Census of H5P Callbacks

Matthew Larson

Overview

This document is a census of property callbacks and property class callbacks in the HDF5 library as of 1.14.4.3, for the purpose of identifying potential issues with the implementation of threadsafety in the H5P module.

This document attempts to list every unique property callback, along with comments about its dependencies and implications for threadsafety, if any. A section on internal library modification of properties and on context modification of properties is also included, since these create threadsafety concerns.

Property list modules which have no unique callbacks (MAPL, LCPL, and FMPL) do not have their own sections here.

Property Callback Overview

Each property (instance of `H5P_genprop_t`) has up to nine unique callbacks assigned to it when it is registered to a property list class via `H5P__register_real()`. Each of these callbacks is optional, although properties which are objects with their own internal memory allocation require unique get/set/copy/create and del/close callbacks to properly implement the copy-by-value semantics expected of property values.

- Create - `herr_t H5P_prp_create_func_t(const char *name, size_t size, void *value)`

  This callback should set up the initial value of the property by modifying the provided `value` buffer. This is necessary when the property is a complex object that cannot be deep copied by a single `memcpy()`. `size` describes the size of value, and `name` is the name of the property being created.

  `value` is a shallow copy of the initial property value provided to `H5P__register_real()`. If this callback returns a negative value, then the potentially modified value is not copied into the property and the creation routine returns an error.

  The initialization done by this callback may consist of simply deep copying the initial value. This deep copy may be implemented via reference counting (as seen in `H5P__facc_file_driver_create()` and `H5P__facc_vol_create()`), or as a 'real' copy with new memory allocation for each dynamically allocated field of the property value. The memory management method this callback uses to enable copy-by-value semantics must be cleaned up during the delete and free callbacks assigned to the same property.

The original dynamically allocated fields under `value`, if any, should not be freed or modified, since these fields are still in use by either the property list class or the original property list. An exception to this is that if reference counting is used to implement copy-by-value, then the underlying fields must be modified to update their reference count.

This callback is invoked in two places by the library: During the creation of a new property list in `H5P__create()`, and when copying a property from one plist to another plist that does not already contain it in `H5P__copy_prop_plist()`. (If the target plist for a copy operation does already contain the property, the copy callback is used instead.)

- Set - `herr_t H5P_prp_set_func_t(hid_t prop_id, const char *name, size_t size, void *value)`

  This callback should modify `value` as necessary for the set operation to follow copy-by-value semantics for the property. This callback is necessary when the value is a complex object with its own internal dynamic memory allocation. This callback may also perform a transformation on the property value, if the internal representation differs from the representation visible to the user.

  `prop_id` is the ID of the property list being modified. `name` is the name of the property being modified. `value` is a shallow copy of the provided value to write. `size` is the size of the buffer value. If this callback returns a negative value, the potentially modified value is not copied into the property and the set routine returns an error.

  If performing a deep copy, the set callback should either allocate new memory for the dynamically allocated fields of the property value, or ’fake’ copy them using reference counting - see `H5P__facc_file_driver_set()` and `H5P__facc_vol_set()` as examples. The memory management method this callback uses to enable copy-by-value semantics must be cleaned up during the delete and free callbacks assigned to the same property.

  If no error occurs, the modified value buffer is copied to the target property after this callback finishes.

  The original dynamically allocated fields under `value`, if any, should not be freed or modified, since these fields are still in use by the application. An exception to this is that if reference counting is used to implement copy-by-value, then the underlying fields must be modified to update their reference count.

  The set callback is used to set the value of a property in a list by `H5P__set_plist_cb()`, and to set the value of a property in a class by `H5P__set_pclass_cb()`.

If the set callback is not defined, the property read operation defaults to a simple `memcpy()` from the application buffer to the property value buffer.

- Get - `herr_t H5P_prp_get_func_t(hid_t prop_id, const char *name, size_t size, void *value)`

  This callback should modify `value` as necessary for the get operation to follow copy-by-value semantics for the property. This is necessary when the property value is a complex object with its own internal dynamic memory allocation. The get callback may also perform a transformation on the property value before providing it to the user, if the representation visible to the user differs from how it is stored in the library.

  `prop_id` is the ID of the property list being queried. `name` is the name of the property being queried. `value` is a shallow copy of the property value that will eventually be returned to the application. `size` is the size of the buffer value. If this returns a negative value, then the user's buffer is not modified and the get routine returns an error.

  If performing a deep copy, the get callback should either allocate new memory for the dynamically allocated fields of the property value, or 'fake' copy them using reference counting - see `H5P__facc_file_driver_get()` and `H5P__facc_vol_get()`. The memory management method this callback uses to enable copy-by-value semantics must be cleaned up during the delete and free callbacks assigned to the same property.

  The original dynamically allocated fields under `value`, if any, should not be freed or modified, since these fields are still in use by the property itself. An exception to this is that if reference counting is used to implement copy-by-value, then the underlying fields must be modified to update their reference count.

  If no error occurs, the modified value buffer is copied to the application buffer by `H5P__get_cb()`.

  If this callback is not defined, the read operation defaults to a simple `memcpy()` from the property's value to the application buffer.

- Copy - `H5P_prp_copy_func_t(const char *name, size_t size, void *value)`

  This callback should modify `value` as necessary for copy-by-value semantics to be upheld when copying this property between property lists. This is necessary when the property value is a complex object that is not fully copied by a single `memcpy()` call.

  `name` is the name of the property being copied. `value` is a shallow copy of the original property value. `size` is the size in bytes of value. If this callback succeeds, then `value` is copied to the new property in the destination property list.

  If this callback returns a negative value, the potentially modified value is not copied into the destination plist and the copy routine returns an error.

This callback may implement a deep copy by copying any allocated fields stored under value, or 'fake' such copying by using reference-counted fields. The memory management method this callback uses to enable copy-by-value semantics must be cleaned up during the delete and free callbacks assigned to the same property.

Note that this callback is used when copying an entire property list, and when copying a property to another list that already contains a property of the same name, but not when copying a property to another list that does not contain a property of the same name. In this last case, the create callback is used instead.

The original dynamically allocated fields under `value`, if any, should not be freed or modified, since these fields are still in use by the original property. The exception to this is that if reference counting is used to implement copy-by-value, then the underlying fields must be modified to update their reference count.

- Encode - `int H5P_prp_encode_func_t(const void *value, void **buf, size_t *size))`

  This callback is used to encode the property value value into the application-allocated buffer `*buf`. `size` describes the size of the destination buffer `*buf`. If the provided buffer is NULL, or if the provided size is zero, then the encode callback should modify `size` to return the necessary buffer size for the encoded value.

  Unlike decode, the encode callback should not increment the provided value pointer after encoding.

- Decode - `H5P_prp_decode_func_t(const void **buf, void *value)`

  This callback is used to decode the encoded property value in `*buf` to the library-allocated buffer `value`.

  The decode callback must increment the pointer `*buf` by the size of the encoded value. This is the reason `buf` is a is provided as a `void**`. This incrementing is necessary for `H5P__decode()` to iterate through all properties in an encoded property list.

- Delete - H5P_prp_delete_func_t(hid_t prop_id, const char *name, size_t size, void *value)

  This callback should clean up any callback-controlled resources under `value` that were allocated during create, set, or copy. It is invoked when a property is deleted from a property list or class, or when the value of a property is replaced by a set operation. The top-level `value` buffer itself should not be freed, as the library frees that buffer during generic property free operations.

  `prop_id` is the ID of the property list the property is being deleted from. `name` is the name of the property being deleted. `value` is the value of the property which is being deleted. `size` is the size of `value`.

If this callback returns a negative value, then an error is returned, but the target property is still deleted.

- Close - `herr_t H5P_prp_close_func_t(const char *name, size_t size, void *value)`

  This callback should clean up any callback-controlled resources under `value` that were allocated during create, set, or copy. This callback is invoked when a property list containing this property is destroyed.

  `name` is the name of the property being closed. `value` is a buffer containing the value of the property being closed. `size` is the size of the buffer value.

  The top-level `value` buffer itself should not be freed, as the library frees that buffer during generic property free operations.

  If this callback returns a negative value, the property list close operation returns an error, but the property list is still closed.

- Compare - `int H5P_prp_compare_func_t(const void *value1, const void *value2, size_t size)`

  This callback should return a positive value if `value1` >`value2`, a negative value if `value2` >`value1`, or zero if `value1` = `value2`. Neither input value should be modified.

  This callback is only the final step of the property comparison operation `H5P__cmp_prop()`. Before this callback is used, the property's names, sizes, and callbacks are compared. If any of these fields are nonequal, the comparison returns early and this callback is not used. If two properties are nonequal due to one not defining a callback which the other property does define, the property which defines the callback is considered greater. If two properties provide different implementations of the same callback, then the first property is considered smaller.

There is substantial overlap in how these callbacks are usually implemented for a single property. Create, set, get, and copy frequently act as wrappers around a copy method for the object type of the property. This custom copy method is necessary for more complex objects that cannot be entirely copied with a simple `memcpy()` from generic H5P routines. For example, consider the property for external file prefixes. The external file prefix's value is a pointer to a pointer to a string, `char**`. The library get routine only copies the intermediate pointer to the external file prefix (`char*`). The get callback is used to copy the underlying string using `strdup()`. The same principle applies to the set, create, and copy callbacks. Similarly, the delete and close callbacks for a property are often implemented as wrappers around the same underlying free function.

The library's default properties and their callbacks can be broadly separated into the following categories:

- Properties with callbacks which use object message callbacks. The get, set, copy, and create callbacks are wrappers around `H5O_msg_copy()`, and the del/close callbacks are wrappers around `H5O_msg_reset()`. The properties in this category are:
  - Dataset Layouts
  - Fill Values
  - External File Lists
  - Object Filter Pipelines
- Properties with callbacks which use module-specific routines to implement their operations. For most of these properties, the get, set, copy, and create callbacks are wrappers around another module's object copy routine, and the del/close callbacks are wrappers around another module's object free routine. Encode and decode callbacks may be wrappers around external encode/decode callbacks, or may perform their work directly.

  Most of these properties either use only threadsafe callbacks, or use callbacks from a module which is planned for a threadsafe implementation.

  The properties which fall into this category are:
  - File Image Info, dependent on H5P and potentially user-defined file image callbacks
  - Data Transformations dependent on H5Z
  - Dataset I/O Selections dependent on H5S
  - File Driver ID and Info, dependent on H5P
  - VOL Connectors dependent on H5VL
  - MPI Communicators dependent on H5mpi
  - MPI Information objects, dependent on H5mpi
  - External Link FAPLs, dependent on H5P
  - Merge Committed Datatype Lists, dependent on H5P.
- Properties with callbacks which have no significant external dependencies besides H5MM and H5VM for memory management. Since the H5MM calls are wrappers to system memory calls, these properties are considered threadsafe.
- Properties with only encode/decode callbacks. These callbacks may be unique for this property, or generic type encode/decode functions defined in H5Pencdec.c. All property callbacks of this type are threadsafe.

- Properties with no defined callbacks. These are typically properties that cannot be encoded because they depend on local context (e.g. type conversion background buffer information). Most Context Return Properties - properties used by the library only to pass values back to the application program - fall into this category.
- Test properties used during library testing in `tgenprop.c`. These modify potentially shared application-level resources in a non-threadsafe way in order to verify that the callbacks were executed as expected.

Most threadsafety concerns come from properties that object message callback, callbacks with dependencies on other library modules, and context return properties.

Property Class Callback Overview

Each property list class (instance of `H5P_genclass_t`) has up to three unique callbacks.

- Create - `herr_t H5P_cls_create_func_t(hid_t prop_id, void *create_data)`

  This callback is invoked when a property list of the given class is created. `prop_id` is the identifier of the property list being created. `create_data` is a pointer to a buffer of application-defined data stored on the parent class of `prop_id`.

  This callback may modify `create_data`, or perform application-defined initialization work on the list `prop_id`. If this callback allocates any resources under `create_data`, then those resources should be released by the corresponding property class close callback.

  If this callback returns a negative value, then the new list is not returned to the user and the property list creation routine returns an error.

  When this callback is invoked, it is invoked for every property list class in the class hierarchy of the list parent class, starting from the immediate parent class and proceeding until the root class.

  If this callback modifies `create_data`, then it is not threadsafe due to modifying a resource which may be accessed by other threads performing plist operations concurrently.

  `create_data` is not copied by the library; the buffer passed in by the application is used directly. If this buffer is dynamically allocated, releasing it is the responsibility of the application.

- Copy - `herr_t H5P_cls_copy_func_t(hid_t new_prop_id, hid_t old_prop_id, void *copy_data)`

  This callback is invoked when copying a property list of the given class. `new_prop_id` is the identifier of the newly created property list copy. `old_prop_id` is the id of the list being copied. `copy_data` is a pointer to application-defined data on the class.

This callback may modify `copy_data`, or it may perform work on the new list or original list. If this callback allocates resources under `copy_data`, then those resources must be released by the corresponding property class close callback.

If this callback returns a negative value, the new list is not returned to the user, and the property list copy function returns an error value.

When this callback is invoked, it is invoked for every property list class in the class hierarchy of the list parent class, starting from the immediate parent class and proceeding until the root class.

If this callback modifies `copy_data` or `old_prop_id`, then it is not threadsafe due to modifying a resource which may be accessed by other threads performing plist operations concurrently.

`copy_data` is not copied by the library; the buffer passed in by the application is used directly. If this buffer is dynamically allocated, releasing it is the responsibility of the application.

- Close - `herr_t H5P_cls_close_func_t(hid_t prop_id, void *close_data)`

  This callback is invoked when a property list of the given class is closed. `prop_id` is the ID of the property list being closed. `close_data` is a pointer to application-defined data on the property list class.

  This callback should release any resources that were allocated under the class's create or copy callbacks.

  If this callback modifies `close_data`, then it is not threadsafe due to modifying a resource which may be accessed by other threads concurrently.

  When this callback is invoked, it is invoked for every property list class in the class hierarchy of the list parent class, starting from the immediate parent class and proceeding until the root class.

  `close_data` is not copied by the library; the buffer passed in by the application is used directly. If this buffer is dynamically allocated, releasing it is the responsibility of the application.

Each callback has a corresponding data field in the property list class. This data is defined at class creation time and provided as a parameter to each callback.

At the time of this document's creation (HDF5 1.14.4.3), the library does not define any of these callbacks on any of its predefined property list classes.

If the test code for the property list class callbacks (`test_genprop_class_callback` in `tgenprop.c`) is indicative of the design intent, then these callbacks may be intended to let users associate reference-counted data with property list classes. Property list create, copy, and close operations would then reference shared data on the class object, and would not be threadsafe if the operations potentially modify that data.

Known Threadsafety Issues

- File Image Info - The property callbacks for File Image Info objects use the file image's memory management callbacks, which may be user-defined callbacks that use reference counting to manage shared buffers. If this is the case, multiple threads using supposedly distinct FAPLs could have race conditions accessing and modifying an underlying shared buffer. The default file image callbacks used by the Core VFD are threadsafe, but the file image callbacks defined for the high-level H5LT interface are not.
- External Link FAPLs - The property callbacks for External Link FAPLs can perform operations on each property within the FAPL, which may include the File Image Info property, so external link FAPLs inherit the potential threadsafety problems of file images.
- Dataset Layouts - The property callbacks for Dataset Layouts use object message callbacks that depend on skip lists, which are not threadsafe.
- Context Return Properties - These properties are potentially modified by the API context upon API routine exist. Since the value buffer is directly modified, in a multithreaded scenario where multiple threads operate with the same property list, this leads to a race condition where the final value of the property depends upon the order the threads execute in, and this could in principle allow reading of partial or torn writes.
- The MPI-I/O driver's open callback `H5FD__mpio_open()` modifies the MPI Info property on the provided FAPL, which can create a race condition when using the same FAPL to open different files.
- The subfiling VFD's open callback `H5FD__subfiling_open()` sets the metadata cache configuration property. The size of the cache configuration struct means that, if two threads using the same FAPL open different files, a non-atomic write could be interrupted by another thread, leading to a malformed buffer.
- Test Callbacks - The callbacks in the library's tests for property and property list class callbacks use a reference-counted application-level resource, which could lead to race conditions.
- Free Lists are used directly or indirectly by several properties. Because the current intention is to disable free lists in the threadsafe build of the library, this is not a critical issue.

Points of Interest

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

- The File Driver ID and information property's del and close callbacks use `H5P__file_driver_free()`, in which the free callback `H5FD_free_driver_info()` is invoked to free driver info before the reference count of the driver id is decremented with `H5I_dec_ref()`. This does not seem to be a race condition, since operations in H5Dfapl don't expect `driver_info` to always be defined even if `driver_id` exists.

  The corresponding copy routine used for this property's other callbacks should be threadsafe, since it increments the reference count before acquiring the ID, which should allow graceful failure in the event that another thread modifies or deletes the object between the two H5I calls (though this behavior should be prevented).

- The same pattern as above applies to the VOL connector property's del and close callbacks, which are wrappers around `H5VL_conn_free()`. The connector info is freed before the reference count of the connector ID is decremented, but other H5VL callbacks treat this as a consistent state.

- Return-Only Properties' global initialization - The return-only properties used by `H5CX__pop_common()` are all initialized with pointers to global values. However, property creation allocates new memory for the given value, so later modifications to these properties do not interact with the global value.

- `H5P_set_driver(), H5_open_subfiling_stub_file()` and `H5FD_subfiling_set_file_id_prop()` sometimes set a property on a FAPL from within an operation where this is potentially unexpected (generally a file driver's open or close callback). But in each of these cases, the property is set to a fixed value, and so the property value will be the same no matter which thread finishes first in a multi-threaded scenario. As such, these are not currently considered a concern for a multithreaded implementation.

Context Return Fields

The API context contains fields called 'return-only properties' and 'return-and-read properties'. If these properties are set on the context, then the corresponding property for each field is directly modified by H5CX upon API context exit, right before an API routine completes. In a multithreaded scenario where multiple threads operate with the same property list, this leads to a race condition where the final value of the property depends upon the order the threads execute in. It also creates the possibility for torn writes if threads' writes interrupt each other, although that is not believed to be possible for any of the fields/properties listed here, due to their values being only a few bytes in size.

All properties corresponding to these context fields use the default library property callbacks.

- `mpio_actual_chunk_opt` - Indicates the chunk optimization mode used during parallel I/O. Its initial value is taken from the global enum `H5D_def_mpio_actual_chunk_opt_mode_g`. Not threadsafe due to potential modification of shared property list.

- `mpio_actual_io_mode` - Indicates the actual I/O mode used for an operation, which may differ from what the application requested. Its initial value is taken from the global enum `H5D_def_mpio_actual_io_mode_g`. Not threadsafe due to potential modification of shared property list.

- `mpio_local_no_coll_cause` - Indicates the (local) cause of broken collective I/O. Its initial value is a pointer to the global enum `H5D_def_mpio_no_collective_cause_g`. Not threadsafe due to potential modification of shared property list.

- `mpio_global_no_coll_cause` - Indicates the (global) cause of broken collective I/O. Its initial value is a pointer to the global enum `5D_def_mpio_no_collective_cause_g`. Not threadsafe due to potential modification of shared property list.

- `no_selection_io_cause` - The cause for not performing selection or vector I/O on the last parallel I/O call. Its initial value is a pointer to the global enum `H5D_def_no_selection_io_cause_g`. Not threadsafe due to potential modification of shared property list.

- `actual_selection_io_mode` - The selection I/O mode actually used for an operation. Its initial value is a pointer to the H5P-local enum `H5D_def_actual_selection_io_mode_g`. Not threadsafe due to potential modification of shared property list.

Library Instrumenting Fields

If the library is built with instrumenting of internal operations for debugging purposes (`H5_HAVE_INSTRUMENTED_LIBRARY`) then the context treats these fields as normal return-only fields by attempting to use them to set the corresponding properties.

However, the corresponding properties for these fields are never defined in the library itself, and so the set operation defaults to a no-op. The properties are defined only during certain tests (e.g. `compact_dataset` in `t_mdset.c`). The definitions listed here are those provided by the library tests.

The writes from these fields to their corresponding properties are non-threadsafe for the same reasons as the other context return fields.

- `mpio_coll_chunk_link_hard` - 'Collective chunk link hard' value. Not threadsafe due to potential modification of shared property list.

  - `mpio_coll_chunk_multi_hard` - ’Collective chunk multi hard’ value. Not threadsafe due to potential modification of shared property list.
  - `mpio_coll_chunk_link_num_true` - ’Collective chunk link num true’ value. Not threadsafe due to potential modification of shared property list.
  - `mpio_coll_chunk_link_num_false` - ’Collective chunk link num false’ value. Not threadsafe due to potential modification of shared property list.
  - `mpio_coll_chunk_multi_ratio_coll` - ’Collective chunk multi ratio collective’ value. Not threadsafe due to potential modification of shared property list.
  - `mpio_coll_chunk_multi_ratio_ind` - ’Collective chunk multi ratio independent’ value. Not threadsafe due to potential modification of shared property list.
  - `mpio_coll_rank0_bcast` - ’Collective rank 0 broadcast’ value. Not threadsafe due to potential modification of shared property list.

Internal Modification of Properties

If the library internally sets a property value on a property list that is exposed to the user from a module that does not lie under the global mutex, the value of the property at any given time depends on the order the threads complete, similar to context return properties.

This section lists all places where the library internally sets a property value using `H5P_set()` or `H5P_poke()` within modules planned for threadsafety, and describes the threadsafety issues or lack thereof. Modules not planned for threadsafety are excluded from this census, since the global mutex should prevent race conditions in those cases.

`H5P_set()` Usage

- `H5CX.c` - The threadsafety issues introduced by Context Return Properties as described in the previous section.

- `H5FDmpio.c` - The MPI-I/O driver’s open callback `H5FD__mpio_open()` sets the MPI Info property on the provided FAPL to a value dependent on the particular file opened. If multiple files are opened with the same FAPL, then the MPI Info the application reads from the FAPL afterwards is dependent on a race condition between threads, and so is not threadsafe.

  Specifically, regardless of whether or not the input FAPL provides MPI Info, the actual MPI Info used to open the file (which may differ from the provided MPI Info) is used to populate the property, and a so race condition can exist between different threads using the same FAPL to open different files.

- `H5Pdxpl.c` - `H5P_set_vlen_mem_manager()` is only used from `H5Dint`, which will reside under the global mutex. Additionally, it only modifies a temporary DXPL that is freed at the end of an internal routine, so it would be threadsafe even if the calling module was not under a mutex. All other property sets in this module are a result of API property set calls.

- `H5Pfapl.c` - `H5P_set_driver` is used by each API call that enables a certain driver on a FAPL. It is used internally in the following functions:

  - `H5P__facc_set_def_driver()`, which is only used to set up the default FAPL during library initialization.

  - `H5P_set_driver_by_name()`, only used by the API function of the same name.

  - `H5P_set_driver_by_value()`, which is used internally by the Family VFD and the Splitter VFD, where it is used to set the FAPLs for distinct files to the default sec2 driver. Each of these VFDs can potentially set this field during file open and file delete operations. This is technically not threadsafe, but since the property is always set to a fixed value (the `H5FD_class_value_t` value of the default driver), this is not considered a significant issue.

  - `H5_open_subfiling_stub_file()`, which is only invoked by the Subfiling VFD's open callback to enable the MPI-I/O driver. This is technically not threadsafe, but since it always sets the property to a fixed value it is not considered a significant issue.

  All other property sets in `H5Pfapl.c` are a result of API property set calls.

- `H5FDsubfiling.c` - The metadata cache configuration property is set during the Subfiling VFD's open callback `H5FD__subfiling_open()`. If the same FAPL is used in multiple threads, all threads must return the same value for this property. However, the size of the value (`H5AC_cache_config_t`) creates the possibility for a partial write to occur, leading to malformed memory. As such, these operations are not threadsafe. All other property sets in this module are a result of API property set calls.

- `H5subfiling_common.c` - Properties are set in the following functions:

  - `H5_open_subfiling_stub_file()` - Modifies a newly created property list, and so no race condition is potentially exposed to the application.

  - `H5_subfiling_set_config_prop()` - Invoked as a direct result of the API call `H5P_set_fapl_subfiling()`.

  - `H5_subfiling_set_file_id_prop()` - Invoked during the Subfiling VFD's open callback. Sets the stub file ID on the provided FAPL. This is technically non-threadsafe, but since it always sets the target property to the same value (`H5FD_SUBFILING_STUB_FILE_ID`), it is not considered a significant issue.

- `H5P.c, H5Pdapl.c, H5Pdcpl.c, H5Pfcpl.c, H5Pgcpl.c, H5Plapl.c, H5Plcpl.c, H5Pmapl.c, H5Pocpl.c, H5Pocpypl.c, H5Pstrcpl.c,` - All property set operations in these modules are a result of API-level property set operations, and the burden of using them in a threadsafe manner falls on the application.

`H5P_poke()` usage

- `H5Pdcpl.c, H5Pdxpl.c, H5Pfapl.c, H5Pocpypl.c` - All poke operations in these modules are a result of API-level property set operations, and the burden of using them in a threadsafe manner falls on the application.

- `H5Pencdec.c` - `H5P__decode()` uses `H5P_poke()` to set the values in a newly decoded property list that is not yet available to the application and cannot be shared between threads, so this usage is threadsafe.

- `H5Pocpl.c` - `H5P_modify_filter()` uses `H5P_poke()`. While the full tree of functions that use it is somewhat complex, each execution paths begins from either a function that creates a new plist internally, or is called from an API module that uses the global lock upon entry.

DAPL Callbacks

DAPL Property Callbacks

These callbacks are defined in `H5Pdapl.c`. The properties they belong to are chunk cache parameters, virtual dataset views, virtual dataset file prefixes, and external file prefixes. The only dependencies on other library modules are trivial invocations of the `H5VM` and `H5MM` API, all of which are threadsafe.

Virtual dataset file prefix callbacks

- `H5P__dapl_vds_file_pref_set` - Wrapper around `strdup`.

- `H5P__dapl_vds_file_pref_get` - Wrapper around `strdup`.

- `H5P__dapl_vds_file_pref_enc` - Encodes virtual dataset file prefix into provided buffer. Uses H5VM and H5MM.

- `H5P__dapl_vds_file_pref_dec` - Decodes virtual dataset file prefix from provided buffer. Uses H5MM to allocate space for the decoded value.

- `H5P__dapl_vds_file_pref_del` - Wrapper around `free`.

- `H5P__dapl_vds_file_pref_copy` - Wrapper around `strdup`.

- `H5P__dapl_vds_file_pref_cmp` - Wrapper around `strcmp`.

- `H5P__dapl_vds_file_pref_close` - Wrapper around `free`.

External file prefix callbacks

- `H5P__dapl_efile_pref_set` - Wrapper around `strdup`.

- `H5P__dapl_efile_pref_get` - Wrapper around `strdup`.
- `H5P__dapl_efile_pref_enc` - Encodes the external file prefix. Uses H5VM and H5MM.
- `H5P__dapl_efile_pref_dec` - Decodes the external file prefix. Uses H5MM to allocate space for the decoded value.
- `H5P__dapl_efile_pref_del` - Wrapper around `free`.
- `H5P__dapl_efile_pref_copy` - Wrapper around `strdup`.
- `H5P__dapl_efile_pref_cmp` - Wrapper around `strcmp`.
- `H5P__dapl_efile_pref_close` - Wrapper around `free`.

Encode/Decode callbacks

- `H5P__encode_chunk_cache_nslots` - Encodes number of chunk slots in the raw data chunk cache into provided buffer. Uses H5VM.
- `H5P__decode_chunk_cache_nslots` - Decodes number of chunk slots in the raw data chunk cache from provided buffer.
- `H5P__encode_chunk_cache_nbytes` - Encodes size of the raw data chunk cache into provided buffer. Uses H5VM.
- `H5P__decode_chunk_cache_nbytes` - Decodes size of the raw data chunk cache from provided buffer.
- `H5P__dacc_vds_view_enc` - Encodes a virtual dataset view (`uint8_t`) into provided buffer.
- `H5P__dacc_vds_view_dec` - Decodes a virtual dataset view (`uint8_t`) from a provided buffer.

DCPL Property Callbacks

These callbacks are found in `H5Pdcpl.c`. The properties they belong to are object storage layouts, fill values, external file lists, space allocation time, and object headers. Space allocation time and object header property callbacks use only generic encoding/decoding functions defined in `H5Pencdec.c`.

The property callbacks for object layouts, fill values, and external file lists invoke object message class callbacks from `H5O_MSG_LAYOUT`, `H5O_MSG_FILL`, and `H5O_MSG_EFL`. Each of these object message classes has several callbacks, but only 'copy' and 'reset' are ever used by these property callbacks.

Due to an indirect dependence on skip lists (and free lists, though those can be disabled) several layout property callbacks are not threadsafe.

Dataset layout callbacks

This property's callbacks act as wrappers around `H5O_msg_copy()` and `H5O_msg_reset()`.

The object layout message copy callback `H5O__layout_copy()` depends on H5D due to `H5D_chunk_idx_reset()` and `H5D__virtual_copy_layout()`.

`H5D__virtual_copy_layout()` depends on the H5SL, H5FL, H5S, and H5I modules. If a failure occurs during the virtual layout copy, then the routine used to clean up allocated resources (`H5D__virtual_reset_layout()`) uses `H5D__virtual_reset_source_dset()`, which in turn uses `H5D_close()`. Even if free lists are disabled at configure time, use of skip lists in `H5D_close()` is not threadsafe, and so `H5O__layout_copy()` and the property callbacks that use it are not threadsafe. `H5D_close()` also interacts with the metadata cache via `H5AC_cork()` and `H5AC_flush_tagged_metadata()`, though the potential threadsafety ramifications of these calls has not been deeply examined.

`H5D_chunk_idx_reset()` uses the reset callback `H5D_chunk_reset_func_t` from `H5D_chunk_ops_t`, which has a distinct implementation for B-Trees, v2 B-Trees, extensible arrays, fixed arrays, non-indexed chunks, and single chunk operations. At the time of this census, each of these reset callbacks is threadsafe and extremely simple.

The object layout reset callback `H5O__layout_reset()` also indirectly depends on `H5D_close()` in the same manner as the object layout copy callback, and so it is also not threadsafe.

- `H5P__dcrt_layout_set` - Copies layout via `H5O_msg_copy()`, not threadsafe due to `H5SL`.
- `H5P__dcrt_layout_get` - Copies layout via `H5O_msg_copy()`, not threadsafe due to `H5SL`.
- `H5P__dcrt_layout_enc` - Encodes layout property. Uses H5S.
- `H5P__dcrt_layout_dec` - Decodes layout property. Uses H5S and threadsafe H5D routines.
- `H5P__dcrt_layout_del` - Frees layout via `H5O__layout_reset()`, not threadsafe due to `H5SL`.
- `H5P__dcrt_layout_copy` - Copy layout via `H5O_msg_copy()`, not threadsafe due to `H5SL`.
- `H5P__dcrt_layout_cmp` - Compare two layout properties. Uses H5S.
- `H5P__dcrt_layout_close` - Frees layout via `H5O__layout_reset`, not threadsafe due to `H5SL`.

Dataset fill value callbacks

This property's callbacks are wrappers around the object message callbacks `H5O_msg_copy()` and `H5O_msg_reset()`.

The fill value object message copy callback (`H5O__fill_copy()`) uses H5T routines to deal with potential type conversion. This involves reading from and potentially writing to the global type conversion path table `H5T_g`. `H5T_g` is local to the H5T module, which exists under the global mutex, so these operations should be threadsafe. There is also a dependence on H5CX through `H5T_convert()`.

The fill value reset callback `H5O__fill_reset()` is similar to `H5O_fill_copy()`, and is also threadsafe.

- `H5P__dcrt_fill_value_set` - Copies fill value via `H5O_msg_copy()`.
- `H5P__dcrt_fill_value_get` - Copies fill value via `H5O_msg_copy()`.
- `H5P__dcrt_fill_value_enc` - Uses `H5T_encode`, which in turn depends on H5FL. Datatype message encoding callback may reference a shared object message, but it should be threadsafe due to the global mutex.
- `H5P__dcrt_fill_value_dec` - Uses `H5O_msg_decode()`. Dependency on H5T, H5F, and non-threadsafe dependence on H5FL if free lists are enabled.

External File List callbacks

This property's callbacks are wrappers around the object message callbacks `H5O_msg_copy()` and `H5O_msg_reset()`.

The external file list copy and reset callbacks (`H5O__efl_copy()` and `H5O__efl_reset()`) only depend on H5MM and are both threadsafe.

- `H5P__dcrt_ext_file_list_set` - Wrapper around `H5O_msg_copy()`.
- `H5P__dcrt_ext_file_list_get` - Wrapper around `H5O_msg_copy()`.
- `H5P__dcrt_ext_file_list_enc` - Encodes the external file list. Uses H5VM and H5MM.
- `H5P__dcrt_ext_file_list_dec` - Decodes the external file list. Uses H5MM.
- `H5P__dcrt_ext_file_list_del` - Wrapper around `H5O_msg_reset()`.
- `H5P__dcrt_ext_file_list_copy` - Wrapper around `H5O_msg_copy()`.
- `H5P__dcrt_ext_file_list_cmp` - Compares two external file lists.
- `H5P__dcrt_ext_file_list_close` - Wrapper around `H5O_msg_reset()`.

DXPL Property Callbacks

These property callbacks are found in `H5Pdxpl.c`. The most significant properties are data transformations and dataset I/O selections. Other properties in this module only have encode/decode callbacks.

The data transformation property callbacks act as wrappers around `H5Z` functions. Because `H5Z` doesn't read or write any global structures, these callbacks are threadsafe.

The dataset I/O selection callbacks act as wrappers around `H5S` functions. Since `H5S` has a threadsafe implementation planned, these callbacks are considered threadsafe.

Data Transformation Property Callbacks

- `H5P__dxfr_xform_set` - Wrapper around `H5Z_xform_copy()`.
- `H5P__dxfr_xform_get` - Wrapper around `H5Z_xform_copy()`.
- `H5P__dxfr_xform_enc` - Encodes a data transform. Has a threadsafe dependency on H5Z and H5VM.
- `H5P__dxfr_xform_dec` - Decodes a data transform. Wrapper around `H5Z_xform_create()`
- `H5P__dxfr_xform_del` - Wrapper around `H5Z_xform_destroy`.
- `H5P__dxfr_xform_copy` -Wrapper around `H5Z_xform_copy()`.
- `H5P__dxfr_xform_cmp` - Compares two data transforms. Uses a threadsafe H5Z call.
- `H5P__dxfr_xform_close` - Wrapper around `H5Z_xform_destroy()`.

Dataset I/O Selection Property Callbacks

- `H5P__dxfr_dset_io_hyp_sel_copy` - Wrapper around `H5S_copy()`.
- `H5P__dxfr_dset_io_hyp_sel_cmp` - Compares two dataset I/O selections. Depends on H5S comparison routines.
- `H5P__dxfr_dset_io_hyp_sel_close` - Wrapper around `H5S_close()`.

Encode/Decode Callbacks

- `H5P__dxfr_bkgr_buf_type_enc` - Encodes the background buffer type.
- `H5P__dxfr_bkgr_buf_type_dec` - Decodes the background buffer type.
- `H5P__dxfr_btree_split_ratio_enc` - Encodes the B-tree split ratio. Depends on H5P routines.
- `H5P__dxfr_btree_split_ratio_dec` - Decodes the B-tree split ratio. Depends on H5P routines.

- `H5P__dxfr_io_xfer_mode_enc` - Encodes the I/O transfer mode.
- `H5P__dxfr_io_xfer_mode_dec` - Decodes the I/O transfer mode.
- `H5P__dxfr_mpio_collective_opt_enc` - Encodes the MPI-I/O collective optimization.
- `H5P__dxfr_mpio_collective_opt_dec` - Decodes the MPI-I/O collective optimization.
- `H5P__dxfr_mpio_chunk_opt_hard_enc` - Encodes the MPI-I/O chunk optimization.
- `H5P__dxfr_mpio_chunk_opt_hard_dec` - Decodes the MPI-I/O chunk optimization.
- `H5P__dxfr_edc_enc` - Encodes the error detect property.
- `H5P__dxfr_edc_dec` - Decodes the error detect property.
- `H5P__dxfr_selection_io_mode_enc` - Encodes selection I/O mode.
- `H5P__dxfr_selection_io_mode_dec` - Decodes selection I/O mode.
- `H5P__dxfr_modify_write_buf_enc` - Encodes the modify write buffer.
- `H5P__dxfr_modify_write_buf_dec` - Decodes the modify write buffer.

FAPL Property Callbacks

These property callbacks are found in `H5Pfapl.c`. The properties they belong to are file driver ID and information, file image info, cache configuration, metadata cache log location, metadata cache image property, VOL connector, MPI communicator, and MPI info.

File Driver ID and Information Callbacks

The create, set, get, and copy callbacks are all wrappers around the in-place copy operation `H5P__file_driver_copy`. Delete and close are wrappers around `H5P__file_driver_free`, which is a wrapper around `H5FD_free_driver_info`. The comparison callback uses `H5FD`, which in turn depends on `H5I` and `H5P`. Since all of these dependent modules are planned for threadsafe implementation, the comparison function is also threadsafe.

In `H5P__file_driver_copy()`, the reference count of the driver id is incremented before it is retrieved via `H5I_object()`. This ordering should avoid race conditions once H5I is threadsafe, and also the library to fail gracefully if another thread modifies the reference count or closes the driver ID between the two H5I calls.

In `H5P__file_driver_free`, the free callback `H5FD_free_driver_info()` is invoked to free driver info before the reference count of the driver id is decremented with `H5I_dec_ref()`. This does not seem to be a race condition, since operations in H5Dfapl don't expect `driver_info` to always be defined even if `driver_id` exists.

- `H5P__facc_file_driver_create` - Wrapper around `H5P__file_driver_copy()`.

- `H5P__facc_file_driver_set` - Wrapper around `H5P__file_driver_copy()`.
- `H5P__facc_file_driver_get` - Wrapper around `H5P__file_driver_copy()`.
- `H5P__facc_file_driver_del` - Wrapper around `H5P__file_driver_free()`.
- `H5P__facc_file_driver_copy` - Wrapper around `H5P__file_driver_copy()`.
- `H5P__facc_file_driver_cmp` - Compares two sets of file driver ID and info. Depend on `H5FD_get_class()`, which is assumed to be threadsafe due to planned threadsafe implementation for H5FD.
- `H5P__facc_file_driver_close` - Wrapper around `H5P__file_driver_free()`.

File Image Info Callbacks

The set, get, and copy operations are all wrappers around `H5P__file_image_info_copy()`. This shared copy function uses callbacks defined on the file image info struct (`H5FD_file_image_info_t`): `image_malloc()`, `image_memcpy()`, and `copy_udata()`.

The delete and close operations are wrappers around `H5P__file_image_info_free()`. This shared free function uses the file image info callback `image_free()`.

These file image memory callbacks default to being wrappers around the threadsafe `malloc`, `memcpy`, and `free`. However, the file image API was designed to allow application programs to use their own file image callbacks which provide the illusion of allocating new memory while actually re-using buffers internally in order to improve performance. If such a set of callbacks is used, then these callbacks deal with a resource shared between the application and the library, and are potentially non-threadsafe.

The library itself contains only two implementations of file image each memory management callback: one in the high-level HDF Lite module (H5LT), and a default implementation which uses the corresponding system memory call. The H5LT implementations are as follows:

- `H5LT.image_malloc()` - Takes a parameter to decide how to manage the buffer. Depending on parameters, new memory may not actually be allocated. May be a no-op, the target buffer may be ’copied’ via reference counting to a FAPL from an application buffer, or ’copied’ to an application buffer from a FAPL. Not threadsafe due to manipulation of a shared buffer.
- `H5LT.image_memcpy()` - Takes a parameter to decide how to manage the buffer. Depending on parameters, may be a no-op, a reference counted ’copy’ from application buffer to FAPL, or a reference counted ’copy’ from FAPL to the application buffer. Not threadsafe due to manipulation of a shared buffer.
- `H5LT.image_realloc()` - Takes a parameter to decide how to manage the buffer. May be a no-op, or may use `realloc()` on the underlying buffer. Not threadsafe due to manipulation of a shared buffer.

- `H5LT.image_free()` - Takes a parameter to decide how to manage the buffer. May act as a no-op, a reference-decrementing 'free', or an actual free if the ref count of the target buffer drops to zero. Also invokes `udata_free()` Not threadsafe due to manipulation of a shared buffer.
- `H5LT.udata_copy()` - Takes a parameter to decide how to manage the buffer. Either a no-op, or increases the reference count of the targeted data without allocating new memory. Not threadsafe due to manipulation of a shared buffer.
- `H5LT.udata_free()` - Takes a parameter to decide how to manage the buffer. Either a no-op or a reference-decrementing 'free' depending on parameters. Not threadsafe due to manipulation of a shared buffer.

Note that the last two file image callbacks - `udata_copy()` and `udata_free()` - have no default implementation. They are required only if the `udata` field on the file image is populated.

The Core VFD, a primary use case for the file image interface, uses the default memory callbacks, making this property threadsafe for the Core VFD.

- `H5P__facc_file_image_info_set` - Wrapper around `H5P__file_image_info_copy()`. Not threadsafe due to file image callbacks potentially being not threadsafe.
- `H5P__facc_file_image_info_get` - Wrapper around `H5P__file_image_info_copy()`.Not threadsafe due to file image callbacks potentially being not threadsafe.
- `H5P__facc_file_image_info_del` - Wrapper around `H5P__file_image_info_free()`.Not threadsafe due to file image callbacks potentially being not threadsafe.
- `H5P__facc_file_image_info_copy` - Wrapper around `H5P__file_image_info_copy()`.Not threadsafe due to file image callbacks potentially being not threadsafe.
- `H5P__facc_file_image_info_cmp` - Compares two sets of file image information. Not threadsafe due to file image callbacks potentially being not threadsafe.
- `H5P__facc_file_image_info_close` - Wrapper around `H5P__file_image_info_free()`.Not threadsafe due to file image callbacks potentially being not threadsafe.

Cache Configuration Callbacks

These callbacks depend only on H5MM and H5VM.

- `H5P__facc_cache_config_enc` - Encodes the cache configuration to a plist.
- `H5P__facc_cache_config_dec` - Decodes the cache configuration from a plist.

- `H5P__facc_cache_config_cmp` - Compares two cache configurations.

Metadata Cache Log Location Callbacks

The metadata cache log location is a string, and these callbacks are mostly wrappers around system string and memory operations. These depend on H5VM and H5MM.

- `H5P__facc_mdc_log_location_enc` - Encodes the metadata cache log location to a plist.
- `H5P__facc_mdc_log_location_dec` - Decodes the metadata cache log location from a plist. Uses H5MM to allocate memory for decoded value.
- `H5P__facc_mdc_log_location_del` - Wrapper around `free`.
- `H5P__facc_mdc_log_location_copy` - Wrapper around `strdup`.
- `H5P__facc_mdc_log_location_cmp` - Wrapper around `strdup`.
- `H5P__facc_mdc_log_location_close` - Wrapper around `free`.

Cache Image Configuration Callbacks

These callbacks use no functions from other modules.

- `H5P__facc_cache_image_config_cmp` - Compares two cache image configurations.
- `H5P__facc_cache_image_config_enc` - Encodes a cache image configration to a plist.
- `H5P__facc_cache_image_config_dec` - Decodes a cache image configuration.

VOL Connector Callbacks

The create, set, get, and copy callbacks are wrappers around `H5VL_conn_copy`. The delete and close callbacks are wrappers around `H5VL_conn_free`. The compare callback uses H5I and H5VL routines. Because these modules are planned for threadsafe implementations, these callbacks are considered threadsafe.

`H5VL_conn_copy()` increments the reference count before acquiring the ID via `H5I_object`, making the use of H5I threadsafe (Assuming the H5I module itself is made internally threadsafe).

`H5VL_conn_free()` frees connector information before decrementing the ref count of the ID. While another thread would be able to access the connector between these two calls, other H5VL callbacks independently check for the existence of `connector_info`, so this shouldn't be a threadsafety concern.

- `H5P__facc_vol_create` - Wrapper around `H5VL_conn_copy`.
- `H5P__facc_vol_set` - Wrapper around `H5VL_conn_copy`.

- H5P__facc_vol_get - Wrapper around H5VL_conn_copy.
- H5P__facc_vol_del - Wrapper around H5VL_conn_free.
- H5P__facc_vol_copy - Wrapper around H5VL_conn_copy.
- H5P__facc_vol_cmp - Compares two sets of VOL connector ID and info. Depends on H5I and H5VL.
- H5P__facc_vol_close - Wrapper around H5VL_conn_free.

MPI Communicator Callbacks

These callbacks act as wrappers around H5mpi.c functions, which in turn make use of MPI routines. Get, set, and copy callbacks all use H5_mpi_comm_dup, delete and close callbacks both use H5_mpi_comm_free.

All MPI routines used are either guaranteed threadsafe, or threadsafe as long as the MPI object they modify is not being operated on by another thread - preconditions which should always hold due to the global mutex and/or the user application logic.

- H5P__facc_mpi_comm_set - Wrapper around H5_mpi_comm_dup().
- H5P__facc_mpi_comm_get - Wrapper around H5_mpi_comm_dup().
- H5P__facc_mpi_comm_del - Wrapper around H5_mpi_comm_free().
- H5P__facc_mpi_comm_copy - Wrapper around H5_mpi_comm_dup().
- H5P__facc_mpi_comm_cmp - Wrapper around H5_mpi_comm_cmp().
- H5P__facc_mpi_comm_close - Wrapper around H5_mpi_comm_free().

MPI Info Callbacks

Like the MPI Communicator callbacks, these callbacks are wrappers around H5mpi.c functions, which are in turn wrappers around MPI routines. Just as for those callbacks, all MPI routines used are threadsafe or threadsafe as long as the target MPI object is not externally modified during operation.

The set, get, and copy callbacks are wrappers around H5_mpi_info_dup. The delete and close callbacks are wrappers around H5_mpi_info_free.

- H5P__facc_mpi_info_set - Wrapper around H5_mpi_info_dup().
- H5P__facc_mpi_info_get - Wrapper around H5_mpi_info_dup().
- H5P__facc_mpi_info_del - Wrapper around H5_mpi_info_free().
- H5P__facc_mpi_info_copy - Wrapper around H5_mpi_info_dup().

- `H5P__facc_mpi_info_cmp` - Wrapper around `H5_mpi_info_cmp()`.
- `H5P__facc_mpi_info_close` - Wrapper around `H5_mpi_info_free()`.

Encode/Decode Callbacks

None of these callbacks use routines from any other module.

- `H5P__facc_fclose_degree_enc` - Encodes file close degree
- `H5P__facc_fclose_degree_dec` - Decodes file close degree
- `H5P__facc_multi_type_enc` - Encodes multi VFD memory type
- `H5P__facc_multi_type_dec` - Decodes multi VFD memory type
- `H5P__facc_libver_type_enc` - Encodes a library version bound
- `H5P__facc_libver_type_dec` - Decodes a library version bound
- `H5P__encode_coll_md_read_flag_t` - Encodes the collective metadata read flag
- `H5P__decode_coll_md_read_flag_t` - Decodes the collective metadata read flag

FCPL Property Callbacks

These property callbacks are found in `H5Pfcpl.c`. This module contains only custom encode/decode callbacks. None of these callbacks use any external routines.

Encode/Decode Callbacks

- `H5P__fcrt_btree_rank_enc` - Encodes the minimum rank of a btree internal node
- `H5P__fcrt_btree_rank_dec` - Decodes the minimum rank of a btree internal node
- `H5P__fcrt_shmsg_index_types_enc` - Encodes the shared message index types
- `H5P__fcrt_shmsg_index_types_dec` - Decodes the shared message index types
- `H5P__fcrt_shmsg_index_minsize_enc` - Encodes the shared message index minimum size
- `H5P__fcrt_shmsg_index_minsize_dec` - Decodes the shared message index minimum size
- `H5P__fcrt_fspace_strategy_enc` - Encodes the free-space strategy.
- `H5P__fcrt_fspace_strategy_dec` - Decodes the free-space strategy

GCPL Property Callbacks

These property callbacks are found in `H5Pgcpl.c`. This module contains only custom encode/decode callbacks. None of these callbacks use any external routines.

- `H5P__gcrt_group_info_enc` - Encodes group info
- `H5P__gcrt_group_info_dec` - Decodes group info
- `H5P__gcrt_link_info_enc` - Encodes link info
- `H5P__gcrt_link_info_dec` - Decodes link info

LAPL Property Callbacks

These property callbacks are found in `H5Plapl.c`. The properties they belong to are external link prefixes, and external link FAPLs.

External Link Prefix Callbacks

These callbacks have only trivial dependencies on H5VM and H5MM routines.

- `H5P__lacc_elink_pref_set` - Wrapper around `strdup()`.
- `H5P__lacc_elink_pref_get` - Wrapper around `strdup()`.
- `H5P__lacc_elink_pref_enc` - Encodes the external link prefix. Uses H5VM and H5MM.
- `H5P__lacc_elink_pref_dec` - Decodes the external link prefix. Uses H5MM.
- `H5P__lacc_elink_pref_del` - Wrapper around `free()`.
- `H5P__lacc_elink_pref_copy` - Wrapper around `strdup()`.
- `H5P__lacc_elink_pref_cmp` - Wrapper around `strcmp()`.
- `H5P__lacc_elink_pref_close` - Wrapper around `free()`.

External Link FAPL Callbacks

These callbacks depend on routines from H5P, and on a threadsafe routine in H5VM. Because H5P has a threadsafe implementation planned, these callbacks are considered threadsafe.

An entire FAPL is stored as a single property for external links. Callbacks which usually copy their property internally (get, set, copy) only do so if the FAPL is non-default, otherwise the callback is a noop. The encode/decode callbacks first serialize to/from a single byte that indicates whether the FAPL is non-default, followed by a serialization of the FAPL itself only if it is non-default.

These callbacks are potentially non-threadsafe, if the external link FAPL contains a file image with callbacks that use reference counting for memory allocation (see File Image Info Callbacks).

- `H5P__lacc_elink_fapl_set` - Duplicates target FAPL if it is non-default, otherwise a no-op. Wrapper around `H5P_copy_plist()`. Not threadsafe due to potential use of non-threadsafe file image callbacks.
- `H5P__lacc_elink_fapl_get` - Duplicates target FAPL if it is non-default, otherwise a no-op. Wrapper around `H5P_copy_plist()`. Not threadsafe due to potential use of non-threadsafe file image callbacks.
- `H5P__lacc_elink_fapl_enc` - Encodes a byte indicating whether the FAPL is non-default, and the entire FAPL if it is non-default. Wrapper around `H5P__encode()`.
- `H5P__lacc_elink_fapl_dec` - Decodes an external link FAPL. Wrapper around `H5P__decode()`.
- `H5P__lacc_elink_fapl_del` - Decreases reference count of FAPL, if it is non-default. Wrapper around `H5I_dec_ref()`. Not threadsafe due to potential use of non-threadsafe file image callbacks.
- `H5P__lacc_elink_fapl_copy` - Duplicates target FAPL if it is non-default, otherwise a no-op. Wrapper around `H5P_copy_plist()`. Not threadsafe due to potential use of non-threadsafe file image callbacks.
- `H5P__lacc_elink_fapl_cmp` - Wrapper around `H5P__cmp_plist()`. Also depends on H5I. Not threadsafe due to potential invocation of `H5P_facc_file_image_info_cmp`. Not threadsafe due to potential use of non-threadsafe file image callbacks.
- `H5P__lacc_elink_fapl_close` - Decreases reference count of FAPL, if it is non-default. Wrapper around `H5I_dec_ref()`. Not threadsafe due to potential use of non-threadsafe file image callbacks.

OCPL Property Callbacks

These callbacks are defined in `H5Pocpl.c`. The only property with callbacks defined in this module is the filter pipeline for object creation.

Filter Pipeline Property Callbacks

This property's callbacks are wrappers around the object message callbacks `H5O_msg_copy()` and `H5O_msg_reset()`.

The set, get, and copy callbacks invoke the object message copy callback for filter pipelines, which is `H5O__pline_copy()`. Besides a dependence on H5FL, this callback is threadsafe, and so the callbacks which use it are threadsafe.

The delete and close callbacks invoke the object message reset callback for filter pipelines - `H5O__pline_reset()`. This callback is threadsafe, so the property callbacks which use it are threadsafe.

This set of callbacks has trivial dependencies on H5VM and H5MM, and a threadsafe dependency on H5Z.

- `H5P__ocrt_pipeline_set` - Wrapper around `H5O_msg_copy()`. Indirect dependence on H5FL.
- `H5P__ocrt_pipeline_get` - Wrapper around `H5O_msg_copy()`. Indirect dependence on H5FL.
- `H5P__ocrt_pipeline_enc` - Encodes the filter pipeline. Depends on H5VM and H5MM.
- `H5P__ocrt_pipeline_dec` - Decodes the filter pipeline. Depends on H5Z, H5VM, and H5MM.
- `H5P__ocrt_pipeline_del` - Wrapper around `H5O_msg_reset()`.
- `H5P__ocrt_pipeline_copy` - Wrapper around `H5O_msg_copy()`. Indirect dependence on H5FL.
- `H5P__ocrt_pipeline_cmp` - Compares two filter pipelines.
- `H5P__ocrt_pipeline_close` - Wrapper around `H5O_msg_reset()`.

OCPYPL Property Callbacks

These callbacks are found in `H5Pocpypl.c`. The only property these callbacks belong to is the merge committed datatype list.

Merge Committed Datatype List Callbacks

Most of these callbacks are wrappers around H5P callback routines for merge committed datatype lists. Besides a dependence on H5FL, these callbacks are threadsafe.

- `H5P__ocpy_merge_comm_dt_list_set` - Wrapper around `H5P__copy_merge_comm_dt_list()`. Depends on H5FL.
- `H5P__ocpy_merge_comm_dt_list_get` - Wrapper around`H5P__copy_merge_comm_dt_list()`. Depends on H5FL.
- `H5P__ocpy_merge_comm_dt_list_enc` - Encodes a merge committed datatype list.
- `H5P__ocpy_merge_comm_dt_list_dec` - Decodes a merge committed datatype list. Depends on H5FL and H5MM.

- `H5P__ocpy_merge_comm_dt_list_del` - Wrapper around `H5P__free_merge_comm_dtype_list()`. Depends on H5FL.
- `H5P__ocpy_merge_comm_dt_list_copy` - Wrapper around `H5P__copy_merge_comm_dt_list()`. Depends on H5FL.
- `H5P__ocpy_merge_comm_dt_list_cmp` - Compares two merge committed datatype lists.
- `H5P__ocpy_merge_comm_dt_list_close` - Wrapper around `H5P__free_merge_comm_dtype_list()`. Depends on H5FL.

H5Pstrcpl Property Callbacks

These callbacks are found in `H5strcpl.c`. This module contains only encode and decode callbacks for character set encodings, which depend on no routines from other modules.

Character Set Encoding Callbacks

- `H5P__strcrt_char_encoding_enc` - Encodes a character set.
- `H5P__strcrt_char_encoding_dec` - Decodes a character set.

Encoding/Decoding Callbacks

These callbacks are defined in `H5Pencdec.c`. They contain only trivial dependencies on H5VM.

- `H5P__encode_size_t` - Encodes a size_t value into a provided buffer.
- `H5P__decode_size_t` - Decodes a size_t value from a provided buffer.
- `H5P__encode_hsize_t` - Encodes an hsize_t value into a provided buffer.
- `H5P__decode_hsize_t` - Decodes an hsize_t value from a provided buffer.
- `H5P__encode_unsigned` - Encodes an unsigned value into a provided buffer.
- `H5P__decode_unsigned` - Decodes an unsigned value from a provided buffer.
- `H5P__encode_uint8_t` - Encodes a uint8_t value into a provided buffer.
- `H5P__decode_uint8_t` - Decodes a uint8_t value from a provided buffer.
- `H5P__encode_bool` - Encodes a boolean value into a provided buffer.
- `H5P__decode_bool` - Decodes a boolean value from a provided buffer.
- `H5P__encode_double` - Encodes a double value to provided buffer.

- `H5P__decode_double` - Decodes a double value from a provided buffer.
- `H5P__encode_uint64_t` - Encode a uint64_t value into a provided buffer.
- `H5P__decode_uint64_t` - Decodes a uint64_t value from a provided buffer.

Test Callbacks

These callbacks are defined in the test module for generic property and property class callbacks, `tgenprop.c`.

Test Property Callbacks

The generic test property callbacks modify a file-local structure `prop1_cb_info`, in order to verify that the correct callbacks have been executed on the correct property lists. Since these functions modify a shared application-level object, they are not threadsafe, although this is not currently an issue since the test is single threaded.

- `test_genprop_prop_crt_cb1` - Increases 'creation count' of shared structure, duplicate the property name, and copies the user-defined value into the shared structure. Not threadsafe due to manipulation of shared object.
- `test_genprop_prop_set_cb1` - Increases 'set count' of shared structure. Stores target property list ID, target property name, and user-defined value in shared structure. Not threadsafe due to manipulation of shared object.
- `test_genprop_prop_get_cb1` - Increases 'get count' of shared structure. Stores target property list ID, target property name, and user-defined value in shared structure. Not threadsafe due to manipulation of shared object.
- `test_genprop_prop_cop_cb1` - Increases 'copy count' on shared structure. Stores target property name and user-defined value in shared struct. Not threadsafe due to manipulation of shared object.
- `test_genprop_prop_cmp_cb1` - Increases 'comparison count' on shared structure. Wrapper around `memcpy`. Not threadsafe due to manipulation of shared object.
- `test_genprop_prop_cls_cb1` - Increases 'close count' of shared structure. Stores traget property name and user-defined value in shared struct. Not threadsafe due to manipulation of shared object.
- `test_genprop_prop_del_cb2` - Increases 'delete count' of shared structure. Stores deleted plist ID in target property name, and user-defined data in the shared structure. Not threadsafe due to manipulation of shared object.
- `test_genprop_prop_cmp_cb3` - Increases 'comparison count' of shared structure. Wrapper around `memcpy`. Not threadsafe due to manipulation of shared object.

Test Property Class Callbacks

Similar to the test property callbacks, these class callbacks increment counters in potentially shared structures to verify that they have been executed during testing. For the same reason, there are potential race conditions when doing operations on the same property list class in multiple threads, and so these callbacks are not threadsafe.

- `test_genprop_cls_crt_cb1` - Increments reference count of creation data, and sets creation data plist pointer to the newly created property list. Not threadsafe due to manipulation of shared object.
- `test_genprop_cls_cpy_cb1` - Increments reference count of copy data, and sets copy data plist pointer to the new copy of the property list. Not threadsafe due to manipulation of shared object.
- `test_genprop_cls_cls_cb1` - Increments the reference count of close data, and sets close data plist pointer to the list being closed. Not threadsafe due to manipulation of shared object.
- `test_genprop_cls_cpy_cb2` - Increments reference count of copy data, and sets copy data plist pointer to the new copy of the property list. Not threadsafe due to manipulation of shared object.