C for non-negative values, and the implementation should cast to `uint64_t` internally for bitwise manipulation. Each bit position is assigned via an explicit constant, not by array order:

```
#define CGNS_FEATURE_BIT_EXTENDED_ELEMENTS     0
#define CGNS_FEATURE_BIT_REORDERED_ELEMENTS    1
#define CGNS_FEATURE_BIT_NGON_V32              2
#define CGNS_FEATURE_BIT_ELEMENT_START_OFFSET  3
#define CGNS_FEATURE_BIT_PARTICLE_ZONE         4
#define CGNS_FEATURE_BIT_PARTICLE_COORDS       5
#define CGNS_FEATURE_BIT_PARTICLE_SOLUTION     6
#define CGNS_FEATURE_BIT_ELEMENT_INTERP        7
#define CGNS_FEATURE_BIT_SOLUTION_INTERP       8
#define CGNS_FEATURE_BIT_HIGH_ORDER_ELEMENTS   9
#define CGNS_FEATURE_BIT_UNKNOWN_MODERN       10
```

A set bit indicates the corresponding feature is present in the file. Bit assignments are permanent:

once assigned to a feature, a bit position must never be reused, even if the feature is deprecated. Using explicit constants (rather than implicit array ordering) ensures that source-code refactoring of the registry table cannot silently change bit assignments and break backward compatibility.

Using a 64-bit type provides 64 feature slots. With the current registry at 11 entries and CGNS adding roughly one to two version-breaking features per major release, this provides ample runway without the overhead of a variable-length representation. If the registry ever approaches 64 entries, a follow-on CPEX can introduce `_CGNS_FeatureMask2` for the overflow bits, maintaining backward compatibility.

Future CPEXs that introduce version-breaking features must allocate the next available bit position and add the corresponding `CGNS_FEATURE_BIT_*` constant. Retired entries should be marked as reserved in the source.

3.8.2 Open-time use

On `cg_open()`, compatibility checking proceeds as follows:

1. `CGNSMinRequiredVersion_t` **present** (Alternative B, CPEX-0049 file): read the node value in $O(1)$ and compare against the configured compatibility version. If the file is rejected and `_CGNS_FeatureMask` is also present, use the bitmask to identify the specific incompatible feature for the error message. No tree walk is performed in either case.
2. `_CGNS_FeatureMask` **only** (Alternative A, CPEX-0049 file): read the bitmask in $O(1)$ and determine the minimum required version by finding the highest-version bit that is set. No tree walk.
3. **Neither present** (legacy file): fall back to the $O(N)$ tree walk. This ensures full backward compatibility with all pre-CPEX-0049 files.

Performance of the legacy fallback. The $O(N)$ tree walk only runs for legacy files *when the application has set restrictive compatibility version* (i.e., `compat_version` < `CG_LIBVER_LATEST`). Under default settings—which is the common case—no feature scan is performed at all, even for legacy files. Furthermore, the scan walks in-memory data structures that `cgi_read()` has already parsed; it does not issue additional I/O. The short-circuit checker (`cgi_check_version_limit`) stops on the first incompatible feature, so the full $O(N)$ cost is only incurred when the file is *compatible* and every detector must confirm absence of violations. Finally, opening a legacy file in `CG_MODE_MODIFY` triggers a one-time scan whose result is written to the file at close (see "Legacy file healing" below), converting all subsequent opens to $O(1)$.

3.8.3 Maintenance

The mask is maintained in memory throughout the file's open lifespan via a `int64_t current_feature_mask` field added to the internal `cgns_file` struct. This makes `cg_close()` an $O(1)$ operation—it simply flushes the in-memory value to the `_CGNS_FeatureMask` attribute.

The lifecycle is:

1. **Open (CPEX-0049 file):** read `_CGNS_FeatureMask` into `current_feature_mask` in $O(1)$.
2. **Open (legacy file):** perform the $O(N)$ tree walk once to populate `current_feature_mask`.

3. **Write path:** whenever `cgi_require_version()` is called for a version-specific feature, simultaneously bitwise-OR the corresponding registry bit into `current_feature_mask`. No additional work is required.

4. **Close:** flush `current_feature_mask` to `_CGNS_FeatureMask` in $O(1)$. Skipped for `CG_MODE_READ` opens.

Crash and abort safety. Because the feature mask is held in memory and only flushed at `cg_close()`, an application crash or `MPI_Abort()` mid-write will leave the file without an updated `_CGNS_FeatureMask`. This is by design: the file is likely truncated or inconsistent in that case, and if it is recoverable, the absent or stale mask simply causes the next reader to fall back to the $O(N)$ tree walk—failing safely rather than silently. No data corruption or false acceptance results from a missing mask.

Legacy file healing. When a legacy file (no mask attribute) is opened in `CG_MODE_MODIFY`, the $O(N)$ scan runs once at open time to populate the in-memory mask. On close, the mask is written to the file. All *subsequent* opens of that file by any CPEX-0049-aware library will find the attribute and skip the scan entirely. This automatic upgrade is a useful side-effect: modifying any node in a legacy file, even a trivial coordinate update, permanently converts it to $O(1)$ compatibility checking for future readers without any explicit user action.

3.9 SIDS Impact

The impact depends on the file-format alternative selected:

Alternative A. The SIDS definition of `CGNSLibraryVersion_t` must be amended from "the version of the library that created the file" to "the minimum library version required to read the file." The existing node structure is unchanged.

Alternative B (recommended). One new node type, `CGNSMinRequiredVersion_t`, is added to the SIDS as a child of the root (same level as `CGNSLibraryVersion_t`). The definition of `CGNSLibraryVersion_t` is **unchanged**.

> **SIDS amendment required.** The SIDS root-node schema (Section 2 of the SIDS document) strictly defines the allowed children. This addition requires a **formal amendment to SIDS Section 2** adding `CGNSMinRequiredVersion_t` to the root-level node list. A corresponding file-mapping update (Section 3.10) must also be published.

Under both alternatives, the `_CGNS_FeatureMask` attribute must be documented in the SIDS file mapping as a hidden `I8` attribute.

3.10 File Mapping Impact

Alternative A. One hidden attribute (`_CGNS_FeatureMask`) is added to the existing `CGNSLibraryVersion_t` node. No new nodes.

Alternative B (recommended). One new node (`CGNSMinRequiredVersion_t`) is added at the root level. The `_CGNS_FeatureMask` attribute is placed on this new node. `CGNSLibraryVersion_t` is unchanged.

Under both alternatives, implementations that do not recognize the new node or attribute must silently ignore them.

4 Backward Compatibility

- **Existing files:** No change. Files written by any prior CGNS version remain readable.
- **Existing applications (3-argument `cg_open`):** The 3-argument `cg_open` call continues to work unchanged. It now expands to `cg_open_with_params(file, mode, CG_PARAMS_DEFAULT, &fn)`, using global defaults.
- **Default bounds** (`low = CG_LIBVER_AUTO, high = CG_LIBVER_LATEST`): Applications that do not call the new API functions inherit these defaults, which reproduce the current behavior—but with a feature-aware check instead of a version-number-only check. The sole behavioral change for existing code is that files from a *newer* library version that use no incompatible features will now be accepted instead of rejected.
- **File format additions** (see Section 3.2): Under Alternative A, the SIDS definition of `CGNSLibraryVersion_t` is amended and one hidden attribute is added. Under Alternative B (recommended), one new node (`CGNSMinRequiredVersion_t`) is added and `CGNSLibraryVersion_t` is unchanged. In both cases, old libraries silently ignore what they do not recognize.

5 Fortran Bindings

The following Fortran subroutines are provided in the `cgns` module (`src/cgns_f.F90`):

```
subroutine cg_set_libver_bounds_f(fn, low, high, ier)
    integer, intent(in)  :: fn, low, high
    integer, intent(out) :: ier

! Query bounds and/or minimum version.
! min_version is OPTIONAL: omitting it skips the O(N) feature scan.
subroutine cg_get_libver_bounds_f(fn, low, high, &
                                  min_version, ier)
    use iso_c_binding, only: C_NULL_PTR, C_LOC
    integer,           intent(in)            :: fn
    integer,           intent(out)           :: low, high, ier
    integer,           intent(out), optional :: min_version

! Parameter object open (4-argument form)
subroutine cg_open_params_f(filename, mode, params, fn, ier)
    character(*),      intent(in)  :: filename
    integer,           intent(in)  :: mode
    type(C_PTR),       intent(in)  :: params
    integer,           intent(out) :: fn, ier
```

The `cg_open` interface in the `cgns` module is overloaded to accept either 3 arguments (`cg_open_f`, legacy) or 4 arguments (`cg_open_params_f`, with params).

For parallel applications, `cgp_open_params_f` is provided with the same signature as `cg_open_params_f`.

Note on `min_version` cost control. Fortran 2003 `OPTIONAL` dummy arguments provide the same NULL-pointer gating as the C API. The wrapper implementation checks `PRESENT(min_version)`: if absent, it passes `C_NULL_PTR` to the C function, skipping the $O(N)$ feature scan entirely; if present, it passes `C_LOC(min_version)`, triggering the scan. This mirrors the C semantics without requiring a separate entry point.

The `CG_LIBVER_*` integer constants are exported from the `cgns` module and may be used directly in Fortran code.

6 Examples

6.1 Example 1: Default Behavior (No API Call)

An application compiled against CGNS 4.4 opens a file written by CGNS 5.0 that contains only structured zones, coordinates, and flow solutions—all features present since CGNS 2.x. Under the current library this fails; under this CPEX it succeeds (with an optional warning).

```
/* No compatibility version call -- defaults apply */
int fn;
if (cg_open("output_from_v5.cgns", CG_MODE_READ, &fn)
        != CG_OK) {
    /* Under current library: fails with version error.
       Under this CPEX: succeeds because no v5-only
       features are present. */
    cg_error_print();
}
/* ... read data as usual ... */
cg_close(fn);
```

6.2 Example 2: Restricting to a Known-Compatible Range

A production solver sets the version bounds so that written files are compatible with a certified post-processing tool chain that only supports up to CGNS 4.0:

```
/* Lock version bounds globally:
   low  = AUTO (write minimum needed),
   high = V40  (reject features beyond 4.0) */
cg_configure(CG_CONFIG_LIBVER_LOW,  (void *)(intptr_t)CG_LIBVER_AUTO);
cg_configure(CG_CONFIG_LIBVER_HIGH, (void *)(intptr_t)CG_LIBVER_V40);

int fn;
cg_open("solver_output.cgns", CG_MODE_WRITE, &fn);
/* AUTO write mode records the minimum version needed,
   capped by high (4000). */
/* ... write grid, solution, BCs ... */
cg_close(fn);
```

6.3 Example 3: Thread-Safe Per-File Configuration

Different threads open files with different version requirements:

```
cg_parameters_t params;
cg_params_create(&params);
cg_params_set(params, CG_PARAM_LIBVER_LOW,
              (void *)(intptr_t)CG_LIBVER_V40);
cg_params_set(params, CG_PARAM_LIBVER_HIGH,
              (void *)(intptr_t)CG_LIBVER_LATEST);

int fn;
cg_open("myfile.cgns", CG_MODE_WRITE, params, &fn);
/* ... use file ... */
cg_close(fn);
cg_params_destroy(params);
```

6.4 Example 4: Analyzing Minimum Required Version

After writing a file, query the minimum version required by the features actually present:

```c
int fn, min_ver;
cg_open("output.cgns", CG_MODE_READ, &fn);
/* Pass NULL for low and high; only compute min_version */
cg_get_libver_bounds(fn, NULL, NULL, &min_ver);
printf("File requires CGNS >= %d.%d\n",
       min_ver / 1000, (min_ver % 1000) / 10);
cg_close(fn);
```

6.5 Example 5: Using cg_configure

```c
#include <stdint.h>

/* Set bounds: AUTO lower, CGNS 4.0 upper */
cg_configure(CG_CONFIG_LIBVER_LOW,
             (void *)(intptr_t)CG_LIBVER_AUTO);
cg_configure(CG_CONFIG_LIBVER_HIGH,
             (void *)(intptr_t)CG_LIBVER_V40);

int fn;
cg_open("constrained.cgns", CG_MODE_WRITE, &fn);
/* ... */
cg_close(fn);
```

6.6 Example 6: Fortran Usage

```fortran
program libver_bounds_example
  use cgns
  implicit none
  integer :: ier, fn, lo, hi, min_ver

  ! Set global bounds: AUTO lower, CGNS 4.0 upper
  call cg_configure_f(CG_CONFIG_LIBVER_LOW,  CG_LIBVER_AUTO, ier)
  if (ier .ne. CG_OK) stop 'Error setting lower bound'
  call cg_configure_f(CG_CONFIG_LIBVER_HIGH, CG_LIBVER_V40,  ier)
  if (ier .ne. CG_OK) stop 'Error setting upper bound'

  call cg_open_f('test.cgns', CG_MODE_READ, fn, ier)
  if (ier .ne. CG_OK) then
     call cg_error_print_f()
     stop 'Open failed'
  end if

  call cg_get_libver_bounds_f(fn, lo, hi, min_ver, ier)
  write(*,*) 'Min version required:', min_ver

  call cg_close_f(fn, ier)
end program
```

6.7 Example 7: Detecting Incompatible Features

A CGNS 4.4 library opens a file written by CGNS 5.0 that contains a `ParticleZone_t` node (introduced in CGNS 4.5 via CPEX 0046). The feature-compatibility scan detects this node and returns an error:

```c
int fn;
if (cg_open("particles_v5.cgns", CG_MODE_READ, &fn)
        != CG_OK) {
    /* Error message indicates which feature requires
       a newer library. */
    printf("%s\n", cg_get_error());
}
```

7 Reference Implementation

A working implementation is available on the feature branch:

https://github.com/brtnfld/CGNS/tree/915

Key changes in the reference implementation:

- `src/cgnslib.h` — Defines the `CG_LIBVER_*` integer constants, `CG_CONFIG_COMPAT_LOW` through `CG_CONFIG_GET_COMPAT_HIGH` tokens, `CG_PARAM_KEY` enum, `cg_parameters_t` type, and all new API prototypes.
- `src/cgnslib.c` — Implements `cg_set_compat_version`, `cg_set_file_compat_version`, `cg_get_file_compat_version`, `cg_params_create/destroy/set`, `cg_open_with_params`, and the revised open/write logic.
- `src/cgns_internals.c` — Feature-version registry with detector functions; `cgi_require_version` for AUTO write-version tracking; `cgi_check_version_limit` for fast-path validation.
- `src/cgns_f.F90` — Fortran module wrappers for all new functions and the overloaded `cg_open` interface.
- `src/tests/test_version_bounds.c` — Eight test cases covering AUTO write mode, explicit version locking, compatibility enforcement on read, compatibility inheritance by MODIFY mode, feature-mismatch error reporting, and per-file compatibility versions.
- `src/tests/test_parameters.c` — Seven test cases for the parameter object lifecycle and the polymorphic `cg_open` macro.

8 Edge Cases and Limitations

8.1 External Links

CGNS supports linked nodes that reference data in external files. Compatibility checking for external links works correctly without user intervention, but the underlying mechanism exposes a pre-existing architectural concern that deserves acknowledgment.

Current behavior. Both the HDF5 (ADFH) and ADF backends resolve external links transparently at the I/O layer. More critically, `cgi_read()` — which is called *before* the feature scan during `cg_open()` — reads the entire file tree eagerly, following all external links and opening all linked files at startup. The feature scan therefore walks already-populated structs; it adds no additional file opens beyond what `cgi_read()` already performed. External link content is fully visible to all detectors.

The HPC I/O storm concern. On parallel file systems such as Lustre or GPFS, opening and parsing dozens of external linked files during `cg_open()` of a master file can cause severe metadata latency—an "I/O storm." This is a pre-existing concern with CGNS's eager `cgi_read()` architecture and is not introduced by this CPEX. However, the addition of compatibility checking makes it more important to resolve, since applications may now call `cg_open()` with restrictive compatibility versions specifically to validate linked content, amplifying the existing performance problem.

Three architectural strategies for future resolution. The following strategies address both the I/O storm and version checking for external links. They are presented as candidates for a follow-on CPEX or as input to an architectural review; they are not required for the initial implementation of CPEX 0049.

Strategy 1: Deferred just-in-time validation (recommended). Make `cgi_read()` lazy: defer reading a linked node until the application first traverses into it (e.g., via `cg_goto`). The compatibility check for that node is performed at the same moment, deep in the traversal path rather than at startup. This eliminates the I/O storm entirely and gives the application a `CG_ERROR` at the exact point of incompatibility. The trade-off is that error detection moves from `cg_open()` to the first traversal call, requiring consistent `CG_ERROR` checking throughout read loops. This strategy requires significant refactoring of the CGNS read path and is a larger effort than CPEX 0049 alone.

Strategy 2: Embedded link version metadata. When `cg_link_write()` creates a link to an external file, the library briefly opens the target, computes its minimum required version, and stores the result as a hidden attribute (e.g., `_CGNS_LinkMinVersion`) on the link node in the parent file. Subsequent `cg_open()` calls can read this $O(1)$ attribute without opening the external file. The limitation is staleness: if the external file is modified after the link is created, the cached attribute becomes incorrect. This strategy requires a file-format change (new hidden attribute) and a corresponding SIDS/file mapping update.

Strategy 3: Write-time cascading version inheritance. When `cg_link_write()` creates a link in AUTO write-version mode, the library opens the target file, calls `cgi_require_version()` with the linked file's minimum version, and thereby bumps the parent file's effective version to cover the union of both files' feature requirements. A downstream reader opening the parent file will then see a version header that reflects the highest-version feature reachable via any link, and can reject the file immediately based on that header alone without opening any linked file. This strategy is the most compatible with the current CPEX 0049 design and requires no structural changes to `cg_open()`.

Which strategy is best? The answer depends on which problem is being solved. A precise comparison requires distinguishing two cases: a *reject* (the version check fails and the file is refused) and an *accept* (the check passes and the file is opened normally).

	Strategy 1	Strategy 2	Strategy 3
Eliminates I/O storm on reject	✓	✓	✓
Eliminates I/O storm on accept	✓	×	×
Requires file format change	×	✓	×
Staleness risk	×	✓	✓
Scope compatible with CPEX 0049	×	×	✓

A critical observation: Strategies 2 and 3 only improve the *reject* path. In the *accept* path, `cgi_read()` still runs and still opens every linked file regardless of which strategy is in place. The I/O storm in the accept case is eliminated *only* by Strategy 1 (lazy `cgi_read()`).

Strategy 2 is strictly inferior to Strategy 3 within the scope of this CPEX: it requires a file-format change and a SIDS update, carries the same staleness risk, provides the same fast-reject benefit, and adds no advantage in the accept case.

Strategy 1 is architecturally the correct long-term solution—it is the only strategy that eliminates the I/O storm in both the accept and reject paths—but it requires refactoring the entire CGNS read path into a lazy model, which is a substantially larger effort than this CPEX and warrants its own proposal.

Recommendation: Implement Strategy 3 in this CPEX as the lowest-risk, highest-compatibility improvement. File a separate architectural proposal for Strategy 1 (lazy `cgi_read()`) as a follow-on effort that would benefit all CGNS users on HPC file systems, not only those using compatibility version checking.

Out-of-band update risk (Strategy 3). Strategy 3 is vulnerable to external file modifications that occur *after* the link is established. If File A links to File B when both require CGNS 3.0, and File B is later modified out-of-band to include a `ParticleZone_t` node (requiring CGNS 4.5), File A's version header is not updated automatically. A CGNS 3.0 reader opening File A will pass the header check and then encounter an incompatible node upon traversing the link, producing a delayed error instead of a clean rejection at open time. This limitation is inherent to any write-time caching approach (including Strategy 2) and must be documented clearly in the MLL reference manual. Users who modify external files after linking should reopen the parent file in `CG_MODE_MODIFY` and close it immediately; this recomputes the version header and `_CGNS_FeatureMask` attribute. Under Alternative B this workaround is straightforward: only `CGNSMinRequiredVersion_t` is updated, while `CGNSLibraryVersion_t` simply records the current library version—provenance is preserved throughout.

8.2 Version Downgrades on Node Deletion

The AUTO write-version mechanism is strictly *incremental*: the effective version only increases as version-specific features are written. If a user opens a file in `CG_MODE_MODIFY`, deletes the only node that required a high version (e.g., removes the last `ParticleZone_t`), and closes the file, the minimum version recorded in the file is *not* downgraded. The file will continue to declare the higher version even though the feature is no longer present.

A full downgrade would require re-scanning the entire file on every close, which is prohibitively expensive. The current behavior is therefore conservative: the version header may over-state the minimum required version after deletions, but it will never under-state it.

Impact of Alternative B. Under Alternative B, this limitation is less severe. The over-stated version resides in `CGNSMinRequiredVersion_t`, while `CGNSLibraryVersion_t` continues to report the true writing library version. Provenance is preserved, and tools that rely on "which library wrote this file?" remain correct. The only consequence is that an older reader that *could* read the file (because the deleted feature is gone) will be incorrectly told it cannot—a conservative false rejection, not a silent misinterpretation. Under Alternative A, the same over-statement would also lose provenance information, compounding the problem.

8.3 Partial Parallel Writes and Version Agreement

In an MPI-IO scenario where different ranks write different features concurrently, the final library version recorded in the file must cover the union of all features written across all ranks.

AUTO mode (sync-at-close). When parallel AUTO is used, each rank tracks its effective version locally as features are written. At `cgp_close` the library performs an `MPI_Allreduce(MPI_MAX)` to compute the collective maximum and writes that value to the version node. No application-level coordination is required—the library handles version agreement transparently.

Explicit `low` bound. When an explicit lower bound is configured, all ranks must set the same value before calling `cgp_open`. If any rank attempts to write a feature whose minimum version exceeds the configured `high` bound, the library returns `CG_ERROR` on that rank. This error is rank-local; other ranks are not automatically notified. Applications must treat a per-rank `CG_ERROR` from a write call as a collective failure requiring abort or coordinated recovery.

9 Open Questions

1. **Alternative A vs. Alternative B.** This proposal recommends Alternative B (new `CGNSMinRequiredVersion_t` node) over Alternative A (overload `CGNSLibraryVersion_t`). The CGNS committee must decide which alternative to adopt before the implementation is merged.

   Arguments for Alternative A: no new SIDS nodes, no file-mapping additions beyond one hidden attribute, maximum structural simplicity.

   Arguments for Alternative B: preserves provenance (`CGNSLibraryVersion_t` continues to record the writing library version), no semantic redefinition of an established SIDS node, and cleaner MODIFY-mode logic.

   See Section 3.2 for the full comparison.

2. **Alternative B node placement: root sibling vs. child of `CGNSLibraryVersion_t`.** If Alternative B is adopted, the committee must decide whether `CGNSMinRequiredVersion_t` is placed at the file root (sibling of `CGNSLibraryVersion_t`) or as a child of `CGNSLibraryVersion_t`.

   Arguments for child placement: all version metadata co-located under one existing node; root namespace does not grow.

   Arguments for root sibling (recommended): clean semantic separation between "written by" and "requires at least"; `CGNSLibraryVersion_t` remains structurally unchanged, preserving Alt. B's core advantage; consistent with existing root-level CGNS metadata nodes.

   See Section 3.2.4 for the full comparison.

3. **Polymorphic `cg_open`: variadic macro vs. explicit 4-argument API.** The proposal currently offers both: a polymorphic `cg_open` macro (C99 variadic argument counting) that dispatches to either the 3-argument legacy form or the 4-argument `cg_open_with_params`, alongside `cg_open_with_params` as a stable explicit entry point.

   Arguments for the variadic macro: transparent to existing code—no source changes required for the 3-argument case.

Arguments against the variadic macro: C99 `__VA_ARGS__` counting is not reliable on all vendor preprocessors; the macro can be suppressed by `#define CGNS_NO_MACROS` if it causes problems.

Arguments for dropping the macro entirely: simpler specification and ABI; users call `cg_open_with_params` explicitly for the 4-argument form and `cg_open` remains the unchanged 3-argument function. Existing code requires no changes since `cg_open` is untouched.

The committee should decide whether the polymorphic macro is worth the preprocessor portability risk, or whether `cg_open_with_params` alone is sufficient.

10 Summary of API Additions

Symbol	Description
`CG_LIBVER_EARLIEST`	Oldest CGNS library version (1050).
`CG_LIBVER_LATEST`	Current library version (5000).
`CG_LIBVER_AUTO`	Sentinel: automatic write-version selection.
`CG_LIBVER_V30`...`CG_LIBVER_V50`	Specific version milestone constants.
`CG_CONFIG_LIBVER_LOW`	`cg_configure` token: set global lower bound.
`CG_CONFIG_LIBVER_HIGH`	`cg_configure` token: set global upper bound.
`CG_CONFIG_GET_LIBVER_LOW`	`cg_configure` token: query global lower bound.
`CG_CONFIG_GET_LIBVER_HIGH`	`cg_configure` token: query global upper bound.
`CGNS_FEATURE_BIT_*`	Explicit bit-position constants for `_CGNS_FeatureMask` (0–10).
`CG_PARAM_KEY`	Enum of parameter keys (100–103).
`cg_parameters_t`	Opaque parameter object type.
`CG_PARAMS_DEFAULT`	NULL sentinel: use global configuration.
`cg_set_libver_bounds`	Set per-file version bounds (`fn`, `low`, `high`).
`cg_get_libver_bounds`	Query per-file bounds and/or minimum required version. Any output pointer may be `NULL`; the $O(N)$ feature scan runs only when `min_version` is non-`NULL`.
`cg_params_create`	Allocate parameter object with defaults.
`cg_params_destroy`	Free parameter object.
`cg_params_set`	Set a parameter by key/value.
`cg_open_with_params`	Open file with explicit parameter object (standard name).
`CGNS_NO_MACROS`	Preprocessor guard: when defined, suppresses the polymorphic `cg_open` macro so `cg_open` resolves to the real 3-argument function.
`cg_open` (macro)	Polymorphic: 3-arg (legacy) or 4-arg (with params); suppressed by `#define CGNS_NO_MACROS`.
`CGNSMinRequiredVersion_t`	*Alt. B:* new SIDS node recording the minimum CGNS version required to read the file. Preserves `CGNSLibraryVersion_t` semantics.

Symbol	Description
_CGNS_FeatureMask	Hidden I8 attribute (HDF5: H5T_NATIVE_INT64; ADF: I8): bitmask of features present; enables $O(1)$ version check on open. Placed on CGNSMinRequiredVersion_t (Alt. B) or CGNSLibraryVersion_t (Alt. A).
cg_set_libver_bounds_f	Fortran: set per-file version bounds (fn, low, high).
cg_get_libver_bounds_f	Fortran: query per-file bounds and (optionally) minimum required version. min_version is an OPTIONAL dummy argument (F2003); when absent, C_NULL_PTR is passed to the C layer, skipping the $O(N)$ scan.
cg_open_params_f	Fortran: 4-argument open with params.
cgp_open_params_f	Fortran (parallel): 4-argument open with params.

11 References

1. CGNS GitHub Issue #915, "Add CGNS version bounds control," https://github.com/CGNS/CGNS/issues/915
2. CGNS GitHub Issue #670, "Buffer overflow in cgi_error using v3.31 thru v4.2" (related version-check code), https://github.com/CGNS/CGNS/issues/670
3. CGNS Confluence, "Resolve issue with release's 3.4.0 version compatibility, the 4.0.0 release, and forward compatibility," https://cgnsorg.atlassian.net/wiki/spaces/CGNS/pages/220463122/
4. HDF5 Reference Manual, H5Pset_libver_bounds, https://docs.hdfgroup.org/hdf5/v1_14/group___F_A_P_L.html
5. CGNS Standard Interface Data Structures (SIDS), https://cgns.org/standard/SIDS/CGNS_SIDS.html
6. CGNS Mid-Level Library—File Operations, https://cgns.org/standard/MLL/api/c_api.html