H5CX Multi-Thread Safety Analysis

Matthew Larson

December 19, 2024

Contents

1 Introduction

This document is an overview of the necessary changes (as of HDF5 1.14.2) to make the H5CX module thread-safe. Some modules which H5CX makes use of, such as H5I and H5VL, have existing thread-safe design proposals and implementations which will be referenced when relevant.

1.1 Assumptions

This document makes a number of assumptions about the implementation of multi-threading within the library.

- The multi-threaded rework of H5I described in this document and implemented in this branch is in use.
- The multi-threaded rework of H5VL described in the Threadsafe H5VL design document is in use.
- The H5P module will either acquire the global mutex on entry, or will be reworked to have a multi-thread safe design.

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 Appendix C.1 for a full description of this structure. Fields on the API context control aspects of the operation, deciding things such as which what layout a dataset has, or whether a metadata read is performed collectively. At the end of an API routine, the API 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.

A 'context field' is a value accessible through the API context. The API context struct has at least one member for each field accessible through the API context interface. Based on how the stored value is populated and accessed, context fields may be organized into five categories: references to property lists, internal fields, cached fields, return-only fields, and return-and-read fields.

3 The API Context Stack

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

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

Most API calls initialize and allocate a new context node on entry using a macro of the form `FUNC_ENTER_API_*`. The new context node is pushed onto the top of the context stack by `H5CX_push()`. At the end of the API call, a macro of the form `FUNC_LEAVE_API_*` frees the topmost context node and removes it from the context stack with `H5CX_pop()`.

3.1 Thread Locality

The API context module was designed and implemented with the future possibility of threadsafety in mind. For that reason, when the library is built with threadsafety enabled, each thread has its own instance of the API context stack, accessed through the thread-global `H5TS_apictx_key_g`. This key is used to set and retrieve the context stack via `H5TS_(get/set)_thread_local_value()`.

The thread local key for the head of the API context stack (as well as 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) that is 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 most potential for inter-thread conflict and greatly simplifies the task of making H5CX multi-thread safe.

4 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 (i.e. property lists) that the user provided.

The '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 Appendix C.3 for a full description of this struct.

A context state object 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, likely because the context state was created before these property lists were added to the API context. As there are no known use cases for storing these fields on the context state at present, there are no plans to add them to the context state.

4.1 Context State Lifecycle

The library defines five operations involving the context state: start, retrieve, restore, finish, and free. Because these operations are meant to operate on the API Context itself, they do not create a context node on entry and release that node on exit like most other library API routines.

'Start state' creates a new context node on top of the stack that is not released at the end of the API routine. This context node will persist until 'finish state' or library shutdown.

'Retrieve state' allocates and populates a context state object, passing it to the application as an untyped buffer. This state object is passed to 'restore state' in order to populate the persistent context node that was allocated by 'start state'. The state object's memory is released by 'free state'.

These routines are only intended to be used from within VOL callbacks, not directly by applications. Attempts to use them directly from applications will encounter unexpected results from the library's automatic API Context node creation overshadowing the context nodes modified by these operations.

4.2 Invalid Context State After State Release

The library allows the 'free state' routine that releases the resources allocated for the context state object to be called before or after the 'finish state' routine that deallocates the context node. If 'free state' is called before 'finish state', then the context node may contain pointers to invalid memory. For example, if the context state was the last reference to the VOL Connector Property, then the stored pointer to the property's information buffer may point to invalid memory after the state release.

These invalid fields aren't accessible by most API routines, since most routines allocate a new context node on entry, and thus won't have the chance to access the older node containing the invalid pointers. Library routines which don't create a new API context on entry should not attempt to access the context anyway, since those routines have no reason to expect a context node to exist. As such, this presents no problems in the library at present.

4.3 Changes for Multi-Thread Safety

No changes to the API Context State are strictly necessary for multi-thread safety.

4.3.1 Potential Approach to Context Invalidation on Context State Release

While no changes to the Context State are necessary, it may be beneficial to invalidate the API Context's pointers upon context state free to make invalid memory access less likely to occur. This would be possible, but introduce a two-way dependence between contexts and the context state. It is not currently planned for implementation.

The following changes would prevent pointers to invalid memory from being left behind by a context state release:

- Add a new field to the context state object to store a reference to one or more context nodes.

  This would be used to find the node with the pointers to erase at context state release time. This field must be able to list multiple nodes, since there is no constraint restricting the 'state restore' action from being performed multiple times with the same state object. As such, one state release might leave multiple nodes with invalid pointers.

- Add a new field to the API Context structure (H5CX_t) to store a reference to the context state that was used to populate this structure, if any.

  This field is necessary to remove the pointer to the node from the state object in the event that the node is released. Otherwise, at state free time, the state might attempt to access an already-released node.

5 Cached Default Property Lists

During the initialization of the API context module at library startup, the property values from the default property lists are copied into H5CX-global 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 one of these structures. This was likely done to avoid poor performance from H5P operations on default property lists.

5.1 Modifying 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.

5.2 Changes for Multi-Thread Safety

While not strictly required, it would be beneficial going forward for the library to prevent modification of default property lists by API routines. This should rule out a large category of potential error that applications could encounter.

6 References to Property Lists

Several API Context fields are the IDs of or direct pointers to property lists. The underlying property list objects are not copied by the API Context. The API Context may store references to a DXPL, LCPL, LAPL, DCPL, DAPL, and FAPL.

6.1 Modification of Property Lists

The property lists which the API Context saves pointers to may be modified during library operations. The library generally attempts to avoid modifying property lists internally during API routines for which this is not the specific goal, but there are instances where it occurs. See the H5P callback census for more information.

6.1.1 Threadsafety Analysis

While the API Context is threadlocal, it is possible for one property list to be shared across multiple threads in a valid manner. As such, multiple threads' API Contexts may contain references to the same property list.

The routines that populate the references to property lists in the API Context (`H5CX_set_(dxpl/dcpl/lcpl/lapl)()`) are threadsafe as long as the property lists are reference counted properly by the application when shared between threads.

Reading Values If H5P resides under the global mutex, then it will not be possible for concurrent operations on the same property or property list to occur through H5CX. This is because when a cached field is read from the API Context, either a previously stored cached value is read, or the value is retrieved through `H5P_peek()`/`H5P_get()`. If a cached value is read, then the properly list itself is not accessed in any way. If the value is read through `H5P_peek()`/`H5P_get()`, then the global mutex acquisition on H5P entry will prevent unsafe concurrent access.

Cached Values After Modification When a property list property value is replaced/updated, the old value is freed by that property's assigned delete callback. In some cases, this can cause the cached value in the API Context to point to invalid memory.

Consider as an example the external file prefix property (H5D_ACS_EFILE_PREFIX_NAME). This property has a dynamically allocated string value. When the value is requested through the API Context, a pointer to this dynamic string value is stored as a cached field. If the API routine goes on to either replace the prefix property with a new value through `H5P_set()` or directly invokes its delete callback, then the memory buffer is released. The API Context's cached value is not updated, and if it were to be queried after this, it would point to invalid memory.

This cannot happen if the library API is used correctly. The library generally does not modify the user-provided property list except in H5P API operations where that is the explicit goal (e.g. H5Pset_*). (For the exceptions to this pattern from H5CX, see the sections on return-only and return-and-read fields.) As such, in most cases, the API Context does not need to worry about property values being released out from under it. Within the library itself, there is no explicit mechanism to prevent accessing an invalid buffer through H5CX.

There are some instances where the library internally modifies application-provided property lists at unexpected times. However, these all occur either under the global mutex, or with features that are not planned for multi-thread support at this time (e.g. subfiling, MPIO). In all cases, the invalidated field is not later queried before the end of the API operation, so no threadsafety issues exist. See the H5P callback census for more information.

6.1.2 Changes for Multi-Thread Safety

It would be possible to introduce a set of cached field invalidation callbacks to prevent even the possibility of invalid memory access through the API context. The design for this set of callbacks is detailed in Section 6.1.3. However, this is not planned for implementation for the following reasons:

- After the multi-threaded H5P rework has been implemented, the API Context will refer to a constant instance of a property list. Previous property values will no longer be released until the context itself is released, and this issue will no longer exist. As such, the cached field invalidation callbacks would be at best an interim solution.
- This system would introduce a direct dependence on H5CX to H5P, which should be avoided when possible
- The callbacks are a complex and likely performance-impacting solution to a problem that can be avoided entirely by careful behavior within the library itself.

6.1.3 Potential Approach to Context Invalidation on Property Release

In keeping with the pattern established for other H5CX operations, each cached field would have a new `H5CX_invalidate_<field_name>()` routine associated with it.

Potentially-populated API context cached fields would need to be invalidated on the following property operations:

- Property Set, `H5P__set_plist_cb`
- Property 'Poke' (Internal library variant of property set), (H5P__poke_plist_cb)
- Property Delete, `H5P__del_plist_cb`

A new structure would be needed to establish a mapping between property names and their associated cache invalidation routines, so that the correct cached field can be invalidated at property release time. This would most likely be a hash table. This hash table would be private to H5P, created during H5P initialization, and updated during the registration of each individual property with a property list class (`H5P__register_real()`).

Since this hash table would be private to the global-mutex locked H5P, the hash table from `uthash` would suffice.

7 Internal Fields

"Internal Fields" on the API Context 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.

7.1 The VOL Connector Property

7.1.1 Overview

The VOL Connector Property consists of a VOL Class ID and a connector-defined information buffer. The API Context treats it as an internal field not associated with any property list, even though the VOL connector property is sometimes set on a FAPL. This is because the property value is drawn from a VOL object and not associated with any property list during most operations.

The VOL connector property field is only used to streamline setting the VOL connector on internal file objects during file open and file creation. As the API Context is private, this access point to the connector property is not exposed to applications or VOL connector operations.

While the regular getter/setter routines only shallow copy the connector property to/from the API Context, the routines that create and tear down an API Context State object perform a deep copy and free on the connector property.

7.1.2 Threadsafety Analysis

It is possible for two concurrent file create/open operations, provided with the same FAPL containing an application-defined VOL connector property, to read from the same connector property through the API Context at the same time.

VOL Class ID The API Context does not increment the reference count of the VOL Class ID when the connector property is shallow copied to the context. The context relies on the parent object of the VOL connector property (either a property list or a VOL object) to be reference counted correctly by the application, which should prevent any attempts to release the VOL Class ID while it is in use by the context.

When the VOL Connector Property is attached to a new file struct, the ref count of the VOL Class ID is incremented. As such, no threadsafety issues exist. (While the ordering of ref count operations is potentially unsafe in HDF5 1.14.2, this is resolved by the multi-threaded implementation of H5VL).

The VOL class ID ref count management during API Context State operations depends on the multi-threaded H5I implementation to be threadsafe.

Connector Information Buffer The connector information buffer is not copied at property list access time, when the property is cached in the API Context, or when it is retrieved from the API Context. As such, any client that retrieves the connector property from the API Context must copy the information buffer before sharing the property.

The only current read from the API Context's VOL Connector Property field is `H5F__set_vol_conn()`, which attaches the connector property to a file struct. This routine properly copies the information buffer, so no threadsafety issues exist.

The information buffer release/copy operations during API Context State management are threadsafe due to the constraints imposed on connector information by the multi-threaded H5VL rework.

7.1.3 Changes for Multi-Thread Safety

This field depends on the multi-threaded implementation of H5VL and H5I in order to be threadsafe, but requires no changes in the API Context.

7.2 The VOL Wrapping Context

7.2.1 Overview

The VOL wrapping context is used by passthrough VOL connectors to pass objects up and down the VOL connector stack. This field is a pointer to a VOL wrapping object which consists of a reference count, a pointer to a connector instance managed by the H5VL module, and a connector-managed object wrapping context buffer. (See Appendix C.7 for more information on this structure.)

The connector instance is managed and reference-counted by H5VL. The object wrapping context buffer is only introspected by connector callbacks which cannot access the API context.

Routines which set the wrapping context field on the API Context must first retrieve the wrapping context from the VOL Connector through a connector-defined callback.

The VOL Wrapping Context is deep copied to the API Context State at state retrieval time by increasing its reference count through H5VL. Similarly, it is released at state release time by decrementing its reference count.

7.2.2 Threadsafety Analysis

The VOL wrapping object is not deep copied when set on or retrieved from the API Context. However, the multi-threaded H5VL rework introduces the constraint that connectors must either have a read-only wrapping context, or that each retrieval of the VOL wrapping context should allocate a new buffer. As such, each API Context node will operate on an independent or constant wrapping context buffer, eliminating the possibility of dangerous concurrent access.

The multi-threaded H5VL rework also handles the connector instances and the wrapping object reference count in a threadsafe manner.

The reference count management during creation and release of a context state object are threadsafe due to the multi-thread H5VL rework.

7.2.3 Changes for Multi-Thread Safety

This field depends on the multi-threaded implementation of H5VL in order to be threadsafe, but requires no changes in the API Context.

8 Cached Fields

A 'cached field' on the API Context is a pointer to a value that originates in a property list. Cached fields are used to short-circuit reads from property lists whenever possible, because read operations on property lists can be quite slow. Most fields on the API Context are cached fields.

At read time, 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's corresponding 'valid' flag (a field of the form `<field_name>_valid`) is set to hopefully speed up later operations.

8.1 Invalid Cached Fields

See Section 6.1.1 for a description of how cached fields may be made invalid by property list modification, why this is not considered a major issue for a multi-thread safe H5CX implementation.

8.2 Variable-Length Memory Management

8.2.1 Overview

The Dataset Transfer Property List (DXPL) has a set of fields that control memory allocation for variable-length datatypes. These fields consist of an allocation routine, an information buffer for the allocation routine, a free routine, and an information buffer for the free routine. These fields allow the user to define their own memory management routines to replace the default malloc and free. These four fields may be collectively cached in the context through the `vl_alloc_info` field.

8.2.2 Threadsafety Analysis

The allocation information buffer and free information buffer may be modified during variable length memory allocation or memory release. However, these routines are only used during Native VOL operations which reside under the global mutex. As long as the application follows the API constraint of not modifying the buffer after providing it to the library, only one thread can access either buffer at a given time.

It is possible for multiple API Context nodes in separate threads to point to the same variable-length allocation buffers, if the property list containing them is provided to multiple threads. However, the global mutex above the native VOL prevents any sort of potentially dangerous concurrent access.

9 Return-only Fields

Return-only fields correspond to properties from property lists that are never read internally by the library and are only set by API Context operations in order to provide information back 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 point, any changed return-only fields have their new values written into the provided property list.

Return-only fields avoid modifying the provided property list if it is the generic default property list (H5P_DEFAULT) or the default for a particular property list class (e.g. H5P_DATASET_XFER_DEFAULT). This is done to uphold the general pattern of not modifying default property lists.

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.

9.1 Threadsafety Analysis

At API exit time, the API Context modifies the provided property list in order to return these values to the application. The modified property list may be shared between threads by the applications. However, since the property list modification is mediated by H5P, which is either under the global mutex or has a multi-thread safe rework, no unsafe memory access or writes are possible.

The only issue is that the value read by an application from a property list's return-only fields after using two concurrent API operations will be determined by a race condition. This is not ideal, but is not considered a problem for multiple reasons. First, the return-only fields are primarily used for debugging and are not widespread in general library use. Secondly, if accurate values in these fields are necessary during concurrent operations, the application has the option of duplicating the property list and providing a copy to each concurrent operation, preventing the conflict.

10 Return-and-read Fields

Return-and-read fields on the API Context 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 the underlying property list's corresponding property, as subsequent reads will always use a cached value.

Note that it is possible to overwrite the cached value via an `H5CX_set_<field>()` operation, but not possible to overwrite the cached value with a `H5CX_get_<field>()` 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 setter 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 its setter. 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 its setter. The context will then contain a value that is not from the underlying property list, but `<field>_valid` is true.

10.1 Threadsafety Analysis

Because the API Context is threadlocal, the internal library reads of return-and-read context values are threadsafe.

The analysis of potential invalid memory accesses from within the library for cached fields also applies to return-and-read fields.

The analysis regarding race conditions on return values from return-only context fields also applies to return-and-read fields.

10.2 Changes for Multi-Thread Safety

The cached field invalidation callbacks described in section 6.1.3 would also be applied to these fields, if implemented. However, for the reasons described in the section on cached fields, these callbacks are not currently planned for implementation.

11 Miscellaneous

11.1 Dependencies on Other Modules

A multi-thread safe implementation of H5CX requires a threadsafe implementation of H5VL and H5I.

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

Other internal library modules, especially H5P, should acquire the global mutex on entry.

11.2 Context Getters with Potentially Shared Return Buffers

Many of the context's getter routines store their result in a buffer provided by the caller. If the caller provides a buffer that resides on an object shared between multiple threads, then these getters might perform a non-threadsafe write.

The responsibility for preventing bad usage of the API Context in this manner falls on its client modules, not the API Context itself.

A survey of the current library indicated no cases where any of these routines are used in an unsafe manner.

- `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()

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 threadsafe as long as the multi-threaded H5I implementation is in use, and the application has correctly reference counted the property list.

  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 threadsafe since H5P will either acquire the global mutex or have a thread-safe rework.

  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 threadsafe since H5P will either acquire the global mutex or have a thread-safe rework.

- `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 threadsafe since H5P will either acquire the global mutex or have a thread-safe rework.

- `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 threadsafe since H5P will either acquire the global mutex or have a thread-safe rework.

- `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 operations are threadsafe since H5P will either acquire the global mutex or have a thread-safe rework.

Similarly, the H5VL and H5I operations are threadsafe for the multi-thread implementations of those modules.

- `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.

  The ID operations are threadsafe in the multi-thread H5I implementation. The H5VL resource release routines are safe in the multi-thread H5VL implementation.

- `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 threadsafe, since H5P either acquires the global mutex or has a multi-thread safe rework.

  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`. Threadsafe as long as the property list is reference counted by the application correctly.

  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`. Threadsafe as long as the property list is reference counted by the application correctly.

- **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`. Threadsafe as long as the property list is reference counted by the application correctly.

  This function is currently unused within the library.

  hid_t H5CX_get_lapl(void) - Retrieves the LAPL ID for the current API context. Threadsafe as long as the property list is reference counted by the application correctly.

- **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. Threadsafe as long as the property list is reference counted by the application correctly.

  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`.

  Threadsafe as long as the application does not modify a property list that it provides to multiple API routines concurrently.

- **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`.

  Threadsafe as long as the application does not modify a property list that it provides to multiple API routines concurrently.

- **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`.

  Threadsafe as long as the application does not modify a property list that it provides to multiple API routines concurrently.

- **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`.

  Threadsafe as long as the application does not modify a property list that it provides to multiple API routines concurrently.

- **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.

  Threadsafe as long as the application does not modify a property list that it provides to multiple API routines concurrently.

- **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.

  Threadsafe as long as the application does not modify a property list that it provides to multiple API routines concurrently.

- **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.

  Threadsafe as long as the application does not modify a property list that it provides to multiple API routines concurrently.

- **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.

  Threadsafe as long as the application does not modify a property list that it provides to multiple API routines concurrently.

- **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.

  Threadsafe as long as the application does not modify a property list that it provides to multiple API routines concurrently.

- **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.

Threadsafe as long as the application does not modify a property list that it provides to multiple API routines concurrently.

- **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.

  Threadsafe as long as the application does not modify a property list that it provides to multiple API routines concurrently.

- **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.

  Threadsafe as long as the application does not modify a property list that it provides to multiple API routines concurrently.

- **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.

  Threadsafe as long as the application does not modify a property list that it provides to multiple API routines concurrently.

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.

  Threadsafe as long as the application does not modify a property list that it provides to multiple API routines concurrently.

- **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.

  Threadsafe as long as the application does not modify a property list that it provides to multiple API routines concurrently.

- **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.

Threadsafe as long as the application does not modify a property list that it provides to multiple API routines concurrently.

- **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.

  Threadsafe as long as the application does not modify a property list that it provides to multiple API routines concurrently.

- **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.

  Threadsafe as long as the application does not modify a property list that it provides to multiple API routines concurrently.

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.

  Threadsafe as long as the application does not modify a property list that it provides to multiple API routines concurrently.

- **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.

  Threadsafe as long as the application does not modify a property list that it provides to multiple API routines concurrently.

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.

  Threadsafe as long as the application does not modify a property list that it provides to multiple API routines concurrently.

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.

Threadsafe as long as the application does not modify a property list that it provides to multiple API routines concurrently.

- **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.

  Threadsafe as long as the application does not modify a property list that it provides to multiple API routines concurrently.

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.

  Threadsafe as long as the application does not modify a property list that it provides to multiple API routines concurrently.

- **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.

  Threadsafe as long as the application does not modify a property list that it provides to multiple API routines concurrently.

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. Threadsafe as long as the application does not modify a property list that it provides to multiple API routines concurrently.

  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.5 VOL Fields

The context's handling of VOL fields is made threadsafe by the multi-thread H5VL rework.

- **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 locally threadsafe with respect to modification of the property.

  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.

  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.

- **vol_wrap_ctx** - The VOL connector's 'wrap context' used by passthrough VOL connectors to wrap VOL objects. Does not correspond to a property in a property list.

  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.

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

  Unlike cached property list fields, if no valid value is found on the API context, then this returns a NULL value.

B.6 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.

  Threadsafe as long as the application does not modify a property list that it provides to multiple API routines concurrently.

- **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.

  Threadsafe as long as the application does not modify a property list that it provides to multiple API routines concurrently.

- **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.

  Threadsafe as long as the application does not modify a property list that it provides to multiple API routines concurrently.

- **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.

  Threadsafe as long as the application does not modify a property list that it provides to multiple API routines concurrently.

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 unwrapping this VOL's own wrapping structure, and then 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 unwrapped object. 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.