Census of H5CX Callbacks

Matthew Larson

June 26, 2024

Contents

1 Introduction

This work-in-progress document is to serve as an overview of and documentation for the H5CX module of the HDF5 library as of the develop branch at time of writing (June 2024, with the most recent release being 1.14.4.3), with a particular focus on identifying potential issues with the implementation of threadsafety in this module.

2 Overview

An API context is effectively a collection of global variables within an API call. API contexts provide a way to get and set values from different library layers during a single API routine.

Each API routine creates an API context object upon entry - an instance of `H5CX_t`. See section C.1 for a full description of this structure. Fields on this API context control aspects of the operation, deciding things such as which VOL to use or whether a metadata read is performed collectively. At the end of an API routine, the context is released. The temporary lifetime and accessor-based usage are key parts of the context's design, intended to avoid common issues associated with global variables.

The fields on a context are modified and retrieved with 'get' and 'set' routines uniquely defined for each field. While every field has a getter, not every field has a setter. It appears that setters were only defined for fields where the library needs to use that setter internally.

After the API context is created upon API call entry, internal library routines provide the context with references to the property lists that are controlling the operation through setters of the form `H5CX_set_*pl()`, such as `H5CX_set_dxpl()`. These setters may also be used in lower levels of the library. For example, `H5Lcreate_soft()` does not populate the context with property lists until the internal `H5L__create_soft_api_common()`, where the LCPL and LAPL are set on the context through `H5CX_set_lcpl` and `H5CX_set_apl` respectively.

The context struct stores the ID of the provided property list(s), as well as (potentially) storing pointers to each underlying `H5P_genplist_t` object. The property list is stored directly in order to optimize field reads by skipping H5I lookup operations. In order to avoid costly duplication, the context stores pointers and references to the original property lists without duplicating them.

Modifications to a field in an API context (via setter routines) only modify the underlying property list if the field is Return-only or Return-and-read. Even then, this modification only takes place at the end of the API call for which the context object was created.

When an API function is entered, the `FUNC_ENTER_API(_NOCLEAR)` macro calls `FUNC_ENTER_API_PUSH`, which uses `H5CX_push()` to initialize a new context for the calling thread. A context is initialized with all of its bytes set to zero, implicitly setting all flags to 'false'.

The API context struct has at least one member for each field accessible through the API context interface. The different types of field within the context are:

- **Reference Property Lists** - The property lists to use for this operation. May be default property lists, or non-default lists provided by the application. The underlying property list objects are not copied. Each property list associated with the context is stored as an ID, and potentially as a pointer to the underlying property list object. The fields of this type are `dxpl(_id)` , `lcpl(_id)`, `lapl(_id)`, `dcpl(_id)`, `dapl(_id)` and `fapl(_id)`.

- **Internal Fields** - Fields that don't correspond to a property from a property list. Fields of this type are often set at a high level and used at a low level. For example, the metadata "tag" is set in high-level group/dataset code and used in the low-level metadata cache.

- **Cached Fields** - Read operations on property lists can be quite slow. To mitigate this, contexts have 'cached fields' which are used to short-circuit property reads whenever possible. If a cached field is valid, then it is read from the context and its value returned to the user with no property list operations performed. If a cached field is invalid, then after the value is retrieved from the underlying property list and saved in the context, the cached field is marked as 'valid' to hopefully speed up later operations.

  Changes to cached field values in the context do not propagate to the original property lists.

  Each cached field must have a corresponding `<field_name>_valid` flag in the context structure that indicates whether the context has a saved value. Context macros depend upon the existence of a flag with this name.

  Cached fields always have a 'get' routine, but may not have a 'set' routine if they are never set by the library.

  Most fields on the API context are of this type, with most of those fields corresponding to properties from a DXPL.

- **Return-only Fields** - Return-only fields correspond to properties from property lists that are never read within the library and are only set with values in order to return those values to the application. When a value is set in a return-only field, that value is stored in the context until the context is destroyed at API return time. At that time, any changed return-only fields have their new values written into the provided property list(s).

  Each return-only field must have a corresponding `<field_name>_set` flag in the context structure that indicates whether the library has populated that field with a value to be written to the property list at API return time. Context macros depend upon the existence of a flag with this name.

- **Return-and-read Fields** - These fields are used to set properties for return to the application, but are also internally queried by the library.

  These fields must have both a `<field_name>_set` flag and a `<field_name>_valid` flag. Context macros depend upon the existence of flags with these names.

  Internal queries to these fields only read from the underlying property list if the context value has never been read or set prior. Setting a value on the context thus modifies the value retrieved from subsequent reads, and over the course of a context's lifetime, there will be at most one read from a property list of a return-and-read field (since later reads will always use the value stored in the context).

  Note that it is possible to overwrite the cached value via an `H5CX_set_*()` operation, but not possible to overwrite the cached value with a `H5CX_get_*()` operation.

  These fields behave in potentially unintuitive ways:

  – The `<field>_set` flag only indicates whether the context has ever performed a set operation on this field.

  – The `<field>_valid` flag only indicates whether the context has ever been populated with a value from the original property list. The `H5CX_set_<field>()` routines don't interact with this flag at all, and so both of the following are possible:

    ∗ The field is never read, and is only written to by `H5CX_set_<field>()`. The context then contains a meaningful value, but `<field>_valid` is false.

    ∗ The field is read from and populated on the context, but is later overwritten on the context by `H5CX_set_<field>()`. The context will then contain a value that is not from the underlying property list, but `<field>_valid` is true.

2.1 Default Values

During the initialization of the context interface, the property values from the default property lists are copied into module-local structures with names of the form `H5CX_def_*_cache`. When the context is queried for a field whose corresponding property list is default (and the context has no valid value to provide), the value is retrieved from these structures.

Note that if the default property list for a class is modified by the application after library initialization, that change will not be reflected in the cache structures, and so the context will continue to provide the original default value.

2.2 API Context State

Some VOL connectors may not perform operations requested through the API in the order that they were requested. One such example is the async VOL, which executes tasks at a later time after requests are submitted via the API. However, it is still necessary that these operations take place with the API context (e.g. property lists) that the user provided.

'API Context State' was introduced in order to allow a set of context information to be stored and retrieved at a later time. An API context state is represented by `H5CX_state_t`. See section C.3 for a full description of this struct.

API context state consists of a set of property lists (DCPL, DXPL, LAPL, LCPL), the VOL connector property, the VOL object wrapper context, and, if parallel HDF5 is enabled, a collective metadata read flag.

Note that the FAPL and the DAPL, despite existing on the context, are not stored in the API context state. This is believed to be because the context state was created before these property lists were added to the context, and not the result of a specific design decision.

2.3 Re-Entrancy and The API Context Stack

Some API routines, such as `H5Literate`, involve user-defined callbacks, which may themselves call API routines. Thus, API contexts must be re-entrant safe. This is handled by storing multiple API contexts on a stack, where the topmost context on the stack is used by the most recent API routine in the call stack.

A single 'node' on the API context stack is represented by the struct `H5CX_node_t` (see section C.2 for a full description). Each node contains a pointer to an `H5CX_t` instance, and a pointer to the next (lower) node in the stack.

Each API call uses the macro `FUNC_ENTER_API(_NOCLEAR)` to initialize and allocate a new context node. This context node is pushed to the top of the context stack by `H5CX_push()`. The end of each API call uses the macro `FUNC_LEAVE_API` to free the context node and remove it from the context stack with `H5CX_pop()`.

There also exists an `H5CX_push_special()` routine, which performs the same work as `H5CX_push()` but without using library memory management routines. This function exists to allow the context stack to be added to when parts of the library are not initialized, such as during library shutdown.

3 Threadsafety

The context module was designed and implemented with the future possibility of threadsafety in mind. For that reason, each thread using the HDF5 library has its own instance of the API context stack. Specifically, each thread has a unique value of `H5TS_apictx_key_g`. This key is used to set and retrieve the context stack and other thread local values (error stack, cancellation counter) via `H5TS_(get/set)_thread_local_value()`.

More specifically, the thread local keys for API contexts and other thread local values are created within an initialization routine (`H5TS_pthread_first_thread_init()` when using POSIX threads, `H5TS_win32_process_enter()` when using win32 threads) to be executed only by the first thread in each process to enter the library.

Because a single invocation of `pthread_key_create()` (or `TlsAlloc()` when using win32 threads) instantiates a unique key value for every thread, this single initialization is all that is needed to set up thread local keys for each thread.

This design eliminates the possibility of inter-thread conflicts for internal fields that exist only on the context. However, inter-thread conflicts still exist for fields that are only shallow-copied to the context, since the underlying shared buffer may be accessed or manipulated by other threads at unpredictable times. The fields which are shallow copied in the context are:

- `vol_connector_prop` - The `connector_info` buffer could be modified at arbitrary times by multiple threads within VOL callbacks.
- `vol_wrap_ctx` - This field is a pointer to a VOL wrapping object which may have shared resources under it, depending on the current VOL connector stack (see section C.7)
- `vl_alloc_info` - This field contains two pointers to external buffers which may be externally modified. Due to the context's function that interact with this field only being invoked from `H5T`, which resides under the global mutex, this is not believed to cause any threadsafety problems.

  Also note that this field's getter uses `H5P_peek()` to avoid copying the underlying value, and so this function inherits the threadsafety problems associated with all property field getters.

Shallow-copied values are not an issue for most property fields, since most fields internally rely on `H5P_get()`, which follows the copy-by-value semantics for property values.

3.1 Library-Internal Modification of Property Lists

Modifications to property lists which may be referenced on a context, when performed from any library module which resides above the global mutex, are potentially not threadsafe. This is because another thread using the same property list might read or write to that same property list concurrently via a context operation. Sharing property lists between threads creates threadsafety problems even without the API Context (see the H5P census for details), but contexts exacerbate the issue.

Outside of threadsafety, internal modification of property lists causes other problems within the context. If a cached field has a value stored on the context, and the underlying property is modified, a context read will return the out-of-date cached value. If the field was not cached before, then the context will return the up-to-date value from the property list.

Fortunately, the library generally tries to avoid modifying property lists internally. For a full listing and description of every instance where the library modifies a property list from a module planned for threadsafety, see the H5P threadsafety census. Most instances of internal property modification are threadsafe for one of the following reasons:

- The property list modification is the result of an API call stated to modify the property list, and so the burden of maintaining threadsafety and avoiding race conditions falls on the application.
- The property list modification is an indirect result of an operation from a module under the global mutex, and so threads will not be executing concurrently.
- The property list modification is performed on a newly created property list that does not yet have a reference in the context, and so cannot be concurrently read from or written to.

3.2 Context Return Fields

Return-only and return-and-read fields create a threadsafety issue when a property list is shared between threads. Each thread's API context will contain a pointer to the same underlying property list, and will attempt to write to the property list upon API routine exit. The final value in the property list then depends on a race condition, and may contain an undefined value if one or more threads' writes interrupt each other.

While return fields are not threadsafe, their context setters (`H5CX_set_<field>()`) are technically threadsafe, since the actual property list modification is delayed until the context is popped from the context stack, and is performed via macro from `H5CX__pop_common()`.

Return-and-read properties are not dependent on internal read patterns, since a set value can never be overwritten afterwards by a query operation.

3.3 Property List Reads

All getter routines that read fields from property lists are inherently unsafe due to the property lists potentially being shared between threads. Consider the scenario where Thread A's read of a property value is interrupted by Thread B writing to that property value. In this case, the final value read by Thread A will be undefined.

This applies to the getters for all Cached Fields and Return-and-Read Fields, since these are the fields which both correspond to a property and are read internally by the library.

`H5CX_retrieve_state()` is affected by this as well, since it must read the properties' values to copy them into the new API state object.

3.4 Property List Setters

The context's property list setters - `H5CX_set_(dxpl/dcpl/lcpl/lapl)()` - are potentially not threadsafe, as the provided property list may be freed after function entry by another thread. This issue only exists for uses of these setters from modules planned for a threadsafe implementation, as otherwise the global mutex should prevent concurrent freeing of the property list.

Modification of the property lists does not create problems for these routines in particular, as only the top-level pointer is manipulated and saved to the context.

3.4.1 H5P_set_apl()

`H5P_set_apl()` has the same problem with its provided property list potentially being freed during operation. Because this function performs introspection on the provided plist with `H5P_peek()`, it also inherits the threadsafety problems caused by direct property list modifications.

3.4.2 H5CX_free_state()

`H5CX_free_state()` uses H5I reference-decrementing routines in a non-threadsafe way on the underlying property lists. If another thread modifies the reference count of a property list concurrently, the list could be assigned an erroneously high reference count and never be freed, causing leaked memory. If the list is assigned an erroneously low reference count, then it may be freed early and leave some references to it pointing to undefined memory.

3.5 VOL Fields

3.5.1 VOL Connector Property

The VOL connector property information structure (`H5VL_connector_prop_t`, see C.5 for a full description) is only shallow copied to the context from the FAPL. This is due to the fact that it is read from the FAPL with `H5P_peek()` to skip the VOL connector property `get` callback, which deep copies it via reference-counting the ID and deep-copying the connector information, as well as the fact that the context's setter only shallowly copies the provided information. As a result, later operations that modify the connector information stored on the FAPL will modify the value on the context as well.

The getter and setter operations for the connector property only manipulate the top-level connector ID (`hid_t`) and the pointer to the connector information buffer. As a result, there is no possibility of the context attempting to read from a malformed buffer during these operations. However, it is possible for another thread to free the connector ID or the connector information after this function is entered. Additionally, the pointer that the context provides may point to a malformed buffer if another thread's modification of the connector information buffer has been interrupted.

3.5.2 VOL Object Wrapping Context

The VOL wrapping context field, `vol_wrap_ctx`, is a pointer to an instance of `H5VL_wrap_ctx_t` (see section C.7 for a full description). The object wrap context stored in this structure, `obj_wrap_ctx`, is defined by each passthrough VOL connector. If it stores references to shared information, then the fact that the wrap context is shallowly copied to the context creates a threadsafety issue. This currently an issue for any passthrough VOL with an object wrap context that references shared memory- currenlty, the the Async VOL and the Log-based VOL.

As with the VOL connector property, the VOL wrap context's setters and getters cannot directly interface with malformed buffers, but may end up providing or setting a context pointer to memory that is invalid as a result of conflict between threads.

3.5.3 VOL Fields in State Functions

Both VOL fields create threadsafety issues within `H5CX_retrieve_state()`.

This routine deep copies the VOL object wrapping context via reference counting, which is unsafe until H5I/H5VL have atomic reference count manipulation.

This routine also deep copies the VOL connector property by reference-counting the underlying connector ID and deep copying the underlying connector information. The reference counting is unsafe until H5I/H5VL have atomic reference count manipulation, and deep copying the information buffer is unsafe since it can be modified concurrently by any thread with access to the FAPL. This also requires the custom VOL connector information copy callback, if any, to be threadsafe.

`H5CX_free_state()` again uses not-currently-threadsafe H5I and H5VL reference-decrementing routines on the VOL wrapper context and the VOL connector property, and requires the VOL connector information free callback, if defined to something besides `free`, to be threadsafe.

3.6 Miscellaneous

Context nodes are allocated through H5FL, so free lists will need to be disabled for the context module to be threadsafe.

4 Points of Interest

These are not currently believed to be threadsafety issues, but they are listed here because they either appear to be potential threadsafety issues on first inspection, or because changes to the library could lead to them becoming threadsafety issues.

4.1 Context State VOL Property Handling

While not specifically a threadsafety issue, there is a potential problem with the API context state's memory management for the VOL connector property information. At state retrieval time (`H5CX_retrieve_state()`), the state allocates a new buffer for the connector information and stores a pointer to it. At state restoration time (`H5CX_restore_state()`), the context stores a pointer to that same buffer. After the state is freed (`H5CX_free_state()`), the context's VOL connector property field will contain a pointer to invalid memory.

4.2 Unchanging Default Property List Values

If the default property list for a class is modified after the context interface is initialized, then the context's default property list values will be from the *original* default property list, not the updated one. This is because the context module copies the default values from default property lists to its own module-local structures during initialization.

4.3 VOL Connector Property Cleanup Ordering

Within `H5CX_free_state()`, during cleanup of the VOL connector property, the connector info is freed before the reference count of the ID is decremented. This initially seems to create a region where another thread could read invalid information, but due to how operations on the connector are implemented in H5VL, this is safe as long as the information free callback properly sets the pointer to NULL.

4.4 Getters with Potentially Shared Return Buffers

Several getter routines which modify the provided arguments to return their value are potentially not threadsafe if the provided buffer is part of an object which is shared between threads. However, these routines are always invoked in a threadsafe way, so this is not currently a problem. Note that many of these functions have unrelated threadsafety issues. The routines that are potentially unsafe with shared return buffers are:

- `H5CX_get_mpi_coll_datatypes()`
- `H5CX_get_btree_split_ratios()`
- `H5CX_get_max_temp_buf()`
- `H5CX_get_tconv_buf()`
- `H5CX_get_bkgr_buf()`
- `H5CX_get_bkgr_buf_type()`
- `H5CX_get_vec_size()`
- `H5CX_get_err_detect()`
- `H5CX_get_filter_cb()`
- `H5CX_get_data_transform()`
- `H5CX_get_vlen_alloc_info()`
- `H5CX_get_modify_write_buf()`

- H5CX_get_mpio_coll_opt()
- H5CX_get_mpio_local_no_coll_cause()
- H5CX_get_mpio_global_no_coll_cause()
- H5CX_get_mpio_chunk_opt_mode()
- H5CX_get_mpio_chunk_opt_num()
- H5CX_get_mpio_chunk_opt_ratio()
- H5CX_get_io_xfer_mode()
- H5CX_get_vol_wrap_ctx()
- H5CX_get_vol_connector_prop()

4.5 Variable-Length Free/Alloc Information

H5CX_get_vlen_alloc_info() retrieves an instance of H5T_vlen_alloc_info_t (see section C.6 for a full description). Two of this struct's values (alloc_info and free_info) are pointers to the original buffers held under the context. Because the corresponding setter does not copy the underlying buffers either, the retrieved pointers point at potentially shared information.

While the context module never acts on the buffers directly, it is possible for the provided buffers to be freed during the set operation, causing the context to store pointers to invalid memory. Another thread could also have a write to one of these buffers interrupted, leading to a context query returning a pointer to invalid memory.

Despite these potential issues (as well as the threadsafe issues that affect every getter for a field that corresponds to a property), this routine is only used from H5T, which will reside under the global mutex. As such, these particular issues are not expected to manifest.

4.6 Miscellaneous

H5CX_set_loc() is threadsafe when parallel HDF5 is enabled, only due to the fact that H5Fmpi is planned to reside under the global lock. In non-parallel HDF5 it is a trivially threadsafe no-op.

5 Proposed New Design

5.1 Property List Handling

Threadsafety issues originating from property list handling will primarily be resolved by changes to H5P. In the threadsafe implementation of H5P, properties will have immutable versions, with modifications to property lists incrementing the version of the list and creating a new immutable property instance. With this new design, the property list that a context is given cannot have its values modified by any other threads, as the context will satisfy all reads from the version of the property list that was current at the time it was provided to the context by `H5CX_set_*pl()`.

Given that the context's property lists are effectively immutable, the idea of removing the property list pointers from the context entirely may arise. Pointers to property lists could be removed if all properties were copied into the context by the `H5CX_set_*pl()` routines. This is a possible implementation, but it would have to read from every single provided property value at context setup time. Storing pointers to the property lists allows the context to entirely avoid costly property lookups for properties which the context never reads.

5.2 Internal Property List Modification

If a cached field has a value stored on the context and the underlying property is modified, then a context read will return the out-of-date cached value. If the value was not previously cached on the context, then the context read will return the up-to-date value. This is inconsistent behavior which should be remedied in this re-design, even though it is not strictly related to threadsafety.

This problem is automatically solved as a consequence of the new H5P design. Contexts will now consistently never return any values that were set on the property list after the list was provided to the context, due to the context always referencing a fixed version. This is a change in the behavior of the context interface, but since the interface is not publicly exposed, it will not force a major version change for the HDF5 library.

5.3 Context Return Fields

At present, past versions of properties in property lists are not planned to be exposed to the application. As such, because modifying a value in a property list populates what is at that time the most recent version of that property, context return fields will generally work the same as they do currently.

There still remains the possibility that a property list provided to multiple threads will have a return property value dependent on the order in which the threads execute, since the thread that executes latest will decide the 'latest property version' on the property list. However, there will no longer be any potential memory safety issues, and so this is considered a minor issue best addressed by applications.

5.3.1 Return-and-Read Fields

Some changes to return-and-read properties would simplify them without changing the behavior of the interface. Specifically, minor changes to flag handling can make it so that `<field>_valid` represents whether the context currently contains a cached value of any origin. There does not seem to be any reason to preserve the current meaning of the valid flag, which only tracks whether a cached value has originated from a read. All that needs to change for this new meaning is for places in the context code that handle library-internal field sets to also modify the `<field>_valid` flag to true. (Because `<field>_set()` prevents later reads from modifying the cached value, that flag already represents whether the context currently contains a value that should be written to a property list.)

5.3.2 Instrumentation Fields

Similarly to return-and-read fields, the context rework provides an opportunity to simplify treatment of these fields. The library configuration that determines whether instrumentation fields are used at all (`H5_HAVE_INSTRUMENTED_LIBRARY`) should also determine whether the properties themselves are defined. As it stands, currently the application must both build HDF5 with instrumentation (and parallelism) enabled, *and* directly register the properties of the given names in order for the introspection information to be provided. There does not seem to be any particular reason to allow a scenario where instrumentation fields are tracked internally on the context but never returned to the application due to the nonexistence of the properties.

5.4 VOL Field Handling

At present, the VOL connector property and the VOL wrapping context fields both present the same kinds of threadsafety problem: they contain underlying buffer(s) which may be modified by other threads, and so the pointers provided by the context interface may point to invalid memory.

5.4.1 The VOL Connector Property

The possibility of the VOL connector property being modified or freed out from under H5CX is not resolved by the H5P changes in isolation, since the context treats the VOL connector property as an independent field with no relationship to the property on the underlying FAPL, despite the fact that they share underlying information. Specifically, they share an `H5VL_class_t` and an information buffer. The shared class should be fine (see section C.5 and C.8), but the shared information is problematic.

Once the H5P redesign has taken place, reworking the `vol_connector_prop` field to be a wrapper around the `H5F_ACS_VOL_CONN_NAME` property on the context's FAPL (in the same manner as most cached fields in the context) should cause all operations on that property to become threadsafe.

The specific changes necessary are:

- The context's FAPL must be added to the API context state struct `H5CX_state_t`. This requires modifying `H5CX_retrieve_state()`, `H5CX_restore_state()`, and `H5CX_free_state()` to set, restore, and free the FAPL id, respectively.
- `H5CX_retrieve_state()`, `H5CX_restore_state()`, and `H5CX_free_state()` should have their special handling of the VOL connector property removed, since it will be covered under the property list copy operations.
- `H5CX_get_vol_connector_prop()` must be changed to pull its value from the underlying FAPL if the valid flag is true. Note that this cannot use the same generic retrieval macros as other cached field without modifying behavior - `H5CX_RETRIEVE_PROP_VALID` uses `H5P_get()`, which invokes the property copy callback if it is defined. This would cause context operations to deep copy the VOL connector property instead of shallow copying it. To avoid changing the context interface, the getter will need to use `H5P_peek()` to shallow copy the value and skip property callback invocation.

  Note that the plist copy operations associated with API context state manipulation *do* invoke the deep copy property callbacks, but this is not a change in behavior since those routines already manually deep copy the VOL connector property.
- `H5CX_set_vol_connector_prop()` - No changes should be necessary, since setters don't interact with the underlying property list for their field.

The only section of this property which may be freely modified by VOL connectors is the connector information buffer (See section C.5 for the details of the VOL connector property and its usage within the library.) Since the connector information exists only on the FAPL, property versioning would prevent any major threadsafety issues. It would still be the case that simultaneous writes to the property would result in a final value dependent on how H5P orders the operations, but this is something that must be handled by the VOL connector itself.

This set of changes will result in only one minor differences in the context interface:
`H5CX_get_vol_connector_prop()` will default to returning the property from the context's provided FAPL, if any, instead of a buffer of zeros.

5.4.2 The VOL Wrap Context

The VOL Wrap Context has threadsafety problems only for passthrough VOLs which define a wrap context containing references to shared information.

Potential solutions to this issue are:

- If multi-threading is primarily intended for use with a bespoke VOL connector, then compatibility with already existing passthrough VOL connectors may not be within this scope of the project at all, and nothing needs to be done, other than perhaps an addition to library config that allows multi-threading within VOL callbacks to be enabled for VOL connectors that specifically support it.
- Resolve this problem at the level of the individual VOL connector behavior - update the Async VOL and the Log-based VOL so that their wrap contexts no longer reference potentially shared information. Pending a deeper investigation into each of these VOL connectors specifically, the scope and difficulty of this option is unknown but likely nontrivial.

5.5 API State

5.5.1 References to Shared Objects

`H5CX_free_state()` releases property list resources allocated during state retrieval by decrementing property list reference counts through `H5I`. Atomic reference count manipulation through `H5I` is necessary for this to be threadsafe.

`H5VL` routines are used to decrement the reference count of the VOL wrapping object. Because the VOL wrapper was deep copied to the API context state, this is threadsafe as long as the application does not attempt to close the VOL wrapping object, and as long as H5VL supports atomic reference count manipulation.

Similarly, the VOL connector property is released through `H5VL_free_connector_info()` to release the underlying information, and `H5I_dec_ref()` to release the VOL connector ID. Both of these fields were deep copied to the state, and so these operations are threadsafe as long as the application does not modify the VOL connector property, and H5I/H5VL reference count modifications are atomic.

The H5P and H5I threadsafety redesigns should make reference counting of property lists from multiple threads threadsafe. No additional effort will be needed to resolve the threadsafety problems that arise during `H5CX_free_state()` due to its reference count operations on property lists.

5.5.2 State Free Memory Safety

At present, if a context state is freed after the context is populated from that state, the context will contain a pointer to an invalid connector information buffer under `vol_connector_prop`. This is not specifically a threadsafety problem, but is still an issue that should be resolved.

This could be said to be the responsibility of VOL connector authors, since an API context state object is only ever freed as the direct result of the API call `H5VLfree_lib_state()`. That being the case, it is still better to proactively avoid the possibility of invalid memory access from inside the library.

This issue is partially resolved by the already-discussed change of having the `vol_connector_prop` field be attached to the `H5F_ACS_VOL_CONN_NAME` property on the FAPL. Then, `H5CX_retrieve_state()` can implicitly duplicate the connector property in a threadsafe way when copying the property lists, and `H5CX_free_state()` will automatically free it in a threadsafe way with H5P reference decrementing (due to the threadsafe H5P/H5I redesign). The VOL connector information buffer could no longer be freed out from the under the context at any point due to immutable property list versioning, eliminating the possibility of a memory error here.

For the VOL wrapping context, the memory safety issue will remain, since the VOL object wrapping context does not correspond to a property. There are two potential solutions to this issue:

1. At context state free time, if the VOL wrapper's reference count will be decremented to zero, check if the current API context's VOL object wrap context field points to the same address storing the VOL object wrap context to be freed. If it does, then before decrementing the reference count, set the context's VOL wrapper field to `NULL`. This prevents later invalid accesses. It is necessary to remove the pointer before freeing the state since otherwise, another thread could potentially try to access the wrapper on the context between the free and the pointer being unset.
2. The API Context's VOL wrap context accessors could be modified to increment/decrement the wrapper's reference count. It should then be impossible for the context's wrapper pointer to be

invalid, since the context itself would constitute a reference that keeps the wrapper from being released.

Given that the VOL connector-defined `get_wrap_ctx` and `free_wrap_ctx` are already invoked through `H5VL` routines which check the reference count of the VOL wrap context struct, this should require minimal changes, if any, to the VOL wrap context handling.

The preferred solution is currently to do a direct comparison of the wrapper address during an API state free, as this is more straightforward and introduces less complexity.

5.6 Miscellaneous

Use of the `H5FL` module for memory optimization must be disabled for H5CX to be threadsafe.

A H5CX Operations

Accessor routines are described along with their corresponding fields in the appendix section on H5CX fields.

A.1 Internal H5CX Macros

These macros are defined to perform operations that are often repeated within the context module for different fields - retrieving property lists and their values, modifying context fields, and modifying context return properties.

- `H5CX_RETRIEVE_PLIST(PL, FAIL)` - Populates the direct property list pointer (`H5P_genplist_t*`) specified by PL by dereferencing the property list ID on the context with H5I. If the property list pointer is already populated, this is a no-op.

  This operation is not threadsafe, since another thread could free the property entire property list concurrently.

  This macro is only used as a helper within other H5CX macros.

- `H5CX_RETRIEVE_PROP_COMMON(PL, DEF_PL, PROP_NAME, PROP_FIELD)` - Retrieves the property `PROP_NAME` from the property list pointer `PL` and store it on the context field `PROP_FIELD`. If the targeted plist is the default `DEF_PL`, then the default value is copied to the context from the appropriate cache structure.

  This operation is not threadsafe, since another thread could modify the target property concurrently.

  This macro is only used as a helper within other H5CX macros.

- `H5CX_RETRIEVE_PROP_VALID(PL, DEF_PL, PROP_NAME, PROP_FIELD)` - Wrapper around `H5CX_RETRIEVE_PROP_COMMON` for fields which have only a `<field>_valid` flag (i.e. cached fields). If the value on the context is already valid, then this is a no-op.

  This operation is not threadsafe, since another thread could modify the target property concurrently.

- `H5CX_RETRIEVE_PROP_VALID_SET(PL, DEF_PL, PROP_NAME, PROP_FIELD)` - Wrapper around `H5CX_RETRIEVE_PROP_COMMON` for fields with both `<field>_valid` and `<field>_set` flags (i.e. return-and-read fields). `PL` is the property list to retrieve the value from. `DEF_PL` is the default property list of `PL`'s class. `PROP_NAME` is the property to read, and `PROP_FIELD` is the context field on which to store the result. This operation is only a no-op if the value on the context is valid *and* the library itself has never set the value on the context for return.

  The operation must update the context value if `<field>_set` is true in order to ensure that library internal queries get the value from the underlying property list, instead of values set for application return. That being the case, internal queries cause the values from previous internal context set operations to be overwritten, while leaving the 'set' flag true. As such, the value that a user application receives from a property that is a context return field can depend on whether the library reads from that field internally - see the section on Return-and-read fields.

  This operation is not threadsafe, since another thread could modify the target property concurrently.

- `H5CX_SET_PROP(PROP_NAME, PROP_FIELD)` - This macro is used to propagate the return context field `PROP_FIELD` to the DXPL property `PROP_NAME` when a context is popped during `H5CX__pop_common()`. If the `<field>_set` flag is false, this is a no-op because the library has not set that field to be returned.

  This operation is not threadsafe, since other threads may read from or write to the property list while it is in the process of being modified by this macro.

- `H5CX_TEST_SET_PROP(PROP_NAME, PROP_FIELD)` - This macro is only defined when using parallel HDF5 with library instrumentation enabled. It is used to modify the library instrumentation field `PROP_FIELD` on the context.

The context is modified if either the underlying property exists, or if this context field has been set before. Otherwise, this is a no-op.

`PROP_FIELD` is both the name of the field to write to, and the value to write to it.

The instrumentation properties for which this macro is used are not guaranteed to exist, even when instrumentation is enabled - see the section on parallel I/O instrumentation fields.

Note that despite the similarity in name, this macro is inverted from `H5CX_SET_PROP` - that macro writes to a property list from the context, while this macro writes to the context from the provided value.

- `H5CX_get_my_context()` - Wrapper around `H5CX__get_context()`. Returns the context stack for the current thread, an instance of `H5CX_node_t **`.

A.2 H5CX Routines

- `herr_t H5CX_init(void)` - This routine initializes the API Context Interface during phase 2 of library initialization. It populates the default plist cache structures (`H5CX_def_*_cache`) from default property lists,

  This routine depends on H5I to retrieve default property lists with `H5I_object()`, but since the library initialization is single-threaded, this should be completely threadsafe.

  Returns a non-negative value on success, and a negative value on failure.

- `int H5CX_term_package(void)` - Terminates the API Context Interface during library shutdown.

  The only actual work this function performs is to free the top node of the context stack and, if the library was built with threadsafety enabled, unset the thread local pointer to the context stack.

  This function always returns zero, but is described as returning a positive value if it affects other interfaces.

- `H5CX_node_t** H5CX__get_context(void)` - Acquire and return the per-thread API context stack, a pointer to a pointer to a collection of `H5CX_node_t` instances. This is done through use of a unique key for each thread generated within H5TS, `H5TS_apictx_key_g`. If this thread's API context stack is not yet defined, then it is initialized and assigned to the thread local API context key.

- `bool H5CX_pushed(void)` - Checks whether the API context has been pushed, i.e. whether or not the context stack contains a context.

  This routine is used internally by H5T, since some H5T operations which make use of the context can take place during setup of the H5T module, before the context interface is initialized and the context stack is populated. Specifically, `H5T__register()` and `H5T__path_find_real()`.

- `void H5CX__push_common(H5CX_node_t *cnode)` - Internal routine which assigns default field values (default property lists, tag/ring, MPI datatypes) to the provided context node before pushing it onto the context stack. This routine is internal to H5CX and only used inside `H5CX_push()` and `H5CX_push_special()`.

- `herr_t H5CX_push(void)` - Allocates a new context node and places it onto the API context stack via `H5CX__push_common()`. Uses H5FL.

- `void H5CX_push_special(void)` - Allocates a new context node and pushes it onto the API context stack, without using library routines from other modules. This function is used during library shutdown, when modules such as H5FL and H5E are not initialized - notice that this function has no return value, whereas the default `H5CX_push()` returns an `herr_t`.

- `herr_t H5CX_retrieve_state(H5CX_state_t **api_state)` - Allocates `*api_state` and populates it with property lists, the VOL property, and the VOL wrap context. The property lists are copied entirely, the VOL wrap context has its reference count incremented, and the VOL connector property increments the reference count on the connector ID and copies the underlying connector info.

The property list copying is unsafe, since other threads may concurrently modify the property values during the read operation.

The reference count management on the VOL wrap context is not threadsafe, since the context stores a pointer to an external VOL wrap context which might have its reference count be simultaneously modified by other threads.

When a VOL connector is set on the context, a shallow copy of the `H5VL_connector_prop_t` is performed. During this function's construction of the API state object, the connector info itself is deep copied from the context to the state object. This copy is not threadsafe, since the original owner of the connector info object could concurrently modify it during the read. After the copy is complete, the new instance may be freely modified without threadsafety concerns.

This function depends on H5FL, H5I, and H5VL.

- `herr_t H5CX_restore_state(const H5CX_state_t *api_state)` - Writes the values provided from `api_state` to the current API context, restoring the saved state.

  A pointer to the VOL wrapping object is saved on the context. Recall that when an API state is created/retrieved, the reference count of the VOL wrapper is increased. As such, it should be impossible for the VOL wrapper to be freed during or before an `H5CX_restore_state()` operation. Since this function only operates on the top-level pointer to the wrapper, and the wrapper cannot be prematurely freed, the wrapper handling is threadsafe.

  When a VOL connector property is saved on the context, it is only shallowly copied. When that same connector property is saved in an API context state, the underlying connector information is deep copied, but the connector ID only has its reference count increased. This should make it impossible for the connector ID to be freed before or during this context operation. As such, this function's handling of the VOL connector property (shallowly copying it from the state to the current context) is threadsafe.

  While not specifically a threadsafety issue, there is a potential problem with the memory management for the connector information. At state retrieval time, the state allocates a new buffer for the connector information and stores a pointer to it. At state restoration time, the context stores a pointer to that same buffer. After the state is freed by `H5CX_free_state()`, the context's VOL connector property field will contain a pointer to invalid memory.

  Note that the H5CX callbacks normally used to set fields on the context are not used here, although the behavior is mostly identical to if they had been, with the addition of setting direct property list pointers to `NULL`.

  Also note that the reference count of the VOL connector property is not incremented here, since that is modified by the context only when a state object is created or destroyed, not when the state object is used to populate the context.

- `herr_t H5CX_free_state(H5CX_state_t *api_state)` - Frees resources allocated by the provided `api_state`, and then frees the `api_state` buffer itself.

  This is implemented by decrementing the reference count of the state's property list IDs, which is not currently threadsafe. The decrementing of the VOL wrapper's reference count is also not threadsafe, since that object is shared.

  The connector info is freed before the reference count of the ID is decremented. As long as the connector information free callback sets the pointer to NULL, H5VL operations do not necessarily expect connector info to exist just because the connector ID still exists. For that reason, the order in which these operations occur is not believed to be a problem. Aside from that, the freeing of the connector info buffer itself depends on the threadsafety of the VOL connector information's free callback.

  Depends on H5FL, H5I, and H5VL.

- `bool H5CX_is_def_dxpl(void)` - Checks if the API context is using the library's default DXPL. While it is possible for other threads to modify the values present in the default DXPL, only the immutable ID is used for this comparison, and so this operation is threadsafe.

- `herr_t H5CX_set_apl(hid_t *acspl_id, const H5P_libclass_t *libclass, hid_t (H5_ATTR_UNUSED) loc_id, bool (H5_ATTR_UNUSED) is_collective)` - Sets the provided access property list on the context, if valid, and does sanity checking and setup for collective operations.

  If the provided `acspl_id` is either a LAPL, DAPL, or FAPL, as determined by introspection on the provided `libclass`, then it is set as the corresponding property ID field on the context.

  If using parallel HDF5 for a metadata read that is not already guaranteed to be collective, this function uses H5P to peek at a value in the provided `ascpld_id` to determine whether to make the metadata read collective. This function may also issue an MPI barrier for debugging, if parallel sanity checking is enabled via `H5_coll_api_sanity_check_g`.

  This function is not threadsafe, since other threads could modify `acspl_id` during its peek operation.

  This function is used by many other library modules to initialize the correct access propriety list on the context. The modules that use this function are `H5A, H5D, H5F, H5G, H5L, H5M, H5O, H5P, H5R, H5S, H5T` and `H5VL`.

  This function depends on H5P and H5I for its operations on property lists.

  `loc_id` and `is_collective` are flagged as unused parameters when parallel HDF5 is disabled.

- `herr_t H5CX_set_loc(hid_t (H5_ATTR_UNUSED) loc_id)` - This function enables the collective metadata read field on the context, and sets up an MPI barrier for parallel operation sanity checking, if enabled via `H5_coll_api_sanity_check_g`.

  This function retrieves the MPI communicator associated with the file, which is threadsafe since `H5Fmpi` is planned to reside under the global lock.

  `loc_id` is flagged as an unused parameter when parallel HDF5 is disabled. With non-parallel HDF5, this entire function is a no-op.

- `herr_t H5CX_pop(bool update_dxpl_props)` - Pops the current API context, removing it from the context stack, releasing its resources, and propagating any return fields to the underlying property lists.

B Context Fields and Accessors

B.1 Property Lists

Note that while the DXPL, DCPL, LCPL, and LAPL fields each have dedicated setter routines, only the LAPL and DXPL fields also have getter routines. The DAPL and FAPL fields have neither a dedicated getter nor a dedicated setter routine. Setting the DAPL, LAPL, and FAPL on the context should be done via `H5CX_set_apl()`. The overlap between this function and `H5CX_set_lapl()` is likely the reason that `H5CX_set_lapl()` is unused. The context module does not provide an interface to directly retrieve the ID of its FAPL, DAPL, LCPL, or DCPL.

The direct pointers to property list objects are typically initialized for non-default plists by the macro `H5CX_RETRIEVE_PLIST`.

- **dxpl(_id)** - The ID of and a pointer to the underlying dataset transfer property list for this API operation. This defaults to the library's default DXPL, with default values pulled from the H5CX-local `H5CX_def_dxpl_cache` (see C.4 for a full description).

  void H5P_set_dxpl(hid_t dxpl_id) - Sets the DXPL ID in the current API context to `dxpl_id`. Not threadsafe due to the fact that the provided DXPL may be freed after function entry by another thread.

  This function is used in `H5FD, H5T, H5VLnative_attr, H5VLnative_dataset,` and `H5FDsubfiling` to modify the context.

  hid_t H5CX_get_dxpl(void) - Retrieves the DXPL ID for the current API context.

- **lcpl(_id)** - The ID of and a pointer to the underlying link creation property list for this API operation. This defaults to the library's default LCPL, with default values pulled from the H5CX-local `H5CX_def_lcpl_cache` (see C.4 for a full description).

  herr_t H5CX_set_lcpl(hid_t lcpl_id) - Sets the LCPL ID in the current API context to `lcpl_id`. Not threadsafe due to the fact that the provided LCPL may be freed by another thread after function entry.

- **lapl(_id)** - The ID of and a pointer to the underlying link access property list for this API operation. This defaults to the library's default LAPL, with values pulled from the H5CX-local `H5CX_def_lapl_cache` (see C.4 for a full description).

  herr_t H5CX_set_lapl(hid_t lapl_id) - Sets the LAPL ID in the current API context to `lapl_id`. Not threadsafe due to the fact that LAPL may be freed by another thread after this function is entered.

  This function is currently unused within the library.

  hid_t H5CX_get_lapl(void) - Retrieves the LAPL ID for the current API context. Not threadsafe due to the fact that another thread could free the LAPL ID stored on the context concurrently.

- **dcpl(_id)** - The ID of and a pointer to the underlying dataset creation property list for this API operation. This defaults to the library's default DCPL, with values pulled from the H5CX-local `H5CX_def_dcpl_cache` (see C.4 for a full description).

  void H5CX_set_dcpl(hid_t dcpl_id) - Sets the DCPL ID in the current API context. Not threadsafe due to the fact that the provided DCPL may be freed after function entry by another thread.

  This function is used in `H5D` to modify the context.

- **dapl(_id)** - The ID of and pointer to the underlying dataset access property list for this API operation. This defaults to the library's default DAPL, with values pulled from the H5CX-local `H5CX_def_dapl_cache` (see C.4 for a full description).

- **fapl(_id)** - The ID of and pointer to the underlying file access property list for this API operation. This defaults to the library's default FAPL, with values pulled from H5CX-local `H5CX_def_fapl_cache` (see C.4 for a full description).

B.2 Internal Fields

These fields exist only on the API context.

B.2.1 Metadata Cache Info

- **tag** - The metadata tag of the current object.

  void H5CX_set_tag(haddr_t tag) - Sets the object tag for the current API context to the provided `tag`. Since `tag` is an instance of `haddr_t` and not a pointer to a buffer, this operation is threadsafe.

  haddr_t H5CX_get_tag(void) - Retrieves the object tag from the current API context.

- **ring** - The current metadata cache ring for metadata cache entries.

  void H5CX_set_ring(H5AC_ring_t ring) - Sets the metadata cache ring on the current API context. Because `ring` is a typedef'd integer, there is no risk of `ring` being freed during operation, so this operation is threadsafe.

  H5AC_ring_t H5CX_get_ring(void) - Retrieves the metadata cache ring from the current API context.

B.2.2 Parallel I/O Settings

These fields are only defined for parallel HDF5.

- **coll_metadata_read** - Whether to use collective I/O for a metadata read operation.

  void H5CX_set_coll_metadata_read(bool cmdr) - Sets the collective metadata read flag on the current API context to `cmdr`.

  bool H5CX_get_coll_metadata_read(void) - Retrieves the collective metadata read flag from the current API context.

- **btype, ftype** - The MPI datatypes for the buffer and file, respectively, when using collective I/O

  herr_t H5CX_set_mpi_coll_datatypes(MPI_Datatype btype, MPI_Datatype ftype) - Sets the MPI datatypes for collective I/O on the current API context to `btype` and `ftype`. Since `MPI_Datatype` is typedef'd integer, this function is threadsafe.

  herr_t H5CX_get_mpi_coll_datatypes(MPI_Datatype *btype, MPI_Datatype *ftype) - Retrieves the MPI datatypes for collective I/O from the current API context, and modifies the provided buffers to return them.

  If the provided buffers for `btype` and `ftype` existed on an object shared between threads, this function would be non-threadsafe. However, all current usages in the library (`H5FD__subfiling_io_helper()`, `H5FD__mpio_read()`, and `H5FD__mpio_write()`) provide local buffers, so this is not currently an issue.

- **mpi_file_flushing** - Whether an MPI-opened file is being flushed

  void H5CX_set_mpi_file_flushing(bool flushing) - Sets the flag on the current API context that indicates whether an MPI-opened file is currently being flushed.

  bool H5CX_get_mpi_file_flushing(void) - Retrieves the flag from the current API context that indicates whether an MPI-opened file is currently being flushed

- **rank0_bcast** - Whether a dataset meets the requirements for reading with the rank 0 process and broadcasting.

  void H5CX_set_mpio_rank0_bcast(bool rank0_bcast) - Sets the flag on the current API context that indicates whether the dataset meets the requirements for reading with the rank 0 process and broadcasting.

  bool H5CX_get_mpio_rank0_bcast(void) - Retrieves the flag from the current API context that indicates whether the dataset meets the requirements for reading with the rank 0 process and broadcasting.

B.3 Return-Only Fields

Each of these fields has an associated `<field_name>_set` flag.

B.3.1 Parallel I/O Return-Only DXPL Fields

These fields are only defined for parallel HDF5.

- `mpio_actual_chunk_opt` - Chunk optimization mode used for a parallel I/O operation. Corresponds to the DXPL property named
  `H5D_MPIO_ACTUAL_CHUNK_OPT_MODE_NAME`.
- `mpio_actual_io_mode` - The I/O mode used for a parallel I/O operation. Corresponds to the DXPL property named
  `H5D_MPIO_ACTUAL_IO_MODE_NAME`.

B.3.2 Parallel I/O Instrumentation Fields

These fields are only defined for parallel HDF5 when library instrumentation is enabled. They are used in the same way as return-only fields, but even when instrumentation is enabled the library does not directly define these properties. Instead, if parallel HDF5 and library instrumentation are enabled, and the application defines these properties, then they will automatically act like return-only properties.

These fields are likely intended only for parallel debugging, which is why the application has the responsible of defining the properties. An example of defining these properties during the library's tests can be seen in `coll_chunktest()` within `t_bigio.c`.

- **mpio_coll_chunk_link_hard** - The 'collective chunk link hard' value. Corresponds to the DXPL property named
  `H5D_XFER_COLL_CHUNK_LINK_HARD_NAME`.

  herr_t H5CX_test_set_mpio_coll_chunk_link_hard(int mpio_coll_chunk_link_hard) - Sets the instrumented 'collective chunk link hard' value on the current API context. This function is itself threadsafe, but leads to a non-threadsafe write to a property list at API context pop time.

- **mpio_coll_chunk_multi_hard** - The 'collective chunk multi hard' value. Corresponds to the DXPL property named
  `H5D_XFER_COLL_CHUNK_MULTI_HARD_NAME`.

  herr_t H5CX_test_set_mpio_coll_chunk_multi_hard(int mpio_coll_chunk_multi_hard) - Sets the instrumented 'collective chunk multi hard' value on the current API context. This function is itself threadsafe, but leads to a non-threadsafe write to a property list at API context pop time.

- **mpio_coll_chunk_link_num_true** - The 'collective chunk link num true' value. Corresponds to the DXPL property named
  `H5D_XFER_COLL_CHUNK_LINK_NUM_TRUE_NAME`.

  herr_t H5CX_test_set_mpio_coll_chunk_link_num_true(int mpio_coll_chunk_link_num_true) - Sets the instrumented 'collective chunk link num true' value in the current API context. This function is itself threadsafe, but leads to a non-threadsafe write to a property list at API context pop time.

- **mpio_coll_chunk_link_num_false** - The 'collective chunk link num false' value. Corresponds to the DXPL property named
  `H5D_XFER_COLL_CHUNK_LINK_NUM_FALSE_NAME`.

  herr_t H5CX_test_set_mpio_coll_chunk_link_num_false(int mpio_coll_chunk_link_num_false) - Sets the 'collective chunk link num false' value on the current API context. This function is itself threadsafe, but leads to a non-threadsafe write to a property list at API context pop time.

- **mpio_coll_chunk_multi_ratio_coll** - The 'collective chunk multi ratio collective' value. Corresponds to the DXPL property named
  `H5D_XFER_COLL_CHUNK_MULTI_RATIO_COLL_NAME`.

herr_t H5CX_test_set_mpio_coll_chunk_multi_ratio_coll(int mpio_coll_chunk_multi_ratio_coll) - Sets the instrumented 'collective chunk multi ratio coll' value on the current API call. This function is itself threadsafe, but leads to a non-threadsafe write to a property list at API context pop time.

- **mpio_coll_chunk_multi_ratio_ind** - The 'collective chunk multi ratio independence' value. Corresponds to the DXPL property named
  `H5D_XFER_COLL_CHUNK_MULTI_RATIO_IND_NAME`.

  herr_t H5CX_test_set_mpio_coll_chunk_multi_ratio_ind(int mpio_coll_chunk_multi_ratio_ind) - Sets the 'collective chunk multi ratio independence' value. This function is itself threadsafe, but leads to a non-threadsafe write to a property list at API context pop time.

- **mpio_coll_rank0_bcast** - The 'collective rank 0 broadcast' value. Corresponds to the DXPL property named
  `H5D_XFER_COLL_CHUNK_MULTI_RATIO_IND_NAME`.

  herr_t H5CX_test_set_mpio_coll_rank0_bcast(bool mpio_coll_rank0_bcast) - Sets the 'collective rank 0 broadcast' value. This function is itself threadsafe, but leads to a non-threadsafe write to a property list at API context pop time.

B.4 Cached Properties

Each of these fields has an associated `<field_name>_valid` flag.

B.4.1 Cached DXPL Properties

- **max_temp_buf** - Maximum temporary buffer size. Corresponds to DXPL property named `H5D_XFER_MAX_TEMP_BUF_NAME`.

  herr_t H5CX_get_max_temp_buf(size_t *max_temp_buf) - Retrieves the maximum temporary buffer size from the current API context, and returns it via `max_temp_buf`.

  Potentially not threadsafe if the provided buffer exists on a shared object. The actual invocations of this function in `H5D__contig_alloc()`, `H5D__typeinfo_init_phase(2/3)()` are threadsafe.

  This function is not threadsafe due to potentially retrieving values from a property list that could be concurrently modified by other threads.

- **tconv_buf** - Pointer to temporary type conversion buffer. Corresponds to DXPL property named `H5D_XFER_TCONV_BUF_NAME`.

  herr_t H5CX_get_tconv_buf(void **tconv_buf) - Retrieves the temporary buffer pointer from the current API context, and returns it in `*tconv_buf`.

  Potentially not threadsafe if the provided buffer exists on a shared object. The only invocation of this function by the library in `H5D__typeinfo_info_phase3()` is threadsafe.

  This function is not threadsafe due to potentially retrieving values from a property list that could be concurrently modified by other threads.

- **bkgr_buf** - Pointer to background buffer for type conversion. Corresponds to the DXPL property named
  `H5D_XFER_BKGR_BUF_NAME`.

  herr_t H5CX_get_bkgr_buf(void **bkgr_buf) - Retrieves the background buffer pointer from the current API context. Returns it in `*bkgr_buf`.

  Potentially not threadsafe if the provided buffer exists on a shared object. The library's only invocations of this function in `H5D__typeinfo_init()` and `H5D__typeinfo_init_phase3()` are threadsafe.

  This function is not threadsafe due to potentially retrieving values from a property list that could be concurrently modified by other threads.

- **bkgr_buf_type** - Type of the background buffer for type conversion. Corresponds to the DXPL property named
  `H5D_XFER_BKGR_BUF_TYPE_NAME`.

  herr_t H5CX_get_bkgr_buf_type(H5T_bkg_t *bkgr_buf_type) - Retrieves the background buffer type from the current API context, and returns it in `bkgr_buf_type`.

  Potentially not threadsafe if the provided buffer exists on a shared object. The only library invocation of this function in `H5D__typeinfo_init()` is threadsafe.

  This function is not threadsafe due to potentially retrieving values from a property list that could be concurrently modified by other threads.

- **btree_split_ratio** - B-tree split ratios. Corresponds to the DXPL property named
  `H5D_XFER_BTREE_SPLIT_RATIO_NAME`.

  herr_t H5CX_get_btree_split_ratios(double split_ratio[3]) - Retrieves the B-tree split ratios from the current API context, and returns them in the provided `split_ratio` array.

  Potentially non-threadsafe if the provided return buffer exists on a shared object, but the only invocation of this routine (in `H5B__split()`) is threadsafe.

  This function is not threadsafe due to potentially retrieving values from a property list that could be concurrently modified by other threads.

- **vec_size** - Size of the hyperslab vector. Corresponds to the DXPL property named
  `H5D_XFER_HYPER_VECTOR_SIZE_NAME`.

  herr_t H5CX_get_vec_size(size_t *vec_size) - Retrieves the hyperslab vector size from the current API context, and returns it in `vec_size`.

  Potentially not threadsafe if the provided buffer exists on a shared object. All the current uses of this function in `H5Dscatgath.c` and `H5Dselect.c` provide a local buffer for the output, so this does not currently pose any threadsafety issues.

  This function is not threadsafe due to potentially retrieving values from a property list that could be concurrently modified by other threads.

- **err_detect** - Error detection information. Corresponds to the DXPL property named
  `H5D_XFER_EDC_NAME`.

  herr_t H5CX_get_err_detect(H5Z_EDC_t *err_detect) - Retrieves the error detection information from the current API context, and returns it in `err_detect`.

  Potentially not threadsafe if the provided buffer exists on a shared object. All current invocations of this function in `H5Dchunk.c` and `H5Dmpio.c` provide a local buffer for the output, so this does not currently pose any threadsafety issues.

  This function is not threadsafe due to potentially retrieving values from a property list that could be concurrently modified by other threads.

- **filter_cb** - The filter callback function for this operation. Corresponds to the DXPL property named
  `H5D_XFER_FILTER_CB_NAME`.

  herr_t H5CX_get_filter_cb(H5Z_cb_t *filter_cb) - Retrieves the I/O filter callback function from the current API context, and returns it in `filter_cb`.

  Potentially not threadsafe if the provided buffer exists on a shared object. All current invocations of this function (in `H5Dchunk.c` and `H5Dmpio.c`) provide a local buffer for the output, so this does not currently pose any threadsafety issues.

  This function is not threadsafe due to potentially retrieving values from a property list that could be concurrently modified by other threads.

- **data_transform** - Data transformation information. Corresponds to the DXPL property named
  `H5D_XFER_XFORM_NAME`.

  herr_t H5CX_get_data_transform(H5Z_data_xform_t **data_transform) - Retrieves the data transformation expression from the current API context, and returns it in `*data_transform`.

If the context already contains a valid field, the underlying DXPL is not queried. Otherwise, `H5P_peek()` is used to read from the DXPL. Because this property list may be shared with other threads, this operation is not threadsafe.

Another potential source of threadsafety issues is the provided buffer - if it exists on a shared object, then conflict with other threads over that object is possible. Every invocation of this function in the library (within `H5Dio.c` and `H5Dscatgath.c`) allocates a local buffer for the output, so this is not currently a threadsafety issue.

This function is not threadsafe due to potentially retrieving values from a property list that could be concurrently modified by other threads.

- **vl_alloc_info** - VL datatype allocation info. The elements of this struct correspond to the following DXPL properties:

  - `H5D_XFER_VLEN_ALLOC_NAME`
  - `H5D_XFER_VLEN_ALLOC_INFO_NAME`
  - `H5D_XFER_VLEN_FREE_NAME`
  - `H5D_XFER_VLEN_FREE_INFO_NAME`

  herr_t H5CX_set_vlen_alloc_info(H5MM_allocate_t alloc_func, void *alloc_info, H5MM_free_t free_func, void *free_info) - Sets the variable-length datatype allocation information on the current API context. This is accomplished by replacing the previous pointers on the context, if any.

  The provided allocation and free callbacks are invoked within `H5T`, which is not planned for a threadsafe implementation. As such, these callbacks do not need to be threadsafe. The information buffers are provided internally by the library for a few different classes of variable-length data - `H5T_vlen_mem_seq_g`, `H5T_vlen_mem_str_g`, and `H5T_vlen_disk_g`. Those buffers are only acted upon within H5T, which should lie under a global lock. Even if this were not the case, the context never operates on the buffers directly, so operations would still be threadsafe.

  Applications cannot provide vlen allocation methods or callback buffers through any public API, so potential non-threadsafety of application-defined methods is not a concern.

  herr_t H5CX_get_vlen_alloc_info(H5T_vlen_alloc_info_t *vl_alloc_info) - Retrieves the variable-length datatype allocation information from the current API context, and returns it in `vl_alloc_info`.

  If the context already contains a valid field, the underlying DXPL is not queried. Otherwise, `H5P_peek()` is used to read from the DXPL. Because this property list may be shared with other threads, this operation is not threadsafe.

  Another potential source of threadsafety issues is the provided buffer - if it exists on a shared object, then conflict with other threads over that object is possible. Every invocation of this function in the library (within `H5T__conv_vlen()`, `H5T_reclaim()`, and `H5T_vlen_reclaim_elmt()`) allocates a local buffer for the output, so this is not currently a threadsafety issue.

  The retrieved values for `alloc_info` and `free_info` are pointers to the original buffers held under the context. Because the corresponding setter does not copy the underlying buffers either, the retrieved pointers point at potentially shared information. Despite this, the context module never acts on these buffers directly, so there is no opportunity for it to e.g. copy a torn write. As such, the vlen information itself creates no threadsafety issues.

- **dt_conv_cb** - Datatype conversion struct. Corresponds to the DXPL property named `H5D_XFER_CONV_CB_NAME`.

  herr_t H5CX_get_dt_conv_cb(H5T_conv_cb_t *cb_struct) - Retrieves the datatype conversion exception callback from the current API context.

  This function is not threadsafe due to potentially retrieving values from a property list that could be concurrently modified by other threads.

- **selection_io_mode** - The selection I/O mode for this operation. Corresponds to the DXPL property named
`H5D_XFER_SELECTION_IO_MODE_NAME`.

  H5CX_get_selection_io_mode(H5D_selection_io_mode_t *selection_io_mode) - Retrieves the selection I/O mode from the current API context.

  This function is not threadsafe due to potentially retrieving values from a property list that could be concurrently modified by other threads.

- **modify_write_buf** - Whether the library can modify write buffers during this operation. Corresponds to the DXPL property name
`H5D_XFER_MODIFY_WRITE_BUF_NAME`.

  H5CX_get_modify_write_buf(bool *modify_write_buf) - Retrieves the modify write buffer flag from the current API context.

  Potentially not threadsafe if the provided buffer exists on a shared object. This context function is used in one place within the library - `H5D__ioinfo_init()`, where its result is stored on the `H5D_io_info_t` parameter used for output. This I/O info initialization function is itself invoked only in `H5D__read()`/`H5D__write()`, where that I/O info object is locally declared. Thus, this context function is currently only used in threadsafe ways within the library.

  This function is not threadsafe due to potentially retrieving values from a property list that could be concurrently modified by other threads.

B.4.2 Parallel Cached DXPL Properties

These fields are only defined if Parallel HDF5 is enabled.

- **io_xfer_mode** - The parallel transfer mode for this request. Corresponds to the DXPL property named
`H5D_XFER_IO_XFER_MODE_NAME`.

  herr_t H5CX_set_io_xfer_mode(H5FD_mpio_xfer_t io_xfer_mode) - Sets the parallel transfer mode on the current API context.

  herr_t H5CX_get_io_xfer_mode(H5FD_mpio_xfer_t *io_xfer_mode) - Retrieves the parallel transfer mode from the current API context.

  Potentially not threadsafe if the provided output buffer exists on a shared object. This function is used from `H5Cmpio, H5Dchunk, H5Dio, H5Dmpio, H5FDmpio`, and `H5FDsubfiling`. In each use case, the output buffer is always locally allocated. As such, this creates no threadsafety issues.

  This function is not threadsafe due to potentially retrieving values from a property list that could be concurrently modified by other threads.

- **mpio_coll_opt** - Whether the transfer is performed with collective or independent I/O. Corresponds to the DXPL property named
`H5D_XFER_MPIO_COLLECTIVE_OPT_NAME`.

  herr_t H5CX_set_mpio_coll_opt(H5FD_mpio_collective_opt_t mpio_coll_opt) - Sets the parallel transfer mode on the current API context.

  herr_t H5CX_get_mpio_coll_opt(H5FD_mpio_collective_opt_t *mpio_coll_opt) - Retrieves the collective/independent parallel I/O option from the current API context.

  Potentially not threadsafe if the provided buffer for output exists on a shared object. Presently, all internal library invocations of this function within `H5Dmpio.c` and `H5FDmpio.c` use a local buffer, so this does not present any threadsafety issues.

  This function is not threadsafe due to potentially retrieving values from a property list that could be concurrently modified by other threads.

- **mpio_chunk_opt_mode** - The type of chunked dataset I/O for this operation. Corresponds to the DXPL property named
`H5D_XFER_MPIO_CHUNK_OPT_HARD_NAME`.

  herr_t H5CX_get_mpio_chunk_opt_mode(H5FD_mpio_chunk_opt_t *mpio_chunk_opt_mode) - Retrieves the collective chunk optimization mode from the current API context.

  Potentially not threadsafe if the provided output buffer exists on a shared object. The only library internal invocation of this function is in `H5D__piece_io()`, where the output buffer is locally allocated. As such, this creates no threadsafety issues.

  This function is not threadsafe due to potentially retrieving values from a property list that could be concurrently modified by other threads.

- **mpio_chunk_opt_num** - The collective chunk threshold for this operation. Corresponds to the DXPL property named
`H5D_XFER_MPIO_CHUNK_OPT_NUM_NAME`

  herr_t H5CX_get_mpio_chunk_opt_num(unsigned *mpio_chunk_opt_num) - Retrieves the collective chunk optimization threshold from the current API context.

  Potentially not threadsafe if the provided output buffer exists on a shared object. The only library internal invocations of this function are in `H5D__piece_io()`, where the output buffer is locally allocated. As such, this creates no threadsafety issues.

  This function is not threadsafe due to potentially retrieving values from a property list that could be concurrently modified by other threads.

- **mpio_chunk_opt_ratio** - The collective chunk ratio for this operation. Corresponds to the DXPL property named
`H5D_XFER_MPIO_CHUNK_OPT_RATIO_NAME`.

  herr_t H5CX_get_mpio_chunk_opt_ratio(unsigned *mpio_chunk_opt_ratio) - Retrieves the collective chunk optimization ratio from the current API context.

  Potentially not threadsafe if the provided output buffer exists on a shared object. The only library internal invocations of this function are in `H5D__obtain_mpio_mode()`, where the output buffer is locally allocated. As such, this creates no threadsafety issues.

  This function is not threadsafe due to potentially retrieving values from a property list that could be concurrently modified by other threads.

B.4.3 Cached LCPL Properties

- **encoding** - The character encoding used for the link name. Corresponds to the LCPL property named
`H5P_STRCRT_CHAR_ENCODING_NAME`. Note that the link creation property list class inherits this property from its parent class - the string creation property list class.

  herr_t H5CX_get_encoding(H5T_cset_t *encoding) - Retrieves the character encoding from the current API context.

  Potentially not threadsafe if the provided output buffer exists on a shared object. This function is used from `H5L__link_cb()` and `H5L__move()`. In both of these cases, the output is a locally allocated or non-shared object from a module that is planned to stay under the global mutex, and so this does not currently pose any threadsafety issues.

  This function is not threadsafe due to potentially retrieving values from a property list that could be concurrently modified by other threads.

- **intermediate_group** - Whether to create intermediate groups during this operation. corresponds to the LCPL property named `H5L_CRT_INTERMEDIATE_GROUP_NAME`.

  herr_t H5CX_get_intermediate_group(unsigned *crt_intermed_group) - Retrieves the 'create intermediate groups' flag from the current API context.

Potentially not threadsafe if the provided output buffer exists on a shared object. This function is used from `H5L__create_real()` and `H5L__move()`, where the output buffer is locally allocated. As such, this currently poses no threadsafety issues.

This function is not threadsafe due to potentially retrieving values from a property list that could be concurrently modified by other threads.

B.4.4 Cached LAPL Properties

- **nlinks** - Number of soft/UD links to traverse. Corresponds to the LAPL property named `H5L_ACS_NLINKS_NAME`.

  herr_t H5CX_set_nlinks(size_t nlinks) - Sets the number of soft and user-defined links to traverse on the current API context.

  herr_t H5CX_get_nlinks(size_t *nlinks) - Retrieves the number of soft and user-defined links to traverse from the current API context.

  Potentially not threadsafe if the provided output buffer exists on a shared object. This function is used from `H5G_traverse()`, `H5G__traverse_special()`, and `H5L__move()`. In each invocation, its output buffer is locally allocated, and so this does not currently pose any threadsafety issues.

  This function is not threadsafe due to potentially retrieving values from a property list that could be concurrently modified by other threads.

B.4.5 Cached DCPL Properties

- **do_min_dset_ohdr** - Whether to minimize the dataset object header. Corresponds to the DCPL property named `H5D_CRT_MIN_DSET_HDR_SIZE_NAME`.

  herr_t H5CX_get_dset_min_ohdr_flag(bool *dset_min_ohdr_flag) - Retrieves the flag that indicates whether the dataset object header should be minimized from the current API context.

  Potentially not threadsafe if the output buffer provided exists on a shared object. Used from `H5D__use_minimized_dset_headers()`, where the output buffer is locally allocated in `H5D__update_oh_info()`. As such, this does not currently pose any threadsafety issues.

  This function is not threadsafe due to potentially retrieving values from a property list that could be concurrently modified by other threads.

- **ohdr_flags** - Object header flags. Corresponds to the DCPL property named `H5O_CRT_OHDR_FLAGS_NAME`. Note that the default DCPL inherits this class from the default OCPL.

  herr_t H5CX_get_ohdr_flags(uint8_t *ohdr_flags) - Retrieves the object header flags from the current API context.

  Potentially not threadsafe if the provided output buffer exists on a shared object. This function is only used from `H5O__create_ohdr()`, where the output buffer is locally allocated. As such, this does not currently create any threadsafety issues.

  This function is not threadsafe due to potentially retrieving values from a property list that could be concurrently modified by other threads.

B.4.6 Cached DAPL Properties

- **extfile_prefix** - Prefix for the external file. Corresponds to the DAPL property named `H5D_ACS_EFILE_PREFIX_NAME`.

  herr_t H5CX_get_ext_file_prefix(const char **extfile_prefix) - Retrieves the prefix for an external file frmo the current API context.

  Potentially not threadsafe if the provided output buffer exists on a shared object. This function is used from `H5D__build_file_prefix()`, where the output buffer is locally allocated. As such, this does not currently create any threadsafety issues.

  This function is not threadsafe due to potentially retrieving values from a property list that could be concurrently modified by other threads.

- **vds_prefix** - Prefix for virtual dataset. Corresponds to the DAPL property named `H5D_ACS_VDS_PREFIX_NAME`.

  herr_t H5CX_get_vds_prefix(const char **vds_prefix) - Retrieves virtual dataset prefix from the current API context.

  Potentially not threadsafe if the provided output buffer exists on a shared object. This function is used from `H5D__build_file_prefix`, where the output buffer is the `H5Dint` module-local `H5D_prefix_vds_env`. Currently, this variable is only otherwise written to during module initialization, and is only read from after the `H5CX_get_vds_prefix()` call, and so this does not raise any threadsafety issues at present. Even if more references to `H5D_prefix_vds_env` were added to `H5Dint`, that module is planned to reside under the global mutex, and so this should still not create any threadsafety issues.

  This function is not threadsafe due to potentially retrieving values from a property list that could be concurrently modified by other threads.

B.4.7 Cached FAPL Properties

- **low_bound, high_bound** - The bound properties for `H5Pset_libver_bounds()`. Correspond to the FAPL properties named `H5F_ACS_LIBVER_(LOW/HIGH)_BOUND_NAME`.

  herr_t H5CX_get_libver_bounds(H5F_libver_t *low_bound, H5F_libver_t *high_bound) - Retrieves library format version bounds from either the context or the FAPL. This function is not threadsafe due to potentially retrieving values from a property list that could be concurrently modified by other threads.

  herr_t H5CX_set_libver_bounds(H5F_t *f) - Retrieves the low and high library format version bounds from the provided file handle `f`, and sets them on the context. If either bound is not defined, the context value will default to `H5F_LIBVER_LATEST`.

B.4.8 Cached VOL Settings

- **vol_connector_prop** - VOL connector ID and info. Does not correspond to a property in a property list. Default value is a buffer filled with 0.

  To set this field on the context, a shallow copy is performed, copying the property's ID and the pointer to its connector information. The getter and setter act only on these pointers, and so are threadsafe with respect to modification of the property. Since the VOL property may be modified by other threads, and those other threads may have their writes delayed indefinitely, it is possible for the context-provided connector property to point to invalid memory.

  Note that neither VOL connector property accessor makes any reference to the FAPL stored on the context, which may also contain a VOL connector property. This is believed to be because when the context was first developed, the FAPL was not stored on the context, and so the VOL connector property was manually inserted.

  herr_t H5CX_set_vol_connector_prop(const H5VL_connector_prop_t *vol_connector_prop) - Sets the VOL connector ID and info on the current API context. This is done through a shallow copy of the VOL connector property. If another thread freed either the connector ID or the connector information part of the provided property during this function, the context would store a pointer to invalid memory. As such, this function is not threadsafe.

  herr_t H5CX_get_vol_connector_prop(H5VL_connector_prop_t *vol_connector_prop) - Retrieves the VOL connector property (connector ID and info) from the current API context. If the context contains a valid value, then the connector property is shallow copied into the provided output buffer. If the context does not contain a valid value, fills the output buffer with zeros.

  If the provided buffer is part of a shared object, this is potentially not threadsafe. However, the only internal use of this function by the library, in `H5F__set_vol_conn()`, provides a local buffer. As such, this is not a problem.

  This function is not threadsafe, since the objects pointed to by the context's `vol_connector_prop` field could be freed by another thread, leaving the context pointing to invalid memory.

- **vol_wrap_ctx** - The VOL connector's 'wrap context' used to create IDs. Does not correspond to a property in a property list.

  When this field is set on a context, what is stored is just a pointer to the external VOL wrapping object. Thus, other threads could potentially modify this field during context operations. Thus, this field and operations involving it are not threadsafe.

  herr_t H5CX_set_vol_wrap_ctx(void *vol_wrap_ctx) - Sets the VOL object wrapping context for an operation. This routine sets the value of a pointer on the API context, and does not copy the underlying VOL wrapper. As a result, several other context operations which act on the VOL wrapper are not threadsafe, due to referencing an object which could be modified by other threads.

  herr_t H5CX_get_vol_wrap_ctx(void **vol_wrap_ctx) - Retrieves the VOL object wrapping context from the current API context.

  Potentially not threadsafe if the provided output buffer exists on a shared object. This function is only used within `H5F__dest()` and `H5VL__set_def_conn()`, where the output buffer is locally allocated. As such, this presents no threadsafety issues.

  Unless cached property list fields, if no valid value is found on the context, then a NULL value is returned. As such, this operation has no direct threadsafety issues.

B.5 Return-and-Read Fields

Each of these fields has an associated `<field_name>_valid` and `<field_name>_set` flag on the context struct.

- **no_selection_io_cause** - The reason for not performing selection I/O this operation. Corresponds to the DXPL property named `H5D_XFER_NO_SELECTION_IO_CAUSE_NAME`.

  void H5CX_set_no_selection_io_cause(uint32_t no_selection_io_cause) - Sets the reason for not performing selection I/O on the current API context.

  herr_t H5CX_get_no_selection_io_cause(uint32_t *no_selection_io_cause) - Retrieves the cause for not performing selection I/O from the current API context.

  This function is not threadsafe due to potentially retrieving values from a property list that could be concurrently modified by other threads.

- **actual_selection_io_mode** - The actual selection I/O mode used for this operation, which may differ from the requested mode. Corresponds to the DXPL property named `H5D_XFER_ACTUAL_SELECTION_IO_MODE_NAME`.

  void H5CX_set_actual_selection_io_mode(uint32_t actual_selection_io_mode) - Sets the actual selection I/O mode on the current API context.

  If the API context is using the default DXPL, then this operation becomes a no-op to preserve the default selection I/O mode.

  herr_t H5CX_get_actual_selection_io_mode(uint32_t *actual_selection_io_mode) - Retrieves the actual I/O mode (scalar, vector, and/or selection) from the current API context.

  Unlike other getter routines for cached property list fields, this operation retrieves its value from the default DXPL cache even when using a non-default DXPL, as long as the value has not been previously set internally by the library or previously cached in the context itself. The purpose of this special behavior is to wipe out any previous selection I/O mode settings.

  This function is not threadsafe due to potentially retrieving values from a property list that could be concurrently modified by other threads.

- **mpio_local_no_coll_cause** - The local reason for breaking collective I/O for this operation. Corresponds to the DXPL property named `H5D_MPIO_LOCAL_NO_COLLECTIVE_CAUSE_NAME`.

  void H5CX_set_mpio_local_no_coll_cause(uint32_t mpio_local_no_coll_cause) - Sets the local reason for breaking collective I/O on the current API context.

If the context is using the default DXPL, then this operation becomes a no-op and does not modify the context.

herr_t H5CX_get_mpio_local_no_coll_cause(uint32_t *mpio_local_no_coll_cause) - Retrieves the local cause for breaking collective I/O from the current API context.

Potentially not threadsafe if the provided output buffer exists on a shared object. Its only internal invocation within the library in `H5D__mpio_get_no_coll_cause_strings()` uses a local buffer, so this presents no threadsafety issues.

This function is not threadsafe due to potentially retrieving values from a property list that could be concurrently modified by other threads.

- **mpio_global_no_coll_cause -** The global reason for breaking collective I/O for this operation. Corresponds to the DXPL property named `H5D_MPIO_GLOBAL_NO_COLLECTIVE_CAUSE_NAME`.

  void H5CX_set_mpio_global_no_coll_cause(uint32_t mpio_global_no_coll_cause) - Sets the global reason for breaking collective I/O on the current API context.

  If the context is using the default DXPL, then this operation becomes a no-op and does not modify the context.

  herr_t H5CX_get_mpio_global_no_coll_cause(uint32_t *mpio_global_no_coll_cause) - Retrieves the global cause for breaking collective I/O from the current API context.

  Potentially not threadsafe due if the provided output buffer exists on a shared object. The only library internal invocation of this function is in `H5D__mpio_get_no_coll_cause_strings()`, where the output is a locally allocated buffer. As such, this creates no threadsafety issues.

  This function is not threadsafe due to potentially retrieving values from a property list that could be concurrently modified by other threads.

C Structure Definitions

C.1 H5CX_t

The `H5CX_t` struct represents an entire API context.

```
typedef struct H5CX_t {
    hid_t           dxpl_id;
    H5P_genplist_t *dxpl;

    hid_t           lcpl_id;
    H5P_genplist_t *lcpl;

    hid_t           lapl_id;
    H5P_genplist_t *lapl;

    hid_t           dcpl_id;
    H5P_genplist_t *dcpl;

    hid_t           dapl_id;
    H5P_genplist_t *dapl;

    hid_t           fapl_id;
    H5P_genplist_t *fapl;

    haddr_t tag;
    H5AC_ring_t ring;

#ifdef H5_HAVE_PARALLEL
    bool         coll_metadata_read;
    MPI_Datatype btype;
    MPI_Datatype ftype;
    bool         mpi_file_flushing;
    bool         rank0_bcast;
#endif

    size_t    max_temp_buf;
    bool      max_temp_buf_valid;
    void     *tconv_buf;
    bool      tconv_buf_valid;
    void     *bkgr_buf;
    bool      bkgr_buf_valid;
    H5T_bkg_t bkgr_buf_type;
    bool      bkgr_buf_type_valid;
    double    btree_split_ratio[3];
    bool      btree_split_ratio_valid;
    size_t    vec_size;
    bool      vec_size_valid;
#ifdef H5_HAVE_PARALLEL
    H5FD_mpio_xfer_t io_xfer_mode;
    bool             io_xfer_mode_valid;
    H5FD_mpio_collective_opt_t mpio_coll_opt;
    bool mpio_coll_opt_valid;
    H5FD_mpio_chunk_opt_t
             mpio_chunk_opt_mode;
    bool     mpio_chunk_opt_mode_valid;
    unsigned mpio_chunk_opt_num;
    bool     mpio_chunk_opt_num_valid;
    unsigned mpio_chunk_opt_ratio;
    bool     mpio_chunk_opt_ratio_valid;
#endif
    H5Z_EDC_t                err_detect;
    bool                     err_detect_valid;
    H5Z_cb_t                 filter_cb;
    bool                     filter_cb_valid;
    H5Z_data_xform_t        *data_transform;
    bool                     data_transform_valid;
    H5T_vlen_alloc_info_t    vl_alloc_info;
    bool                     vl_alloc_info_valid;
    H5T_conv_cb_t            dt_conv_cb;
    bool                     dt_conv_cb_valid;
    H5D_selection_io_mode_t selection_io_mode;
    bool                     selection_io_mode_valid;
    bool modify_write_buf;
    bool modify_write_buf_valid;

#ifdef H5_HAVE_PARALLEL
    H5D_mpio_actual_chunk_opt_mode_t mpio_actual_chunk_opt;
    bool mpio_actual_chunk_opt_set;
    H5D_mpio_actual_io_mode_t
             mpio_actual_io_mode;
    bool     mpio_actual_io_mode_set;
    uint32_t mpio_local_no_coll_cause;
    bool     mpio_local_no_coll_cause_set;
    bool     mpio_local_no_coll_cause_valid;
    uint32_t mpio_global_no_coll_cause;
    bool mpio_global_no_coll_cause_set;
    bool mpio_global_no_coll_cause_valid;
#ifdef H5_HAVE_INSTRUMENTED_LIBRARY
    int mpio_coll_chunk_link_hard;
    bool mpio_coll_chunk_link_hard_set;
    int  mpio_coll_chunk_multi_hard;
    bool mpio_coll_chunk_multi_hard_set;
    int  mpio_coll_chunk_link_num_true;
    bool mpio_coll_chunk_link_num_true_set;
    int mpio_coll_chunk_link_num_false;
    bool mpio_coll_chunk_link_num_false_set;
    int mpio_coll_chunk_multi_ratio_coll;
    bool mpio_coll_chunk_multi_ratio_coll_set;
    int mpio_coll_chunk_multi_ratio_ind;
    bool mpio_coll_chunk_multi_ratio_ind_set;
    bool mpio_coll_rank0_bcast;
    bool mpio_coll_rank0_bcast_set;
#endif
#endif
    uint32_t no_selection_io_cause;
    bool no_selection_io_cause_set;
    bool no_selection_io_cause_valid;

    uint32_t actual_selection_io_mode;
    bool actual_selection_io_mode_set;
    bool actual_selection_io_mode_valid;
```

```
        H5T_cset_t encoding;
        bool       encoding_valid;
        unsigned intermediate_group;
        bool     intermediate_group_valid;

        size_t nlinks;
        bool   nlinks_valid;

        bool    do_min_dset_ohdr;
        bool    do_min_dset_ohdr_valid;
        uint8_t ohdr_flags;
        bool    ohdr_flags_valid;

        const char *extfile_prefix;
        bool        extfile_prefix_valid;
        const char *vds_prefix;
        bool        vds_prefix_valid;

        H5F_libver_t low_bound;
        bool         low_bound_valid;
        H5F_libver_t high_bound;
        bool high_bound_valid;

        H5VL_connector_prop_t vol_connector_prop;
        bool  vol_connector_prop_valid;
        void *vol_wrap_ctx;
        bool  vol_wrap_ctx_valid;
} H5CX_t;
```

C.2 H5CX_node_t

This structure represents an entry on the API context stack. It consists of an API context, and a pointer that can be used to find the next (lower) context in the stack.

```
typedef struct H5CX_node_t {
    H5CX_t              ctx;
    struct H5CX_node_t *next;
} H5CX_node_t;
```

C.3 H5CX_state_t

This structure represents a 'saved' API context state which may be 'resumed' later, used primarily for VOL connectors which execute operations in an unpredictable order.

```
typedef struct H5CX_state_t {
    hid_t                 dcpl_id;
    hid_t                 dxpl_id;
    hid_t                 lapl_id;
    hid_t                 lcpl_id;
    void                 *vol_wrap_ctx;
    H5VL_connector_prop_t vol_connector_prop;

#ifdef H5_HAVE_PARALLEL
    bool coll_metadata_read;
#endif
} H5CX_state_t;
```

C.4 Cached Default Property List Structures

These structures are populated at H5CX initialization time during library startup. They store values from default property lists in order to avoid costly H5P lookup operations for default property lists.

C.4.1 H5CX_dxpl_cache_t

```
typedef struct H5CX_dxpl_cache_t {
    size_t    max_temp_buf;
    void     *tconv_buf;
    void     *bkgr_buf;
    H5T_bkg_t bkgr_buf_type;
    double    btree_split_ratio[3];
    size_t    vec_size;
#ifdef H5_HAVE_PARALLEL
    H5FD_mpio_xfer_t io_xfer_mode;
    H5FD_mpio_collective_opt_t mpio_coll_opt;
    uint32_t mpio_local_no_coll_cause;
    uint32_t mpio_global_no_coll_cause;
    H5FD_mpio_chunk_opt_t
             mpio_chunk_opt_mode;
    unsigned mpio_chunk_opt_num;
    unsigned mpio_chunk_opt_ratio;
#endif
    H5Z_EDC_t                err_detect;
    H5Z_cb_t                 filter_cb;
    H5Z_data_xform_t        *data_transform;
    H5T_vlen_alloc_info_t    vl_alloc_info;
    H5T_conv_cb_t            dt_conv_cb;
    H5D_selection_io_mode_t selection_io_mode;
    uint32_t                 no_selection_io_cause;
    uint32_t actual_selection_io_mode;
    bool modify_write_buf;
} H5CX_dxpl_cache_t;
```

C.4.2 H5CX_lcpl_cache_t

```
typedef struct H5CX_lcpl_cache_t {
    H5T_cset_t encoding;
    unsigned   intermediate_group;
} H5CX_lcpl_cache_t;
```

C.4.3 H5CX_lapl_cache_t

```
typedef struct H5CX_lapl_cache_t {
    size_t nlinks;
} H5CX_lapl_cache_t;
```

C.4.4 H5CX_dcpl_cache_t

```
typedef struct H5CX_dcpl_cache_t {
    bool    do_min_dset_ohdr;
    uint8_t ohdr_flags;
} H5CX_dcpl_cache_t;
```

C.4.5 H5CX_dapl_cache_t

```
typedef struct H5CX_dapl_cache_t {
    const char *extfile_prefix;
    const char *vds_prefix;
} H5CX_dapl_cache_t;
```

C.4.6 H5CX_fapl_cache_t

```
typedef struct H5CX_fapl_cache_t {
    H5F_libver_t low_bound;
    H5F_libver_t high_bound;
} H5CX_fapl_cache_t;
```

C.5 VOL Connector Property

This structure stores the ID of a VOL connector and a buffer of information for FAPLs.

```
typedef struct H5VL_connector_prop_t {
    hid_t       connector_id;
    const void *connector_info;
} H5VL_connector_prop_t;
```

As its name suggests, this structure is the value of the property named `H5F_ACS_VOL_CONN_NAME` on a FAPL when using a VOL connector. Both of its fields are provided to the library through the API call `H5Pset_vol()`, which modifies the provided FAPL. The set VOL function is not usually invoked directly by an application, since most VOLs provide a routine of the form `H5Pset_fapl_<vol_name>()` which registers the connector with the library to generate a connector ID and sets the connector information to a value decided by the VOL connector's author.

C.5.1 Connector ID

The 'connector id' is the ID assigned to the VOL connector by `H5VLregister_connector()`. Its underlying object is an instance of `H5VL_class_t` (see section C.8), the struct that defines a VOL connector's callbacks as well as some metadata such as name and version number. (Note that the 'connector id' discussed here, an instance of `hid_t` that is assigned by H5I, is distinct from each VOL connector's unique 'connector value' or 'connector identifier', a constant `H5VL_class_value_t` set by the VOL connector's author.)

After this property has been set on a FAPL, the underlying VOL connector (instance of `H5VL_class_t`) is constant. At setup time, it is copied from the application/VOL-provided buffer, and the VOL name has its own memory allocated internally. The VOL connector struct is not freed until `H5VLunregister_connector()` is invoked or the library is shutdown (see section C.8 for more information on the VOL class structure)

C.5.2 Connector Information

The connector information is a generic buffer provided to allow applications to more finely control VOL connector operations. It is stored on the FAPL intially, and then shared with the API context. Each VOL connector may optionally provide a set of callbacks to manipulate it in the connector's `info_cls` field. These callbacks are copy, compare, free, convert to string, and convert from string.

- The copy callback is invoked when the connector is set for use on a FAPL. It should deep copy the connector information in such a manner that the original data can be freed.
- The free callback is used to free the connector-specific information, and should release any resources allocated by the info copy callback. If the copy callback is defined, this must be defined.
- The compare callback determines if two provided connector-specific data structs are identical.

- The to-string and from-string callbacks convert the provided connector information to and from a configuration string, which may be used to e.g. set up a VOL connector from the command line.

The info class also stores a 'size' field indicating the total size of the connector information buffer. All known VOLs that use the info class provide a fixed value for this field that is the size of a custom VOL connector info struct.

Presently, these callbacks and the connector information buffer as a whole are not threadsafe due to connector information inheriting the threadsafety issues of all property list values, as well as due to connector information being shared between the API context and the FAPL during an operation.

C.6 Variable-Length Allocation Information

```
typedef struct {
    H5MM_allocate_t alloc_func; /* Allocation function */
    void           *alloc_info; /* Allocation information */
    H5MM_free_t     free_func;  /* Free function */
    void           *free_info;  /* Free information */
} H5T_vlen_alloc_info_t;
```

C.7 VOL Object Wrapping Structures

The VOL object wrapping context exists to support stacked VOL connectors. After a VOL object representing an HDF5 data model object (dataset, group, etc.) is returned to the HDF5 library, it is attached to a generic VOL object struct. If passthrough VOLs are in use, then the object will first be 'wrapped' before being attached.

The VOL wrap context structure stores its own reference count, a pointer to the VOL connector it belongs to, and a pointer to an internal buffer that acts as the VOL object wrapping context passed to VOL wrap callbacks.

```
typedef struct H5VL_wrap_ctx_t {
    unsigned rc;
    H5VL_t  *connector;
    void    *obj_wrap_ctx;
} H5VL_wrap_ctx_t;
```

To explain the VOL object wrapping context more deeply, it is beneficial to first understand VOL object wrapping in general. There are two types of VOL connectors - passthrough and terminal. Terminal VOL connectors interact directly with storage, while passthrough connectors reside between the library and another VOL connector, or between two VOL connectors. With passthrough VOL connectors, it is possible to use more than one VOL connector in a single 'stack'. This is when VOL object wrapping and unwrapping is necessary.

Consider the example of opening a file. The native VOL's file open callback returns a buffer containing the native VOL-defined `H5F_t` structure representing an HDF5 file. This VOL-specific object is cast to `void*` and attached to the 'data' field of an `H5VL_object_t` within the library's VOL layer.

```
typedef struct H5VL_object_t {
    void   *data;      /* Pointer to connector-managed data for this object  */
    H5VL_t *connector; /* Pointer to VOL connector struct                    */
    size_t  rc;        /* Reference count                                    */
} H5VL_object_t;
```

Later, API routines will be provided with this object, and pass the 'data' field back down to native VOL callbacks. Since the native VOL's callbacks were written to expect and operate on `H5F_t` and similar native VOL structures, this works fine.

Now consider using the library's Internal Passthrough VOL (`H5VL_pass_through_g`) on top of the native VOL. The native VOL should return an `H5F_t` in a void* buffer to the passthrough VOL, and the passthrough VOL should similarly return its own unique structure (`H5VL_pass_through_t`) in a void* buffer to the library VOL interface, where it will again be attached to the 'data' field of an `H5VL_object_t`. This process of going from a lower VOL's object representation to a higher VOL's representation is called "wrapping" the object.

```
typedef struct H5VL_pass_through_t {
    hid_t under_vol_id; /* ID for underlying VOL connector */
    void *under_object; /* Info object for underlying VOL connector */
} H5VL_pass_through_t;
```

Later, API calls will be provided with this VOL object and will pass the 'data' field containing an `H5VL_pass_through_t` to the Passthrough connector. The Passthrough connector must "unwrap" its own structure in order to access the object provided by the next VOL on the stack (`H5F_t`) and pass the object that to the lower VOL's own callback.

If the unwrapping of VOL objects did not occur, then when passing the pointer stored in an `H5VL_object_t` down the VOL stack, the lower VOLs would almost certainly crash. This is because, to use the current example, the native VOL expects to receive an `H5F_t`, but would be provided with an `H5VL_pass_through_t`, and the improper casting of the buffer would lead to undefined behavior.

Similarly, if the wrapping process did not occur, then subsequent API calls using the `H5VL_object_t` would almost certainly crash, because higher VOLs would receive structures defined by lower VOLs when expecting to operate on their own structures.

In general, all passthrough VOLs must wrap objects provided from lower layers in their own structures before returning them, and must unwrap objects provided from higher layers before passing them down to lower layers. The terminal VOL does not need to wrap or unwrap objects - since there should be no VOL connector below it - and as such its wrap callbacks should not be defined.

In the majority of cases, this process of wrapping and unwrapping is performed directly by the VOL callbacks that interact with objects, *not* the callbacks in `wrap_cls`. For example, the Internal Passthrough VOL's file open callback `H5VL_pass_through_file_open()` wraps the object received from the lower VOL via `H5VL_pass_through_new_obj()`, which constructs the passthrough VOL's own object-representing struct `H5VL_pass_through_t`.

In some cases, however, the library needs to create or unwrap an ID or object pointer from somewhere that does not pass through the VOL layer. As such, it is necessary for the library to have some way to directly invoke the VOL connector stack's wrapping and unwrapping functionality. This is the purpose of the wrap callbacks that each passthrough VOL should define through the VOL class's `wrap_cls` field, an instance of `H5VL_wrap_class_t`. For example, the API function `H5VLget_file_type()` may need to create a VOL object which is a wrapped version of the provided file object.

C.7.1 VOL Object Wrapping Callbacks

Each VOL class may provide a set of object wrapping callbacks via its `wrap_cls` field. These callbacks must be defined for passthrough connectors, and should not be defined for terminal VOL connectors.

```
typedef struct H5VL_wrap_class_t {
    void *(*get_object)(const void *obj);
    herr_t (*get_wrap_ctx)(const void *obj, void **wrap_ctx);
    void *(*wrap_object)(void *obj, H5I_type_t obj_type, void *wrap_ctx);
    void *(*unwrap_object)(void *obj);
    herr_t (*free_wrap_ctx)(void *wrap_ctx);
} H5VL_wrap_class_t;
```

Each callback should recursively invoke the same callback for the next VOL in the connector stack via a public H5VL operation either before or after performing its own operation.

A passthrough VOL's wrap context should contain the ID and the wrap context of the next VOL in the stack. If this information is not stored in the wrap context, then the `wrap_object()` callback will not have enough information to recursively wrap the provided object.

Similarly, a passthrough VOL's wrapped object should contain the ID and the object initially provided from the next VOL in the stack. If this information is not stored on the wrapped object, then the `unwrap_object()` callback will not have enough information to recursively unwrap a wrapped object.

The expected semantics for each individual callback are as follows:

- `void *get_object(const void *obj)` - Should return a pointer to the object at the very bottom of the connector stack (e.g. the object provided by the terminal VOL connector). `obj` is a buffer containing an object that has been wrapped by this VOL and each lower VOL in the current connector stack.

  Performing this will entail using `H5VLget_object()` to recursively invoke the `get_object()` callback, if any, of the next VOL in the stack. This routine must be provided with the library ID of the next VOL connector in the stack, and a pointer to the object stored from the next VOL. It is the responsibility of each passthrough VOL connector to somehow store this information. The most straightforward way is to store the information for this operation is on the passthrough VOL's own wrapped object - for example, the Internal Passthrough VOL's wrapper struct `H5VL_pass_through_t` consists entirely of these two pieces of information.

  This callback is similar to `unwrap_object()`, but does not disturb or modify the provided wrapping structure.

- `herr_t get_wrap_ctx(const void *obj, void **wrap_ctx)` - Retrieve the VOL object wrapping context for the current VOL and all connectors below it. `obj` is an object wrapped by this VOL connector and all connectors below it. The wrap context should be returned in dynamically allocated memory under `*wrap_ctx`. This memory should be freed by a later call to `free_wrap_ctx()`.

  The wrap context is an object allocated by passthrough VOLs in order to store any information necessary for lower VOLs to perform object wrapping. For example, the Internal Passthrough connector's wrap context consists of the next VOL's ID and the next VOL's wrap context, which is the the information necessary for the Internal Passthrough connector's `wrap_object()` callback to recursively invoke the next VOL's `wrap_object()` callback through `H5VLwrap_object()`.

  Note that while this callback is used to set up the VOL's wrapping context, it is provided with an object that is already wrapped by this VOL. If this is the first-time wrap context setup for this VOL, then the provided object must have been wrapped by one of the VOL's object manipulation callbacks.

- `void *wrap_object(void *obj, H5I_type_t obj_type, void *wrap_ctx)` - Should perform this VOL's wrapping operation on the provided object and return it. `obj` is an object which has already been wrapped by every lower VOL. After this callback completes, the object should also be wrapped by this VOL. `obj_type` is the type of the provided object, and `wrap_ctx` is a VOL-defined buffer which contains information necessary to perform the wrapping.

  Performing this will entail using `H5VLwrap_object()` to invoke the `wrap_object()` callback of the next VOL in the stack, before performing this VOL's own wrapping of the object. `H5VLwrap_object()` requires both the connector ID and the wrap context for the next VOL, which should be provided through `wrap_ctx`.

  The provided `obj` buffer should not be modified, since it was provided by another VOL and its contents are unknown.

- `void *unwrap_object(void *obj)` - Should return a pointer to the object at the very bottom of the connector stack (e.g. the object provided by the terminal VOL connector). `obj` is a buffer containing an object that has been wrapped by this VOL and each lower VOL in the current connector stack.

  Performing this will entail first unwrapping this VOL's own struct, and then recursively unwrapping the underlying object through `H5VLunwrap_object()`. An object wrapped by this VOL should provide the information necessary to invoke `H5VLunwrap_object()`, and so this callback does not need to receive the wrap context as a parameter.

This callback is similar to `get_object()`, except that this callback should perform cleanup and release any resources allocated during the wrapping process by `wrap_object()`.

- `herr_t free_wrap_ctx(void *wrap_ctx)` - Should release the VOL wrapping context `wrap_ctx`, releasing any resources allocated within `get_wrap_ctx()`. This will involve directly freeing this VOL's own allocated resources before recursively freeing the resources allocated by all lower VOLs through `H5VLfree_wrap_ctx()`.

The VOL object wrapping context is allocated initially by the `get_wrap_ctx()` callback. Whether or not it is threadsafe depends on what objects retain access to the wrapping context, which in turn depends on how the library uses this function internally. This function is called from two higher level functions.

The first is `H5VL_get_wrap_ctx()`, which returns the context directly to the caller. This is only used by the public API's `H5VLget_wrap_ctx()`, but neither of these calls save the VOL wrapping context on the API context, and so its threadsafety is dependent entirely on how it manipulated by the application.

The second routine using `get_wrap_ctx()` is `H5VL_set_vol_wrapper()`. This is used within the VOL layer to populate the API context with VOL wrapper information before VOL-defined object callbacks are invoked. No pointers to the VOL wrapping context are saved anywhere besides on the API context, and so the VOL wrapping context should present no threadsafety issues at this layer. The existence of the VOL wrap context does increase the reference count of its corresponding VOL, but reference counting of H5VL objects should be made threadsafe as part of making H5VL/H5I threadsafe.

In summary, the VOL wrapping context field stored on the API context is threadsafe as long as all active passthrough VOLs do not define a wrap context containing references to information that may be modified by other threads.

C.7.2 Review of Existing Passthrough VOL Connectors

Because VOL wrap contexts must recursively contain pointers to wrap contexts for lower VOLs, a non-threadsafe implementation of the wrap context implicitly makes the wrap contexts of higher VOLs in the stack not threadsafe. This section considers the threadsafety of each passthrough VOL in isolation.

- The Internal Passthrough VOL connector defines a VOL wrap context that contains only the ID and the wrap context of the next VOL connector. As long as the next VOL is not unregistered concurrently and itself defines a threadsafe VOL context, this wrap context is threadsafe.
- The External Passthrough VOL has an identical VOL wrap context to the Internal Passthrough VOL, and thus defines a threadsafe VOL wrap context.
- The Cache VOL has an identical VOL wrap context to the Internal Passthrough VOL, and thus defines a threadsafe VOL wrap context.
- The LowFive VOL has an identical VOL wrap context to the Internal Passthrough VOL, and thus defines a threadsafe VOL wrap context.
- The dset-split VOL has an identical VOL wrap context to the Internal Passthrough VOL, and thus defines a threadsafe VOL wrap context.
- The Async VOL defines a non-threadsafe wrap context. The Async wrap context stores a VOL-wrapping of a library object, which is unwrapped from the object provided to `get_wrap_ctx()`. Consider the case where the application provides the same object (via a `hid_t` wrapper around `H5VL_object_t`) to two threads calling different API functions. In this case, it would be possible for an Async VOL callback operation in one thread to modify the structure while an Async VOL callback operation in another thread reads from it.
- The Log-based VOL, similarly to the Async VOL, defines its wrap context to contain object information retrieved from the provided object. Since that object may be provided to multiple API functions by the application, and thus may be concurrently modified by different Log-based VOL operations, this wrap context is not threadsafe.

C.8 VOL Connector

This structure represents a VOL connector. It provides the callbacks that implement a VOL's operations, as well as metadata about the VOL connector such as connector version, name, and its unique identifier value.

```
typedef struct H5VL_class_t {
    /* Overall connector fields & callbacks */
    unsigned           version;          /**< VOL connector class struct version number */
    H5VL_class_value_t value;            /**< Value to identify connector                */
    const char        *name;             /**< Connector name (MUST be unique!)           */
    unsigned           conn_version;     /**< Version number of connector                */
    uint64_t           cap_flags;        /**< Capability flags for connector             */
    herr_t (*initialize)(hid_t vipl_id); /**< Connector initialization callback          */
    herr_t (*terminate)(void);           /**< Connector termination callback             */

    /* VOL framework */
    H5VL_info_class_t info_cls; /**< VOL info fields & callbacks  */
    H5VL_wrap_class_t wrap_cls; /**< VOL object wrap / retrieval callbacks */

    /* Data Model */
    H5VL_attr_class_t     attr_cls;     /**< Attribute (H5A*) class callbacks */
    H5VL_dataset_class_t  dataset_cls;  /**< Dataset (H5D*) class callbacks   */
    H5VL_datatype_class_t datatype_cls; /**< Datatype (H5T*) class callbacks  */
    H5VL_file_class_t     file_cls;     /**< File (H5F*) class callbacks      */
    H5VL_group_class_t    group_cls;    /**< Group (H5G*) class callbacks     */
    H5VL_link_class_t     link_cls;     /**< Link (H5L*) class callbacks      */
    H5VL_object_class_t   object_cls;   /**< Object (H5O*) class callbacks    */

    /* Infrastructure / Services */
    H5VL_introspect_class_t introspect_cls; /**< Container/conn introspection cls callbacks */
    H5VL_request_class_t    request_cls;    /**< Asynchronous request class callbacks */
    H5VL_blob_class_t       blob_cls;       /**< 'Blob' class callbacks */
    H5VL_token_class_t      token_cls;      /**< VOL connector object token cls callbacks */

    /* Catch-all */
    herr_t (*optional)(void *obj, H5VL_optional_args_t *args, hid_t dxpl_id,
                       void **req); /**< Optional callback */
} H5VL_class_t;
```

During the duration of its registration with the library, each field on this structure should be constant. Clearly the struct version, connector name, unique connector value, connector version, and connector capability flags should not change during runtime. The rest of the fields on this structure are pointers to callbacks, which should also not be replaced during runtime.

The only field which seems potentially modifiable during operation is the `size` field on `info_cls`, which describes the size of the buffer containing connector-specific information to be stored on the FAPL. However, an examination of all officially registered VOL connectors that make use of connector-specific information (passthrough VOL connector, Async VOL, cache VOL, LOG-based VOL, DAOS VOL, dset split VOL, PDC VOL, LowFive) shows that the info size is assigned the of a VOL-defined information struct, making it a constant after registration.