Making H5P Multi-Thread Safe:
A Sketch Design

John Mainzer
Lifeboat, LLC

Draft
02/26/25

Due to the absence of design documentation for the H5P package this document is an attempt to outline the reverse-engineered design of H5P, and to develop a design for a multi-thread safe re-implementation of H5P. For completeness Appendix 3 contains a design document for HDF5 generic properties found on the internet but not included in the current HDF5 documentation maintained by The HDF Group.

Contents:

Introduction

H5P provides support for the property lists used throughout HDF5. It must depend on H5I since property lists are accessed via IDs – instances of hid_t to be precise. Similarly, it must depend on H5E for error reporting. While in principle, there seems no reason why H5P proper should have any other dependencies, it should be no surprise that this is not the case.

More generally, since H5P includes support for getting and setting the various properties associated with the various standard property lists – FCPL (File Creation Property List), FAPL (File Access Property List), etc. – there must be at least minimal interaction with the packages configured with these properties. In principle, this should be limited to structure definitions, constants, and a bit of sanity checking – all items with little or no multi-thread safety significance. While this is certainly true in some cases, at present I don't know the extent to which it is true in general.

This version of the RFC is a large improvement over the previous version, but it is still a work in progress. While I have completed my review of the core property list code, and an initial review of the initialization process for HDF5 library defined property list classes, I have only glanced at the various HDF5 library specific property list class and property specific callbacks – which are a source of the interactions with packages configured via the property lists.

While this deficit must be addressed before we commit to any of the designs outlined in this RFC, I am now in position to discuss the structure of the property list facility in some detail – and have done so in the next section.

This discussion is longer and more detailed than I would like. However, a good understanding of the current implementation is necessary to motivate the solutions proposed.

H5P at Present

In the context of HDF5, a property list is conceptually a list of (<name>, <value>) pairs used to pass configuration data into HDF5 API calls.[1] Given this use case, it is convenient to define property list classes, which facilitate creation of default property lists for a given purpose (i.e., creating an HDF5 file) which have a specified set of (<name>, <value>) pairs. These property lists can then be edited as necessary for the purpose at hand.

Further, new properties can be added to both property list classes and property lists. At the application program level, the major use case for this feature is adding configuration data for externally developed Virtual File Drivers (VFDs) and VOL connectors to the appropriate HDF5 defined property lists.

[1] In a few cases, it is also used to pass data back out to the application program.

H5P exists to provide property list services to the HDF5 library proper, and also to application programs. The basic services may be summarized as follows:

- Create, delete, copy, modify, and compare property list classes, and also iterate through properties in a property list class.

  Here a property list class is best thought of as an archetype for newly created property lists of the desired structure. It specifies the properties that appear in a new property list derived from a target class along with their default values. While the actual implementation is different, conceptually new property list classes are created by duplicating an existing class, and then either adding new properties or overwriting the default values of inherited properties.

- Create, delete, copy, compare, encode, and decode property lists. Get and set the values of properties in a property list. Delete properties from a given property list. Insert new properties into a given property list. Iterate through the properties in a property list.

In terms of internal architecture, the property list code employs a design that echoes a number of ideas from object-oriented programming languages. This design decision creates a number of issues for both the existing single thread implementation and any revision or re-write to add multi-thread support.

H5P Data Structures

There are three main structures used in H5P:

- H5P_genprop_t – used represent properties,

- H5P_genclass_t – used to represent property list classes, and

- H5P_genlist_t – used to represent property lists.

There is also H5P_libclass_t, which is used to construct tables that control the construction of the HDF5 library defined property list classes. As my review of H5P initialization is still in progress, a discussion of this structure will have to wait for the next revision of this RFC. For those who can't wait, there are preliminary notes in Appendix 2.

Properties

The fundamental elements in property lists and property list classes are properties, which are stored in instances of H5P_genprop_t – whose definition is reproduced below.

```
/* Define structure to hold property information */
typedef struct H5P_genprop_t {
   /* Values for this property */
   char *            name;        /* Name of property */
   size_t            size;        /* Size of property value */
   void *            value;       /* Pointer to property value */
   H5P_prop_within_t type;        /* Type of object the property is within */
   hbool_t           shared_name; /* Whether the name is shared or not */

   /* Callback function pointers & info */
   H5P_prp_create_func_t  create; /* Function to call when a property is created */
   H5P_prp_set_func_t     set;    /* Function to call when a property value is set */
   H5P_prp_get_func_t     get;    /* Function to call when a property value is
                                     retrieved */
   H5P_prp_encode_func_t  encode; /* Function to call when a property is encoded */
   H5P_prp_decode_func_t  decode; /* Function to call when a property is decoded */
   H5P_prp_delete_func_t  del;    /* Function to call when a property is deleted */
   H5P_prp_copy_func_t    copy;   /* Function to call when a property is copied */
   H5P_prp_compare_func_t cmp;    /* Function to call when a property is compared */
   H5P_prp_close_func_t   close;  /* Function to call when a property is closed */
} H5P_genprop_t;
```

At the fundamental level, a property consists of a name, a value stored in a buffer, and an optional set of functions to be called on property

- create,
- set,
- get,
- encode,
- decode,
- delete (from a property list),
- copy,
- compare, and
- close (of the host property list).

Since there are no constraints on theses callbacks, they present an obvious risk from a multi-thread perspective. For properties defined by the HDF5 library, this should be manageable, since this code is under the control of the library. Externally defined properties are of greater concern.

The type field is used to note whether the property is contained in a property list class, or a property list proper.

Property List Classes

Property list classes can be thought of as templates for property lists – in essence they specify the properties contained in property lists based on a given property list class, along with their default values[2]. Property list classes are stored in instances of H5P_genclass_t – whose definition is reproduced below.

```
/* Define structure to hold class information */
struct H5P_genclass_t {
   struct H5P_genclass_t *parent;      /* Pointer to parent class */
   char *                 name;        /* Name of property list class */
   H5P_plist_type_t       type;        /* Type of property */
   size_t                 nprops;      /* Number of properties in class */
   unsigned               plists;      /* Number of property lists that have been
                                          created since the last modification to
                                          the class */
   unsigned               classes;     /* Number of classes that have been derived
                                          since the last modification to the class */
   unsigned               ref_count;   /* Number of outstanding ID's open on this
                                          class object */
   hbool_t                deleted;     /* Whether this class has been deleted and is
                                          waiting for dependent classes & proplists
                                          to close */
   unsigned               revision;    /* Revision number of a particular class
                                          (global) */
   H5SL_t               * props;       /* Skip list containing properties */

   /* Callback function pointers & info */
   H5P_cls_create_func_t  create_func; /* Function to call when a property list
                                          is created */
   void *                 create_data; /* Pointer to user data to pass along to
                                          create callback */
   H5P_cls_copy_func_t    copy_func;   /* Function to call when a property list
                                          is copied */
   void *                 copy_data;   /* Pointer to user data to pass along to
                                          copy callback */
   H5P_cls_close_func_t   close_func;  /* Function to call when a property list
                                          is closed */
   void *                 close_data;  /* Pointer to user data to pass along to
                                          close callback */
};
```

Conceptually, a property list class is a named list of properties with default values, decorated by functions to be called when a derived property list is created, copied, or closed, and finally, a pointer to its parent property list class.

The pointer to the parent class is central here, as the set of properties contained in the property list class is defined to be set of all properties that appear in a given instance of H5P_genclass_t A, **plus all properties that appear in the singly linked list of instances of H5P_genclass_t for which A is the first entry** (see below for management of duplicate properties). This linked list must always terminate in the ROOT property list class, which is created by the HDF5 library at initialization, and contains no properties.

While no single instance of H5P_genclass_t can contain two or more properties of the same name, different instances of H5P_genclass_t may contain properties of the same name. When

[2] But note that properties may be either added to or removed from individual property lists.

searching for a property in a linked list of instances of H5P_genclass_t, the first such property encountered shadows any later properties of the same name.

The properties (really instances of H5P_genprop_t) that reside in a property list class are stored in a skip list – which is implemented in the H5SL package. Note that this package is not thread safe, and – to a first order approximation – has been the primary cause of HDF5 performance degradation seen since HDF5 version 1.6[3].

nprops is used to track the number of unique properties in the target instance of H5P_genclass_t.

The classes field is used to maintain a count of the number of property list classes whose parent pointers point to this instance of H5P_genclass_t. Similarly, the plists field is used to maintain a count of the number of property lists that point to this instance of H5P_genclass_t with their pclass fields (see below).

The type field indicates which of the HDF5 defined property list classes this is, with an additional option for user defined property list classes. The definition of H5P_plist_type_t is reproduced below.

```
typedef enum H5P_plist_type_t {
   H5P_TYPE_USER             = 0,
   H5P_TYPE_ROOT             = 1,
   H5P_TYPE_OBJECT_CREATE    = 2,
   H5P_TYPE_FILE_CREATE      = 3,
   H5P_TYPE_FILE_ACCESS      = 4,
   H5P_TYPE_DATASET_CREATE   = 5,
   H5P_TYPE_DATASET_ACCESS   = 6,
   H5P_TYPE_DATASET_XFER     = 7,
   H5P_TYPE_FILE_MOUNT       = 8,
   H5P_TYPE_GROUP_CREATE     = 9,
   H5P_TYPE_GROUP_ACCESS     = 10,
   H5P_TYPE_DATATYPE_CREATE  = 11,
   H5P_TYPE_DATATYPE_ACCESS  = 12,
   H5P_TYPE_STRING_CREATE    = 13,
   H5P_TYPE_ATTRIBUTE_CREATE = 14,
   H5P_TYPE_OBJECT_COPY      = 15,
   H5P_TYPE_LINK_CREATE      = 16,
   H5P_TYPE_LINK_ACCESS      = 17,
   H5P_TYPE_ATTRIBUTE_ACCESS = 18,
   H5P_TYPE_VOL_INITIALIZE   = 19,
   H5P_TYPE_MAP_CREATE       = 20,
   H5P_TYPE_MAP_ACCESS       = 21,
   H5P_TYPE_REFERENCE_ACCESS = 22,
   H5P_TYPE_MAX_TYPE
} H5P_plist_type_t;
```

The remaining fields – ref_count, deleted, and revision – appear to exist to support modifications to property list classes at run time.

[3] I will argue later that skip lists should be removed from the property list implementation.

The revision field is set to a new, unique integer each time a property list class is created or modified. However, since multiple instances of H5P_genclass_t containing identical values are possible, its utility is questionable.

Once created, property list classes are normally stored in the index and (at least from a user perspective) are accessed via IDs. However, it is possible for a property list class (specifically an instance of H5P_genclass_t) to have multiple IDs, or no ID at all. The ref_count field is used to track the number of IDs in the index. The deleted field is set to TRUE when ref_count drops to zero. However, an instance of H5P_genclass_t is not deleted until the ref_count, classes, and plists fields all drop to zero. Further, it is possible for an instance of H5P_genclass_t which has lost all its references in the index to be re-inserted in the index – at which point ref_count is incremented and the deleted field is set back to FALSE. The mechanics of this will be discussed in greater detail when we discuss operations on property list classes and property lists below.

Property Lists

Conceptually, a property list is simply a list of (name, value) pairs. In H5P, property lists are stored in instances of H5P_genlist_t – whose definition is reproduced below.

```
/* Define structure to hold property list information */
struct H5P_genplist_t {
   H5P_genclass_t *pclass;     /* Pointer to class info */
   hid_t           plist_id;   /* Copy of the property list ID (for use in
                                  close callback) */
   size_t          nprops;     /* Number of properties in class */
                               // This comment appears to be in error.  In the
                               // context of a property list, nprops seems to be
                               // the number of properties in the property list.
                               // While the number of properties in a property list
                               // and in its immediate parent property list class will
                               // usually match, this need not be the case.
   hbool_t         class_init; /* Whether the class initialization callback
                                  finished successfully */
   H5SL_t *        del;        /* Skip list containing names of deleted properties */
   H5SL_t *        props;      /* Skip list containing properties */
};
```

Here, the props skip list contains all the properties that may have a value that is different from the default values listed in the properties of the same name in the linked list of property list classes (strictly speaking of instances of H5P_genclass_t) whose head is pointed to by the pclass field[4]. I say “may” here, as properties are copied from the base property list class(es) to the newly created property list if they have “copy” or “create” callbacks associated with them. While these callbacks are run on the default value to generate the value for the property in the new property list, this need not modify the property value.

[4] Or the parent property list class(es) for short.

As shall be seen, it is also possible for a property list to contain properties that do not appear in the parent property list class(es), or for a property to be reset to its default value without being removed from the props skip list.

The del skip list contains the names of properties that have been deleted from the property list. This is necessary, as in the absence of its name appearing in the del skip list, any property that doesn't appear in the props skip list is assumed to retain its default value – which is obtained by searching the parent property list class(es), starting with pclass→props.

The class_init boolean is used to record whether the create_func's (if any) of the parent property list class(es) completed without error when the property list was created. If they didn't, the corresponding close_func's are not run when the property list is closed.

Like property list classes, property lists are entered into the index. Unlike property list classes, the resulting IDs appear to be unique, and are stored in the plist_id field of the associated instance of H5P_genlist_t.

With this discussion of the underlying data structures in hand, we now proceed to discussions of selected operations on property list classes and property lists. For details on omitted operations, please see the appendices, or the source code.

Operations on properties are always incidental to these operations, and thus are covered in passing.

Selected Operations on Property List Classes

The details of the various public and private API calls that operate on property list classes are discussed in detail in the appendices. Some of these operations are what one would expect, and thus are not covered here.

This section boils these discussions down as much as possible, and focuses on operations that are either central, that highlight issues with the existing implementation, or that are likely to present challenges to a multi-thread safe implementation.

Property List Class Creation

New property lists are created via calls to

```
hid_t H5Pcreate_class(hid_t parent, const char *name,
                      H5P_cls_create_func_t create,
                      void *create_data, H5P_cls_copy_func_t copy,
                      void *copy_data, H5P_cls_close_func_t close,
                      void *close_data);
```

All HDF5 library property list classes are created by the H5P package during initialization, so there is no library private API call for this purpose. The package function call for creating a new property list class (but not installing it in the index) is

```
H5P_genclass_t *H5P__create_class(H5P_genclass_t *par_class,
                                  const char *name, H5P_plist_type_t type,
                                  H5P_cls_create_func_t cls_create,
                                  void *create_data,
                                  H5P_cls_copy_func_t cls_copy,
                                  void *copy_data,
                                  H5P_cls_close_func_t cls_close,
                                  void *close_data);
```

which is called by H5Pcreate_class() and other public API calls.

H5Pcreate_class does pretty much what one would expect – it calls H5P__create_class() to

- Allocate a new instance of H5P_genclass_t,
- Copy the supplied data into it,
- Create the props skip list,
- Set the nprops, plists, and classes fields to zero,
- Set deleted to FALSE,
- Set revision to a unique number, and
- Set ref_count to zero.

Assuming that the par_class parameter is not NULL, it sets the parent field to point to the parent property list class, and increments parent→classes.

After H5P__create_class() returns, H5Pcreate_class inserts the new property list class in the index, which increments its ref_count field in passing.

Property List Class Copy

While one wouldn't expect much use for a property list copy facility internally, the current property list implementation uses it heavily when modifying existing property list classes.

Interestingly, while there is a public API call

```
hid_t H5Pcopy(hid_t plist_id)
```

that can create a copy of a property list class, insert it in the index, and return the id of the copy, the user level documentation on this API call doesn't mention this capability.

This is may be a documentation bug, as there is the obvious use case of creating a customized version of a HDF5 defined property list class[5]. On the other hand, it may be an attempt to steer the user towards H5Pcreate_class() for this purpose, as that would at least allow the possibility of different property list class names. But if this is the case, why leave in the capability?

In contrast, the H5P package internal function

```
H5P_genclass_t * H5P__copy_pclass(H5P_genclass_t *pclass)
```

is used heavily in operations that modify existing property list classes, and is outlined here in preparation for these discussions.

H5P__copy_pclass() first calls H5P__create_class() to construct a duplicate of the instance of H5P_genclass_t that forms the source property list class. It then walks the props skip list of the source, duplicates each property, and inserts it into the props skip list in the duplicate. Note that the function does not insert the new instance of H5P_genclass_t in the index, or increment the classes field in the parent property list class.

Adding and Deleting Properties from an Existing Property List Class

As suggested earlier, the decision to employ a design that echoes ideas from object-oriented programming languages presents a number of issues that are not handled gracefully. Here is where the cracks begin to show.

An obvious question when modifying existing property list classes is what happens to existing property lists and property list classes that are derived from the property list class that is being modified? Several possible solutions present themselves:

1. Avoid the problem by making property list classes with derived property lists or property list classes read only.

2. Propagate changes through all derived property lists and property list classes.

3. Version the property list class so that the previously derived property lists and property list classes can still reference the version of the property list class from which they are derived.

[5] For example, a version of the FAPL property list class that includes a property needed to configure a user supplied VFD.

I don't have design documentation for the implementation of the property list facility in HDF5 – indeed, it is quite possible that such documentation never existed[6]. Thus, I don't know what considerations went into the choice between these options and any others I have missed.[7] However, from examination of the code, option 3 appears to have been selected. I say appears, as this apparent decision is not implemented uniformly.

In the public API, new properties can be inserted into an existing property list class[8] via two different calls:

```
herr_t H5Pcopy_prop(hid_t dst_id, hid_t src_id, const char *name)
```

and

```
herr_t H5Pregister2(hid_t cls_id, const char *name, size_t size,
                    void *def_value, H5P_prp_create_func_t create,
                    H5P_prp_set_func_t set, H5P_prp_get_func_t get,
                    H5P_prp_delete_func_t prp_del,
                    H5P_prp_copy_func_t copy,
                    H5P_prp_compare_func_t compare,
                    H5P_prp_close_func_t close);
```

Internally, all property list classes are created and populated during H5P package initialization, and hence there is not a library private function for this purpose. The call used here is

```
herr_t H5P__register_real(H5P_genclass_t *pclass,
                          const char *name,
                          size_t size,
                          const void *def_value,
                          H5P_prp_create_func_t prp_create,
                          H5P_prp_set_func_t prp_set,
                          H5P_prp_get_func_t prp_get,
                          H5P_prp_encode_func_t prp_encode,
                          H5P_prp_decode_func_t prp_decode,
                          H5P_prp_delete_func_t prp_delete,
                          H5P_prp_copy_func_t prp_copy,
                          H5P_prp_compare_func_t prp_cmp,
                          H5P_prp_close_func_t prp_close)
```

which simply inserts the new properties into the property list without versioning. This works, because at this point the target property list can't have any derived property lists or property list classes.

H5Pregister2() first looks up a pointer to the target property list class (call it A) and then calls H5P__register().

6 Until relatively recently, RFCs proposing modifications or extensions to the HDF5 library usually discussed only public API changes and motivation – with little or no attention to implementation details.

7 But note Appendix 3, in which I have reproduced what appears to be an early RFC for the current implementation of H5P.

8 Including HDF5 library defined property list classes.

H5P__register() starts by checking to see if A has any immediately derived property list classes or property lists.[9] If it doesn't, H5P__register() simply calls H5P__register_real() to:

- Create the specified property and insert it into A (specifically to insert it into the skip list pointed to by the props field),

- Increment the nprops field, and

- Set A's revision field to a new unique number.

In contrast, if A does have immediately derived property list classes or property lists, H5P__register():

- Calls H5P__copy_pclass() (see above) to duplicate A – call this copy A-Prime.

- Calls H5P__register_real() (see above) to create the new property and insert it into A-Prime.

After H5P__register() returns, H5Pregister2() checks to see if A-prime has been created. If it has, it calls H5I_subst() to replace A in the index with A-Prime and H5P_close_class() to decrement the ref_count field of A – which will be deleted when its ref_count, classes, and plists fields all drop to zero.

Observe that as a result of this process, we now have (at least) two versions of the property list class A – of which only the latest one is accessible via the index, with the previous version(s) being accessible only via the property list classes and property lists derived from that version. Note that this matches the user documentation, which states that modifications to property list classes only affect property lists created after the modification.

As shall be seen, this design decision has further implications when the user attempts to look up the parent class of a property list – see the section **Getting a Property Lists Parent Property List Class** below.

The public call for deleting a property from a property list class is

```
herr_t H5Punregister(hid_t pclass_id, const char *name)
```

[9] A property list class A has immediately derived property list class(es) or property list(s) if there exists one or more property list classes (i.e., instances of H5P_genclass_t) whose parent fields point to A and/or one or more property lists (i.e. ,instances of H5P_genlist_t) whose pclass fields point to A. This is determined via inspection of A's classes and plists fields. If either of these fields contain a positive value, the property list class A has immediately derived property list class(es) and or property list(s).

All property deletions in the HDF5 library are done within the H5P package, and thus there is no internal API call for this purpose. The H5P package function that performs property deletions from property list classes is

```
herr_t H5P__unregister(H5P_genclass_t *pclass, const char *name)
```

In addition to being called by H5Punregister() it is also used in H5Pcopy_prop().

H5Punregister() simply looks up a pointer to the instance of H5P_genclass_t (call it A) that forms the root of the data structure that represents the target property list class, passes this pointer along with the name of the target property to H5P__unregister(), and returns.

H5P__unregister() makes no test for immediately derived property lists or property list classes. It unconditionally:

- Deletes the named property from A's list of properties (an error is flagged if the named property doesn't exist),

- Decrements the nprops field, and

- Sets A's revision field to a new unique number.

This has the effect of deleting the named property from all derived property lists that have not yet changed the value of the target property from its default – which will cause the H5Pget() and H5Pset() calls to fail when invoked on the deleted property.

As this directly contradicts the user documentation, and since the behavior of H5P__unregister() is appropriate in its other use in H5Pcopy_prop(), this is probably a bug – which should be fixed if the current property list implementation is retained.

Deleting Property List Classes

The public API for deleting a property list class is

```
herr_t H5Pclose_class(hid_t plist_id);
```

There is no corresponding internal API or H5P package function – which should be no surprise given that a property list class can't be deleted until its ref_count, classes, and plists fields all drop to zero.

H5Pclose_class() calls H5I_dec_app_ref() with the id associated with the target property list class – call it A. This has the effect of decrementing the reference count on the index entry (not to be confused with the reference counts on A). If this index reference count drops to zero, H5I deletes the entry from the index, and calls H5P__close_class_cb() to allow H5P to do the appropriate cleanup.

H5P__close_class_cb() calls H5P__close_class() which calls H5P__access_class() which does the real work.

H5P__access_class() is a bit of a Swiss army function. Depending on its parameters, it will increment or decrement the ref_count, classes, or plists fields of the target property list class. When all of these fields drop to zero, it decrements the classes field of the parent property list class via a recursive call to itself, and then discards the target property list class.

Iteration over Properties in a Property List Class

The main point to be made here is that the H5Piterate() public API exists, and presents a variety of multi-thread safety issues. While we are far from making such decisions, it should probably be dealt with like the other iteration functions encountered in this effort.

Selected Operations on Property Lists

As with property list classes, the details of the various public and private API calls that operate on property lists are discussed in detail in the appendices.

As before, the objective is to cover central operations, to highlight issues with the existing implementation, and to point out challenges for a multi-thread safe implementation.

Note that some public API calls operate on both property list classes and property lists – and thus appear both in this section, and the previous discussion of selected operations on property list classes.

Property List Creation, Deletion, and Copy

The public API call to create a new property list is

```
hid_t H5Pcreate(hid_t cls_id)
```

Its internal API cognate is

```
hid_t H5P_create_id(H5P_genclass_t *pclass, hbool_t app_ref);
```

which is called almost immediately by H5Pcreate(), which looks up the pointer to the parent property list class (call it P), and then passes it to H5P_create_id().

H5P_create_id() first calls H5P__create() which does most of the work. In particular it:

- Allocates and initializes a new instance of H5P_genlist_t, along with its associated skip lists. Call this new property list A

- Scans the properties in the parent property list class(es) starting with P, being careful to skip any properties with duplicate names after the first instance has been encountered. For each such property, it tests to see if it has a create callback, and if so, it duplicates the property (calling its create callback in passing) and inserts it into the skip list pointed to by A's props field.

- Calls H5P__access_class() to increment P's plists field.

- Returns a pointer to A

After H5P__create() returns, H5P_create_id() inserts A into the index, and then scans the list of parent property list class(es), starting with P and executes their create callbacks (the create_func field in H5P_genclass_t). If all these functions complete successfully, its sets A's class_init field to TRUE. Regardless of the success of the create callbacks, H5P__create returns the new ID associated with A, which is returned to the user by H5Pcreate().

The public API call for discarding a property list is

```
herr_t H5Pclose(hid_t plist_id);
```

Its private API cognate is

```
herr_t H5P_close(H5P_genplist_t *plist)
```

which is used extensively in error cleanup and also by H5I to discard property lists whose reference count has dropped to zero.

H5Pclose() calls H5I_dec_app_ref() with the id associated with the target property list – call it A. This has the effect of decrementing the reference count on the index entry. If this index reference count drops to zero, H5I deletes the entry from the index, and calls H5P__close_list_cb() to discard A.

H5P_close_list_cb() is just a pass through function that calls H5P_close()

H5P_close() does the actual work. It is a long and involved function – simplifying as much as possible, it

- Tests to see if A's class_init field is TRUE. If it is, it scans the list of parent property list class(es), starting with *pclass and executes each property list class's close callback (the close_func field in H5P_genclass_t) in order.

- Scans the entries in A's props skip list, and calls the close callback for each property that has one.

- Scans the entries in the parent property list class(es) props skip list, starting with the parent property list class pointed to by A's parent field. For each such entry, it tests to see if a property of that name appears in A's props or del skip lists, or earlier in the current scan. If not, it calls that property's close callback if it exists.

- Decrement A's immediate parent property list class's plists field via a call to H5P__access_class(). Recall that may result in the discard of one or more of the property lists class(es) of A.

- Discard the props and del skip lists maintained by A, along with their contents.

- Discard A's base instance of H5P_genlist_t.

The public API call to copy property lists is

```
hid_t H5Pcopy(hid_t plist_id)
```

Recall that this API call also works on property list classes, although this fact is not mentioned in the user documentation. The private API cognate is

```
hid_t  H5P_copy_plist(const H5P_genplist_t *old_plist, hbool_t app_ref)
```

The process for copying a property list is similar to that for creating a property list. The major differences are the use of property copy callbacks, and the need to copy properties from the base property list. See the appendices for details.

Property Insertion and Deletion

The whole idea of either inserting or deleting properties from an existing property list is a departure from the object-oriented notions that appear to have been the basis of the property list facility. However, it does allow user processes to insert new properties in a property list without modifying HDF5 library defined property list classes, or creating modified copies of the same. As such, it has obvious uses for properties that are only needed once.

The public API for inserting properties into an existing property list is

```
herr_t H5Pinsert2(hid_t plist_id, const char *name, size_t size,
                  void *value, H5P_prp_set_func_t prp_set,
                  H5P_prp_get_func_t prp_get,
                  H5P_prp_delete_func_t prp_del,
                  H5P_prp_copy_func_t prp_copy,
                  H5P_prp_compare_func_t prp_cmp,
                  H5P_prp_close_func_t prp_close)
```

The private API cognate is

```
herr_t H5P_insert(H5P_genplist_t *plist, const char *name, size_t size,
                  void *value, H5P_prp_set_func_t prp_set,
                  H5P_prp_get_func_t prp_get,
                  H5P_prp_encode_func_t prp_encode,
                  H5P_prp_decode_func_t prp_decode,
                  H5P_prp_delete_func_t prp_delete,
                  H5P_prp_copy_func_t prp_copy,
                  H5P_prp_compare_func_t prp_cmp,
                  H5P_prp_close_func_t prp_close)
```

H5Pinsert2() looks up a pointer to the target property list (call it A) and then passes this pointer to H5P_insert() along with the necessary parameters.

H5P_insert() verifies that the skip list referenced by A's props field doesn't contain a property of the supplied name, and returns an error if it does.

It then checks A's deleted list (referenced by the del field) to see if it contains the supplied name, and removes it from the deleted list it is found.

If the supplied name doesn't appear in the deleted list, H5P_insert() next searches for a property of the supplied name in A's parent property list class(es), and returns an error if this search is successful.

With this sanity checking complete, H5P_insert() creates the new property as specified by its parameters, inserts it in the skip list referenced by A's props field, increments A's nprops field, and returns

The public API for deleting properties from property lists is

```
herr_t H5Premove(hid_t plist_id, const char *name)
```

Its private API cognate is

```
herr_t H5P_remove(H5P_genplist_t *plist, const char *name)
```

H5Premove() looks up a pointer to the target property list (call it A) and then passes this pointer to H5P_remove() along with the name of the target property. H5P_remove() in turn calls H5P__do_prop() to do the actual work.

H5P__do_prop() is another swiss army function. It scans the target property list for the target name, and then applies one of the supplied functions depending on where the target property is found. In this case, it performs as follows:

Test to see if the target name already appears in A's deleted list, and fail if it does.

Search for the target property in A's props skip list. If it is found, run the properties deletion callback, insert the properties name in A's deleted list, remove the property from the skip list, free it, decrement A's nprops field, and return.

If the search of A's props skip list fails, search the properties of A's parent property list class(es) for the target property. If it is found, run the properties deletion callback **on a copy of the properties value[10]**, insert the properties name in A's deleted list, decrement A's nprops field and return. Note that the target property is not removed from its host property list class.

If both searches fail, H5P__do_prop() fails.

Property Value Get and Set

The public API call to get the value of a property in a property list is

```
herr_t H5Pget(hid_t plist_id, const char *name, void *value)
```

Its internal API cognate is

```
herr_t H5P_get(H5P_genplist_t *plist, const char *name, void *value)
```

Conceptually, these routines look up the target property in the target property list, and copies its value into the supplied buffer. The actual implementation is a bit more complex.

H5Pget() looks up a pointer to the target property list (call it A) and passes this pointer into H5P_get() with its other parameters.

H5P_get() calls H5P__do_prop() with the appropriate parameters to perform the lookup and then returns.

In this configuration, H5P_do_prop() first scans A's deleted list for the target name, and returns failure if this search succeeds.

If the search of A's deleted list fails, H5P__do_prop() searches first A's props skip list, and then the props skip lists of the parent property list class(es) until a property of the specified name is found.

If the search fails, H5P__do_prop() returns failure.

If the search succeeds, H5P__do_prop() tests to see if the target property has a get callback.

[10] At present, I don't know why this is done. I expect that investigation of the call back functions will reveal the answer.

If it doesn't, H5P__do_prop() just memcpy's the target property's value buffer into the supplied buffer and returns.

If it does, H5P__do_prop() allocates a temporary buffer of size equal to the target property's value buffer, copies the value buffer into the temporary buffer, runs the properties get function on the temporary buffer, copies the temporary buffer into the supplied buffer[11], discards the temporary buffer and returns.

The public API to set the value of a property in a property list is

```
herr_t H5Pset(hid_t plist_id, const char *name, const void *value)
```

Its private API cognate is

```
herr_t H5P_set(H5P_genplist_t *plist, const char *name, const void *value)
```

Conceptually, these routines look up the property of the supplied name in the target property list, and set its value to the supplied value. In practice, this is complicated by property specific set callbacks, and the possibility that the target property may still have its default value, and thus not reside in the target property list.

H5Pset() looks up a pointer to the target property list (call it A) and passes this pointer into H5P_set() with its other parameters.

H5P_set() calls H5P__do_prop() with the appropriate parameters to perform the lookup and then returns. Note that in this case, the supplied function pointers are different – one for the case in which the target property is in A's props skip list, and another for the case in which the target property is found in one of A's property list class(es).

In this configuration, H5P__do_prop() first scans A's deleted list for the target name, and returns failure if this search succeeds.

It then searches A's props skip list for the target property.

If the target property is found, H5P__do_prop()

- Tests to see if the property's set call back is defined. If it is defined, H5P__do_prop() allocates a buffer of size equal to the property's value buffer, memcpy's the supplied buffer into the temporary buffer, and runs the property's set callback on the temporary buffer[12].

---

[11] Again, the reason for this is an unknown to me. A review the property callbacks is needed to determine if this is really necessary, and if so why.

[12] Yet another epicycle whose motivation must be investigated.

- Tests to see if the property's del callback is defined, and if so calls it on the property's value.

- Memcpy's the new value into the property's value field, either from the supplied buffer (if the set callback is undefined) or from the above temporary buffer if it is.

- Frees the temporary buffer if it exists, and

- Returns

If the target property is not found in A's props skip list, H5P__do_prop() searches for a property of the specified name in the parent property list class(es). If such a property is found, H5P__do_prop()

- Checks to see if the target property's set callback is defined, and if so, set up the temporary buffer as above.

- Duplicates the property

- Inserts the duplicate property into A

- Memcpy's the new value into the duplicate property's value field, either from the supplied buffer (if the set callback is undefined) or from the above temporary buffer if it is.

- Frees the temporary buffer if it exists, and

- Returns

If both searches fail, H5P__do_prop() fails.

Property List Encode and Decode

Perhaps the only points to be made here is that the facility for encoding and decoding property lists exists, that it is restricted to property lists derived from one of the HDF5 library defined types, and that only properties with encode / decode callbacks are encoded / decoded.

To my knowledge, there is no documentation on the file format of encoded property lists. The format is very simple, so this isn't a major issue.

Getting a Property Lists Parent Property List Class

The public API

```
hid_t H5Pget_class(hid_t plist_id)
```

exists to return the id of the immediate parent property list class of the supplied property list. It has an internal API cognate

```
H5P_genclass_t *H5P_get_class(const H5P_genplist_t *plist)
```

which fortunately doesn't have the problems that H5Pget_class() has.

In principle, this should be simple. Property list classes are stored in the index, and are referenced by their ID when the user creates a property list based on one of the extant property list classes. Thus, it should be simple for a property list to report the ID of the property list class it is immediately derived from.

However, the policy decision to version property list classes that have immediately derived property list classes or property lists when the target property list class is modified breaks this[13], as the relevant version of the property list class may no longer exist in the index.

H5Pget_class() solves this problem by unconditionally inserting the instance of H5P_genclass_t pointed to by its parent field into the index, and returning the newly allocated ID.

This has several knock-on effects:

- A given property list class can have multiple IDs – and therefore multiple entries in the index.
- The index can contain multiple property list classes with the same name but different contingents of properties, or perhaps the same properties but with different default values.
- Since it is also possible to have multiple property list classes with identical definitions but different IDs, the process of comparing property list classes is greatly complicated. Instead of a simple comparison of IDs or revision numbers, a deep, field by field comparison is required. See H5Pequal() in the first appendix.

This introduces a level of potential confusion into the public API that seems unacceptable. If at all possible, this should be resolved.

[13] See the section **Adding and Deleting Properties from an Existing Property List Class** above for further discussion.

Comments on the Current Implementation of Property Lists

As should be obvious from the above, the data structures used to implement property lists are complex. While I expect it is possible, implementing the mutual exclusion required to avoid data structure corruption in a multi-thread environment would be challenging, and likely slow due to the multitude of locks. More importantly, its complexity would make it hard to avoid accidentally inserting either deadlocks or holes in mutual exclusion during routine maintenance. Finally, a complex MT solution will exacerbate the lock ordering problem between packages.

The use of skip lists (implemented in H5SL) is also worthy of comment. Skip lists were once used extensively throughout the HDF5 library. However, profiling exercises indicated that the shift to skip lists was the primary cause of the slowdown in HDF5 performance since HDF5 1.6.

Much of this slowdown was located in H5P as skip lists are fairly heavy weight data structures, property lists seldom if ever exceed 25 entries, and (at the time) properties were looked up repeatedly. The introduction of H5CX largely mitigated this issue as it (among other things) caches the values of commonly accessed property list entries. However, property list operations remain expensive, and other optimization efforts have pushed H5P back up near the top of the optimization to-do list.

Given this move away from skip lists for performance reasons, it is hard to argue for developing a MT safe implementation for use in property lists.

If the above points are not sufficient to rule out any attempt to retrofit multi-thread safety on the current property list implementation, there are also the various interesting features of the current implementation as outlined above. In particular, while one can argue the merits of the versioning approach to modifications to property list classes, the current implementation introduces too much complexity, and too much potential confusion into the public API. Further, I am not aware of any use case for it.

Given all the above, the remainder of this RFC operates on the assumption that a re-design and re-implementation of the core property list code is the only viable approach. Needless to say, this re-implementation should change the public and private API’s as little as possible, and then only with prior consultation with the developer and user communities.

Multi-Thread Issues in H5P

Before outlining the design of a proposed redesign and re-implementation of property lists, it will be useful to list the issues that should be addressed.

Use of other HDF5 packages in H5P

Any implementation of property lists must use H5E (error reporting) and H5I (indexing).

At least for internal calls, H5E was largely multi-thread safe for internal calls – modulo its dependency on H5I. Public API calls raise a number of issues, but for the time being, these can be addressed by keeping them under the global mutex.

Since the last revision of this document, H5I has been modified to use a lock free hash table, making it fundamentally lock free. Unfortunately, there is no guarantee that its callbacks are lock free, and thus they are executed under the global lock. Further, since these callbacks can't be rolled back, it is necessary to lock the relevant index entries while these callbacks are in progress[14].

The net effect of this is that at least in the context of HDF5 library proper, there shouldn't be much in the way of lock ordering concerns from H5E and H5I in H5P – as long as H5P can handle non multi-thread safe callbacks the same way as H5I.

Whether this is practical comes down to the particulars of the interactions that H5P has with the various packages that use property lists derived from HDF5 library defined property list classes both for configuration and (in a few cases) to return data to the user remains.

These packages implement property specific callbacks that must be executed when various operations are performed on the associated properties. In my experience, few if any of these callbacks do anything significant from a multi-thread safety perspective. However, all these callbacks must be reviewed, documented, and tamed where necessary. Construction of a census of these callbacks and the required analysis is in progress as of this writing.

A more subtle issue is the possibility of modifications to property lists while calls referring to them are in progress. While this is a user error, we still have to keep it from corrupting internal data structures.

After some thought, I have come to the conclusion that the problem of packages that use property lists to return data to the user is best solved by requiring the user to create thread specific copies of the property list in question. Absent this, there is a fundamental race condition that will be difficult if not impossible to address satisfactorily. Note, however, that we can't force users to do this, so we must ensure that this doesn't corrupt our data structures, even if it does return garbage to the user.

---

[14] As we have not yet settled on a thread package for this work, these locks are currently implemented via atomics. Once we do choose a threading package for this work, these home brew locks will likely be replaced with recursive mutexes.

Finally, there is the matter of user defined properties and property list classes. Both of these have user defined callbacks – other than asking nicely and running these callbacks under the global mutex unless assured that they are thread safe, we don't have any way of filtering out code that could cause deadlocks and such. Unfortunately, I don't see any practical solutions other than careful user documentation or disallowing these callbacks in the multi-thread case.

Multi-thread thread issues in H5P proper

There is the usual problem of potential public API race conditions. This is an unsolvable problem from the perspective of H5P – all H5P can do is to execute operations in some order, and to keep its data structures in an internally consistent state. It will be the responsibility of the client to either avoid race conditions of the above type, or to handle them gracefully.

Proposed Solution

In the previous version of this document, the notion of simplifying H5P by making property list classes read only if they have any derived property lists or property list classes was floated. While this would work with HDF5 proper, there are use cases for adding properties to existing property list classes that make this suggestion unworkable.

This means that, at least from an external perspective, we must maintain the support for versioning property list classes, with the requirement that any such changes must not be reflected in any preexisting property list classes or property lists.

While the following design could be modified to permit it, as presented, it does not expose old versions of property list classes to the user. As discussed above, this feature invites confusion, and should, in my opinion, be dropped if at all possible.

As per H5I, the objective of the H5P redesign is to make the property list code fundamentally lock free.

The following design replaces skip lists with lock free singly linked lists (LFSLLs). This design choice is predicated on the proposition that property lists and property list classes will be relatively short – making the cost of linear searches on sorted lists acceptable. Needless to say, it will be necessary to maintain statistics to see if this proposition is correct. If it isn't, replacing the LFSLLs with lock free hash tables will be a simple fix.

Secondly, the design replaces the current practice of creating and maintaining copies of property list classes with versioning properties within property list classes and property lists. While the primary impetus behind this decision was to allow modifications to property list classes and property lists to be effectively atomic, it should have the side benefit of reducing memory footprint.

Data Structures

Properties, or more correctly versions of properties, are stored in instances of H5P_mt_prop_t. See below for definitions of this structure, and its supporting structures H5P_mt_prop_aptr_t and H5P_mt_prop_value_t.

H5P_mt_prop_t is a modified version of H5P_genprop_t, with added fields to support lock free operation and versioning. Close relatives to the lock free algorithms used in this design are discussed in "The Art of Multiprocessor Programming" by Maurice Herlihy, Victor Luchangco, Nir Shavit, and Michael Spear, and thus will be glossed over here.

The create_version field is used to record the version of the containing property list class or property list in which the represented property version was added. The delete_version field will normally be zero, but will be set to the version of the containing property list class or property list at which the property was deleted.

The chksum field contains a 32 bit checksum on the name of the property – this is done for the convenience of the LFSLL (and lock free hash table if it comes to it) and to speed up lookups in the table of inherited properties in property lists (of which more later).

Both property list classes and property lists maintain LFSLLs containing instances of H5P_mt_prop_t, one for each version of each property that has appeared in the containing property list class or property list[15]. This list is sorted first by hash code, then by name[16], and finally by creation_version in descending order. Since all searches for properties must have a target version of the containing property list or property list class, this allows easy determination of the property version (if any) present in the target version.

See the header comments on H5P_mt_prop_t, H5P_mt_prop_aptr_t and H5P_mt_prop_value_t for further details.

```
/****************************************************************************************
 *
 * struct H5P_mt_prop_aptr_t
 *
 * Struct H5P_mt_prop_aptr_t is a structure designed to contain a pointer to an instance
 * of H5P_mt_prop_t and a deleted flag in a single atomic structure.  This is necessary,
 * as instances of H5P_mt_prop_t will typically appear in lock free singly linked lists.
 *
 * For correct operation, these lists require the next pointer and the deleted flag to
 * be accessed and modified in a single atomic operation.
 *
 * With padding, this structure is 128 bits, which allows true atomic operation on
 * many (most?) modern CPUs.  However, it this becomes a problem, we can obtain the
 * same effect by stealing the low order bit of the pointer for a deleted bit -- which
 * works on all CPU / C compiler combinations I have tried.
 *
```

[15] Not quite – as shall be seen, property lists refer to their parent property list classes for default values of inherited properties.

[16] To allow for hash code collisions.

```
 * The individual fields of the structure are discussed below:
 *
 * ptr:        Pointer to an instance of H5P_mt_prop_t, or NULL.
 *
 * deleted:    Boolean flag.  If this instance of H5P_mt_prop_aptr_t appears as a
 *             field in an instance of H5P_mt_prop_t, the instance of H5P_mt_prop_t is
 *             logically deleted if this flag is TRUE, and not if the flag is FALSE.
 *
 *             If an instance of H5P_mt_prop_aptr_t appears elsewhere, the flag is
 *             meaningless.
 *
 * dummy_bool_1:
 * dummy_boot_2:
 * dummy_bool_3: The dummy_bool_? fields exist to pad H5P_mt_prop_aptr_t out to 128 bits,
 *             and allow prevention of insertion of garbage into an atomic instance of
 *             H5P_mt_prop_aptr_t, thus avoiding spurious failures of
 *             atomic_compare_exchange_strong().  They should always be set to false.
 *
 ****************************************************************************************/

typedef struct H5P_mt_prop_aptr_t {

   struct H5P_mt_prop_t * ptr;

   bool             deleted;

   bool             dummy_bool_1;
   bool             dummy_bool_2;
   bool             dummy_bool_3;

} H5P_mt_prop_aptr_t;


/****************************************************************************************
 * struct H5P_mt_prop_value_t
 *
 * Properties in a property list consist of a void pointer and a size.  To avoid race
 * conditions, the size and pointer must be set atomically.  This structure exists
 * to facilitate this.
 *
 * ptr:        Void pointer to the value, or NULL if the value is undefined.
 *
 * size:       size_t containing the size of the buffer pointed to by the ptr field,
 *             or zero if ptr is NULL.
 *
 * Note: The above fields will usually have a total size of 128 bits.  However, since
 * the size of size_t is not fixed across all 64 bit compilers, there is the potential
 * for occult failures in atomic_compare_exchange_strong() when garbage gets into
 * the un-used space in the structure. (recall that the sum of the sizes of the fields
 * in a structure need not equal the allocation size of the structure.)
 *
 * For now, it should be sufficient to assert that sizeof(size_t) = 8.
 * However, we will have to deal with the issue eventually.  For example, I have read
 * that size_t is a 32 bit value on at least some compilers targeting Windows.
 *
 ****************************************************************************************/

typedef struct H5P_mt_prop_value_t {

   void * ptr;
   size_t size;

} H5P_mt_prop_value_t;
```

```
/*******************************************************************************************
 * struct H5P_mt_prop_t
 *
 * Struct H5P_mt_prop_t is a revised version of H5_genprop_t designed for use in a
 * multi-thread safe version of H5P.  The data structures supporting property lists
 * are lock free to the extent practical, and thus instances of H5P_mt_prop_t will
 * typically appear in lock free singly linked lists.
 *
 * Further, to support versioning in property list classes,instances of H5P_mt_prop_t
 * in property list classes maintain reference counts of the number of property lists
 * that refer to them for default values, the revision number at which they inserted
 * into the containing property list class or property list, and (if deleted) the
 * revision number at which the deletion took place.
 *
 * The fields of H5P_mt_prop_t are discussed individually below.
 *
 * tag:        Integer value set to H5P_MT_PROP_TAG when an instance of H5P_mt_prop_t
 *           is allocated from the heap, H5P_MT_PROP_VALID_ONFL_TAG when an
 *             instance has been added to the property free list but could still possibly be
 *             accessed, and to H5P_MT_PROP_INVALID_TAG just before it is released back
 *             to the heap. The field is used to validate pointers to instances of H5P_mt_prop_t,
 *
 *
 * next:       Atomic instance of H5P_mt_prop_aptr_t, which combines a pointer to the
 *           next element of the lock free singly linked list with a deleted flag.
 *           If there is no next element, or if the instance of H5P_mt_prop_t is
 *           not in a LFSLL, this field should be set to {NULL, false}.
 *
 * sentinel:    Boolean flag.  When set, this instance of H5P_mt_prop_t is a sentinel
 *           node in the lock free singly linked list -- and therefore does not
 *           represent a property.
 *
 * in_prop_class: Boolean flag that is set to TRUE if this instance of H5P_mt_prop_t
 *           resides in a property list class, and false otherwise.  Note that
 *           the ref_count field is un-used if this field is false.
 *
 * ref_count:   Atomic integer used to track the number of property list properties
 *           that point to this instance of H5P_mt_prop_t.  This field must be
 *           zero if in_prop_class is false.
 *
 *           Note that this ref_count is only increased when a new property list
 *           is created, and is decremented when the property list is discarded.
 *
 *           Thus this instance of H5P_mt_prop_t can be safely deleted if:
 *
 *                1) the ref count drops to zero, and
 *
 *                2) this property has been either deleted or superseded
 *                  in the property list class.
 *
 *
 * Property Chksum, Name & Value:
 *
 * Instances of H5P_mt_prop_t will typically appear in lock free singly linked lists,
 * which must be sorted by property name, and then by decreasing create_version (i.e.
 * highest create_version first).
 *
 * The lock free singly linked list used to store most instances of H5P_mt_prop_t
 * requires sentinels at the beginning and end of the list with values (conceptually)
 * of negative and positive infinity respectively.  This is a bit awkward with strings,
 * so for this reason, the (name, creation_version) key is augmented with a 32 bit
 * checksum on the name, converting the key to a (chksum, name, creation_version)
 * triplet.  Note that the check sum is a 32 bit unsigned value, which is stored
 * in an int64_t.  Thus we can use LLONG_MIN and LLONG_MAX as our negative and
 * positive infinity respectively.
 *
```

```
* The addition of the chksum changes the sorting order to checksum, name, and
* then decreasing create_version.  This ordering, along with the delete_version
* field, allows us to operate on specific versions of a property list classes and
* property lists -- thus allowing concurrent operations without introducing
* corruption.
*
*
* chksum:     int64_t containing a 32 bit checksum computed on the name field
*           below, or LLONG_MIN or LLONG_MAX if either the head or tail
*           sentinel in the lock free SLL respectively.
*
*           Since this field is constant for the life of the instance of
*           H5P_mt_prop_t, and is set before the instance is visible to more
*           than one thread, it need not be atomic.
*
* name:       Pointer to a dynamically allocated string containing the name of the
*           property.  This field is not atomic, as the string should be allocated,
*           and initialized, and the name field set before the instance of
*           H5P_mt_prop_t is visible to more than one thread.  Since the name
*           is constant for the life of the instance of H5P_mt_prop_t, this should
*           be sufficient for thread safety.
*
* value:      Atomic structure containing the pointer to the buffer containing the
*           value of the property, and its size.
*
* create_version: Atomic integer which is set to the version of the containing
*           property list class or property list in which this property was
*           inserted.
*
* delete_version: Atomic integer which is set to the version of the containing
*           property list class or property list in which the property was
*           deleted.  If the property has not been deleted, this field contains
*           zero.
*
*
*
*
* Property Callback Functions:
*
* The following fields are pointers to the callback functions associated with the
* property, combined with a Boolean indicating whether all callbacks are thread safe.
* If this flag is not set, all callbacks are protected by the global mutex.
*
* The descriptions of the callbacks are all taken from Matt Larson's "Census of H5P Callbacks",
*           and are a major improvement on the existing documentation.  These descriptions
*           may have to be modified to reflect the re-implementation of H5P.
*
* callbacks_mt_safe: Boolean flag used to indicate whether all callbacks are
*           multi-thread safe.  If this field is not set, the global mutex must
*           be held when the callback is called.
*
* create:     Function to call when a property is created.
*
*           Signature:
*
*              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:      Function to call when a property value is set.
*
*         Signature:
*
*             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:      Function to call when a property value is retrieved.
*
*         Signature:
*
*           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.
*
* encode:   Function to call when a property is encoded.
*
*         Signature:
*
*           herr_t
*           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:   Function to call when a property is decoded.
*
*         Signature:
*
*           herr_t
*           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.
*
* del:      Function to call when a property is deleted.
*
*           Signature:
*
*              herr_t
*              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.
*
* copy:     Function to call when a property is copied.
*
*           Signature:
*
*              herr_t
*              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.
 *
 * cmp:       Function to call when a property is compared.
 *
 *          Signature:
 *
 *             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.
 *
 * close:     Function to call when a property is closed.
 *
 *          Signature:
 *
 *             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.
 *
 ****************************************************************************************/

#define H5P_MT_PROP_TAG
#define H5P_MT_PROP_VALID_ONFL_TAG
#define H5P_MT_PROP_INVALID_TAG


typedef struct H5P_mt_prop_t {

   uint32_t                 tag;

   _Atomic H5P_mt_prop_aptr_t   next;

   bool                     sentinel;

   bool                     in_prop_class;
   _Atomic uint64_t          ref_count;

   int64_t                  chksum;
   char *                   name;
   _Atomic H5P_mt_prop_value_t  value;
```

```
    _Atomic uint64_t           create_version;
    _Atomic uint64_t           delete_version;

    bool                       callbacks_mt_safe;
    H5P_prp_create_func_t      create;
    H5P_prp_set_func_t         set;
    H5P_prp_get_func_t         get;
    H5P_prp_encode_func_t      encode;
    H5P_prp_decode_func_t      decode;
    H5P_prp_delete_func_t      del;
    H5P_prp_copy_func_t        copy;
    H5P_prp_compare_func_t     cmp;
    H5P_prp_close_func_t       close;

} H5P_mt_prop_t;
```

Property list classes are stored in instances of H5P_mt_class_t. See below for the definition of this structure.

As with H5P_genprop_t, H5P_mt_class_t is a version of H5P_genclass_t modified to support multi-thread. The major changes from H5P_genclass_t are as follows:

Properties (or more correctly, versions of properties) are stored in a LFSLL as described above. Properties inherited from ancestor property list classes are copied into the LFSLL associated with the new property list class at creation time. Only the current versions of properties are copied. This is a matter of code simplification. The current search through the inheritance tree could be used, it would just be more complex. Note, however, that we must still execute the inherited callbacks.

The current version of a property list class is stored in the curr_version field, and all read operations on the property list class must be directed at a version no greater than this value at the point at which the operation starts. This has the effect of making the read effectively atomic as any subsequent modifications to the property list will not be visible to the read.

When an operation that modifies the property list class[17] starts, it must first obtain a version number for the change. This is done via an atomic fetch and increment on the next_version field. The returned new version number is used to tag the change.

In the event of a property insertion or modification, a new instance of H5P_mt_prop_t is allocated and initialized as appropriate, tagged with the new version in its create_version field, and then inserted into the LFSLL of properties in sorted order. Recall that this list is sorted by hash, name, and then by decreasing creation version.

For property deletions, the version of the target property that is valid at version one less than the target version is found, and its delete version is set to the changed version number.

---

[17] i.e. a property create, delete, or modify.

Observe that by requiring all read operations on the property list class to target a version no greater than curr_version, we force modifications to the property list class to remain invisible until the curr_version field is incremented – thus making modifications effectively atomic. While this clearly works if no more than one modification is in progress at one time, there are issues with concurrent modifications.

If multiple operations are in progress simultaneously, it is usually sufficient to ensure that operations complete in issue order (i.e. the order in which change version numbers are issued). If, however, a modify or insert and a delete are in progress on the same property simultaneously, they must be executed in issue order.[18]

See the header comment on H5P_mt_class_t for further details.

```
/****************************************************************************************
 *
 * struct H5P_mt_active_thread_count_t
 *
 * Struct H5P_mt_active_thread_count_t is structure designed to contain a counter of the
 * number of threads currently active in the host structure and opening and closing flags
 * in a single atomic structure.  The objectives are to prevent access to the containing
 * structure during setup, and to provide a mechanism for delaying the discard of the
 * containing structure until all threads currently active in the structure have exited.
 *
 * The possibility of access to a property list class or property list that is in the
 * process of being set stems from two points.
 *
 * First,some callbacks that must be called during setup require the ID of the host
 * property list.  This requires that the property list be inserted into the index,
 * which in turn makes the incomplete property list accessible to other threads.
 *
 * Even in the absence of these callbacks, both property list classes and property lists
 * must be inserted into the index before they are completely set up.  While it is
 * improbable, this makes them accessible to other threads via iterations on the host
 * indexes.
 *
 * To prevent this, the opening flag in the contained instance of
 * H5P_mt_active_thread_count_t is initialized to TRUE, and not set to FALSE until setup
 * completes.  Any thread that wants to access the host property list much check this flag
 * on entry, and fail if it is TRUE.
 *
 * In principle, it should be impossible for any thread to access a property list class
 * or property list that is in the process of being taken down.  However, it seems prudent
 * to have a mechanism to detect the case where it does, and to manage it gracefully. Note
 * that in debug builds we should throw an assertion failure whenever a circumstance that
 * is forbidden occurs.  One could argue that in production builds we should log the issue
 * and handle it gracefully – I am not sure I agree, but this is a discussion for another
 * time.
 *
 * In the typical case of a thread that reads or modifies the host data structure, it must
 * first do an atomic fetch on the associated instance of H5P_mt_active_thread_count_t and
 * fail if either the opening or closing flag is set.  If neither flag is set, it must
 * increment the thread counter in the local copy, and attempt to overwrite the shared copy
 * with the local copy using a call to atomic_compare_exchange_strong(). If this fails, it
 * must repeat the procedure until successful, or until the closing flag is set.  When the
```

[18] This added bit of complexity could be avoided by inserting a tombstone instance of H5P_mt_prop_t into the LFSLL, and deleting the deletion_version field from H5P_mt_prop_t. Given the expected infrequency of simultaneous modifications to property list classes, it is probably simplest to just serialize these operations.

```
 * thread is done with host data structure, it must again load the associated instance of
 * H5P_mt_active_thread_count_t, decrement the thread count in the local copy, and attempt
 * to overwrite the shared copy with the local copy with another call to
 * atomic_compare_exchange_strong() – repeating the procedure until successful.  Note that
 * the flags are ignored in this case.
 *
 * Similarly, a thread that is about to discard the host structure must first do an atomic
 * fetch on the associated instance of H5P_mt_active_thread_count_t and fail either the opening
 * or closing flag is set – preferably with an assertion failure.  If the closing flag is not
 * set, it must set it in the local copy, and attempt to overwrite the shared copy with the
 * local copy using a call to atomic_compare_exchange_strong(). If this fails, it must repeat
 * the procedure until successful.  Once the closing flag is set, it must verify that no threads
 * are active in the host structure – either throwing an error or waiting until the thread
 * count drops to zero as appropriate.
 *
 * The individual fields of the structure are discussed below:
 *
 * count:      Number of threads currently active in the host structure.
 *
 * opening:    Boolean flag that is set to TRUE during while the host property list class or
 *             property list is in the process of being set up.  It must be set to FALSE
 *             once setup is complete.
 *
 *             This is necessary since the host property list class or property list must
 *             be inserted into the appropriate index before setup is complete.
 *
 * closing:    Boolean flag that is set to TRUE iff the host structure is about to be
 *             discarded.
 *
 ******************************************************************************************/

typedef struct H5P_mt_active_thread_count_t {

   uint64_t        count;
   bool           opening
   bool           closing;

} H5P_mt_active_thread_count_t;

/******************************************************************************************
 *
 * struct H5P_mt_class_ref_counts_t
 *
 * Property lists (i.e. instances on H5P_mt_class_t in the new implementation) need to
 * maintain reference counts on the number of derived property list classes, the number
 * derived property lists, and whether the property list class still exists in the index.
 *
 * One can argue that these three ref counts should be combined into a single reference
 * count.  For now, at least, I am inclined to retain this design feature for the
 * following reasons:
 *
 * First, maintaining these reference counts separately seems likely to have some
 * debugging benefits, in that it provides more information about the current derivatives
 * of the property list class than a single reference count.
 *
 * Second, given that we must replicate the behavior of the current implementation quite
 * closely in the single thread case, it seems to me that gratuitous design changes
 * should be avoided.
 *
 * This, however raises the issue of how to keep the different reference counts
 * synchronized, and it particular, how to avoid the case in which the combined
 * reference counts drop to zero, discard is initialized, and another thread comes
 * in and tried to increment one of the reference counts.
 *
 * Solve this by combining the various reference counts into a single atomic structure,
 * and not allowing any reference count to be incremented once all the reference counts
```

```
 * have dropped to zero.
 *
 * This structure is intended to fulfill this role.  The individual fields are discussed
 * below.  Observe that the size of the structure is less that 128 bits, which should
 * allow true atomic operation on most modern machines.
 *
 * pl:          Number of property lists immediately derived from this property
 *              list class, and still extant.
 *
 * plc:         Number of property list classes immediately derived from this property
 *              list class, and still extant.
 *
 * deleted:     Boolean flag indicating whether this property list class has been deleted
 *              from the index.  This field is set to FALSE on creation, and set to TRUE
 *              when the reference count on the property list class in the index drops
 *              to zero.
 *
 * dummy_bool_1:
 * dummy_bool_2:
 * dummy_bool_3: The dummy_bool_? fields exist to pad H5P_mt_prop_aptr_t out to 128 bits,
 *              and allow prevention of insertion of garbage into an atomic instance of
 *              H5P_mt_prop_aptr_t, thus avoiding spurious failures of
 *              atomic_compare_exchange_strong().  They should always be set to false.
 ****************************************************************************************/

typedef struct H5P_mt_class_ref_counts_t {

   uint64_t pl;
   uint32_t plc;
   bool_    deleted;

   bool         dummy_bool_1;
   bool         dummy_bool_2;
   bool         dummy_bool_3;

}  H5P_mt_class_ref_counts_t


/****************************************************************************************
 * struct H5P_mt_class_sptr_t
 *
 * The H5P_mt_class_sptr_t combines a pointer to H5P_mt_list_t with a serial number
 * in a 128 bit package. It is intended to allow instances of H5P_mt_class_t to be
 * linked together in a singly linked list – specifically in a free list.
 *
 * This combination of a pointer and a serial number is needed to prevent ABA
 * bugs.
 *
 * ptr:         Pointer to an instance of H5P_mt_class_t.
 *
 * sn:          Serial number that should be incremented by 1 each time a new
 *              value is assigned to ptr.
 *
 ****************************************************************************************/

typedef H5P_mt_class_sptr_t {

  struct H5P_mt_class_t * ptr;
  unsigned long long int  sn;

} H5P_mt_list_sptr_t;


/****************************************************************************************
 *
```

```
* struct H5P_mt_class_t
*
* Revised version of H5P_genclass_t designed for use in a multi-thread safe version
* of the HDF5 property list module (H5P).
*
* At the conceptual level, a property list class is simply a template for constructing
* a default version of a member of a class of property lists, with the default properties,
* each with that property's default value.
*
* This simple concept is complicated by the requirement that modifications to property
* list classes can not affect preexisting derived property lists or property list classes.
*
* The single thread version of H5P addressed this problem by duplicating property list
* classes with derived property lists and/or property list classes whenever they are
* modified.  The modification is applied to the duplicate, and the duplicate replaces
* the base version in the index.  This approach has a number of problems, not the
* least being that it makes it possible for multiple versions of the property list
* class to exist in the index, and thus be visible to the user.
*
* Instead, the multi-thread version of property list classes maintains back versions
* of all properties tagged with the property list class version in which they were
* created (and possibly deleted).  All properties are ref counted with the number of
* properties in derived property lists that refer to them for default values.  Since
* the ref count on a version of a property can only be incremented when that property
* version appears in the current version of the property list class, this means that
* back versions of properties may be safely discarded once their ref counts drop to zero.
*
* Note that this no longer need be the case if we allow back version of property list
* classes to be visible outside of H5P.  Note also that this is a semantic change in
* the H5P API from the single thread version, albeit an obscure one, and to my thinking,
* very much in the right direction.
*
* More importantly, if all operations on a property list address a specific version,
* and all modifications are effectively atomic, concurrent operations can occur without
* the potential for data corruption as long as all modifications trigger an increment
* of the property list class version number.
*
* Making modifications to a property list class effectively atomic is slightly tricky,
* as, to give an obvious example, inserting a new property and incrementing the
* version number can't be made atomic without heroic measures.  However, by
* targeting every operation at a specific version, we can make changes in progress
* effectively invisible since new or modified properties are represented by new
* instances of H5P_mt_prop_t with creation property list class versions higher than
* the current version, and thus don't become visible until the property list class
* version is incremented to the point that they become visible.
*
* However, if multiple modifications to the property list class are in progress
* simultaneously, there is a race condition between the issue of a new version
* number to be used to tag a modification, and the increment of the property list
* class version number when the modification completes.
*
* Conceptually, this can be handled by waiting to increment the version number until
* the current version number is one less than the issued version number.  Use of a
* condition variable is the obvious solution here -- but we will sleep and try again
* until we settle on a threading package.
*
* There is also a potential race condition if a delete and either a modify or an
* insert on a single property is in progress at the same time.  In this case, the
* operations must proceed in target version issue order.
*
* A second fundamental difference between the single thread and the multi-thread
* implementations of the property list class, is that properties are stored on a
* lock free singly linked list (LFSLL) instead of a skip list.  This LFSLL list is
* sorted first by a hash on the property name, second by property name (to allow for
* hash collisions), and finally by creation version in decreasing order.
*
```

```
* Given that property lists are typically short (less than 25 properties), and that
* the LFSLL will be searched only on property insert, delete, or modification, the
* LFSLL should be near optimal for this application. However, if the number of
* properties (or back versions of same) balloon and cause performance issues, it
* will be easy enough to replace the LFSLL with a lock free hash table.
*
* Finally, for code simplicity, properties inherited from the parent property list
* class are copied into the LFSLL of properties in the derived property list class.
* There is nothing magic about this, and we can revert to the old system if there
* is some reason to do so.
*
* Note, however, that it is still necessary for property list classes to maintain
* pointers to their parent property list classes due to the requirement that
* all close functions in ancestor property list classes be called on close.
*
* With this outline of H5P_mt_class_t in hand, we now address individual fields.
*
*                                        JRM -- 5/22/24
*
* The fields of H5P_mt_prop_t are discussed individually below.
*
* tag:        Integer value set to H5P_MT_CLASS_TAG when an instance of H5P_mt_class_t
*             is allocated from the heap, and to H5P_MT_CLASS_INVALID_TAG just before
*             it is released back to the heap.  The field is used to validate pointer
*             to instances of H5P_mt_class_t,
*
* parent_id:   ID assigned to the immediate parent property list class in the index.
*             As the parent cannot be deleted until its ref counts drop to zero,
*             it must exist at least as long as this property list class.
*
*             This field is not atomic, as it is set before this property list class
*             is inserted in the index, and thus becomes visible to other threads.
*
* parent_ptr:  Pointer to the instance of H5P_mt_class_t that represents the immediate
*             parent property list class in the index.  As the parent cannot be
*             deleted until its ref counts drop to zero, it must exist and this pointer
*             must be valid at least as long as this property list class.
*
*             This field is not atomic, as it is set before this property list class
*             is inserted in the index, and thus becomes visible to other threads.
*
* parent_version: Version of the parent property list class from which this property
*             list class is derived.
*
* name:       Pointer to a string containing the name of this property list class.
*
*             This field is not atomic, as it is set before this property list class
*             is inserted in the index, and thus becomes visible to other threads.
*
*
* id:         Atomic instance of hid_t used to store the id assigned to this property
*             list class in the index.  This field is atomic, as it can't be set
*             until after the instance of H5P_mt_class_t is registered, and thus
*             visible to other threads.  That said, once set, this field will not
*             change for the life of the property list class.
*
* type:       Type of the property list class.  This field is constant for the life
*             of the property list class, and is set before the instance of
*             H5P_mt_class_t becomes visible to other threads.
*
* curr_version: Atomic uint64_t containing the current version of the property list
*             class.  This version number is incremented each time a modification to
*             the property list class is completed.
*
*             A uint64_t is used, as at present there is no provision for a roll over.
*             Given the relative infrequency of modifications to property list
```

```
*           classes, 64 bits is probably sufficient for all reasonable cases.
*           However, a roll over must never occur, and an error should be flagged
*           if it does.
*
*           To allow for an undefined deletion version, the curr_version must be
*           no less than 1.
*
* next_version: Atomic uint64_t containing the version number to be assigned to the
*           next modification of the property list class.
*
*           When no modifications to the property list class are in progress,
*           next_version should be one greater than curr_version.
*
*           When a modification to a property list class begins, it does a fetch
*           and increment on next_version, performs its changes and tags them with
*           the returned version, and finally increments the curr_version.
*
*           Note, that to avoid exposure of partial modifications, increments
*           to curr_version must be executed in next_version issue order.  Thus, a
*           thread that modifies the property list class, must not increment
*           curr_version until its value is one less that the version number it
*           obtained when it started.
*
*                       Further, if a modify or insert and a delete on the same property are
*           active at the same time, they must be executed in issue order.
*
* pl_head:     Atomic Pointer to the head of the LFSLL containing the list of properties
*           (i.e. instances of H5P_mt_prop_t) associated with the property list class.
*           Other than during setup, this field will always point to the first node
*           in the list whose value will be negative infinity.
*
*           Entries in this list are sorted first by a hash on the property name,
*           second by property name (to allow for hash collisions), and finally
*           by creation version in decreasing order.  Other than during setup,
*           the first and last entries in the list list will be sentinel entries
*           with hash values (conceptually) of negative and positive infinity
*           respectively.
*
*
* log_pl_len:  Number of properties defined in the property list class at the
*           current version.  Note that, in general, this value will not be
*           correct for all versions of the property list class, and will be
*           briefly inaccurate even for the current version during property
*           insertions and deletions.  Thus when an exact value is required,
*           the property list must be scanned for the correct value for the
*           desired version.
*
* phys_pl_len: Number of instances of H5P_mt_prop_t in the property list.  This
*           number includes sentinel nodes, and both current and superseded
*           instances of H5P_mt_prop_t.  Note that this value will be briefly
*           incorrect during property insertions, deletions, and modifications.
*           Modification of a property causes this value to change, and for
*           modification, a new instance H5P_mt_prop_t is inserted with the
*           desired changes and a new creation version.
*
* ref_count:   Atomic instance of H5P_mt_class_ref_counts_t which combines:
*
*             the number of property lists immediately derived from this
*             property list class, and still extant (ref_count.pl),
*
*             the number of property lists classes immediately derived from
*             this property list class, and still extant (ref_count.plc),
*
*             and a boolean flag (ref_count.deleted) indicating whether this
*             property list class has been deleted from the index.
*
```

```
*           into a single atomic structure – thus ensuring synchronization between
*           these three different values.
*
*           Once ref_count.pl and ref_count.plc have dropped to zero, and deleted
*           is set to TRUE, the property list class may be discarded.  Further,
*           neither ref_count.pl nor ref_count.plc may be incremented once this
*           condition is obtained.
*
*           Further, observe that once this condition holds, the reference counts
*           on all versions of all properties in the property list class must be
*           zero.
*
* The following fields are pointers to the callback functions associated with the
* property along with pointers to data to be passed to these functions when called.
* These are combined with a Boolean indicating whether all callbacks are thread safe.
* If this flag is not set, all callbacks must be protected by the global mutex.
*
* The descriptions of the callbacks are all taken from Matt Larson's "Census of H5P
* Callbacks", and are a major improvement on the existing documentation.  These
* descriptions may have to be modified to reflect the re-implementation of H5P.
*
* Quoting from Matt's document:
*
*    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.
*
* We need to determine if there are any other uses for these callbacks.
*
* callbacks_mt_safe: Boolean flag used to indicate whether all callbacks are
*           multi-thread safe.  If this field is not set, the global mutex must
*           be held when the callbacks are called.
*
* create_func: Function to call when a property list is created.
*
*           Signature:
*
*               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.
*
* create_data: Pointer to user data to pass along to create callback.
*
* copy_func:   Function to call when a property list is copied.
*
*           Signature:
*
*              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.
*
* copy_data:   Pointer to user data to pass along to copy callback.
*
* close_func:  Function to call when a property list is closed.
*
*           Signature:
*
*              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.
 *
 * close_data:  Pointer to user data to pass along to close callback.
 *
 *
 * Free list and shutdown management fields:
 *
 * thrd:       Atomic instance of H5P_mt_active_thread_count_t.  This field is used
 *           verify that no threads are active in an instance of H5P_mt_class_t
 *           during setup or takedown of the structure prior to discard.  See
 *           the header comment on H5P_mt_active_thread_count_t for a discussion of
 *           how its fields must be maintained when a thread wants to access the
 *           host instance of H5P_mt_class_t.
 *
 * fl_next:     Atomic instance of H5P_mt_class_sptr_t – a pointer to H5P_mt_class_t
 *           augmented with a serial number to avoid ABA bugs.  This field is
 *           included to support a free list of instances of H5P_mt_class_t.
 *
 *
 * Statistics:
 *
 * TBD.
 *
 ******************************************************************************************/

#define H5P_MT_CLASS_TAG
#define H5P_MT_CLASS_INVALID_TAG

typedef struct H5P_mt_class_t {

   uint32_t                          tag;
   hid_t                             parent_id;
   H5P_mt_class_t *                  parent_ptr;
   uint64_t                          parent_version;
   char *                            name;
   _Atomic hid_t                     id;
   H5P_plist_type_t                  type;
   _Atomic uint64_t                  curr_version;
   _Atomic uint64_t                  next_version;

   /* List of properties, and related fields */
   H5P_mt_prop_t *                   pl_head;
   _Atomic uint32_t                  log_pl_len;
   _Atomic uint32_t                  phys_pl_len;

   /* reference counts */
   _Atomic H5P_mt_class_ref_counts_t ref_count;

   /* Callback function pointers & info */
   H5P_cls_create_func_t             create_func;
   void *                            create_data;
   H5P_cls_copy_func_t               copy_func;
   void *                            copy_data;
   H5P_cls_close_func_t              close_func;
   void *                            close_data;

   /* shutdown and free list management fields */
   _Atomic H5P_mt_active_thread_count_t thrd;
   _Atomic H5P_mt_class_sptr_t       fl_next;

   /* Stats: *;

} H5P_mt_class_t;
```

Property lists are stored in instances of H5P_mt_list_t. See below for the definition of this structure, and its supporting structures H5P_mt_list_prop_ref_t and H5_mt_list_table_entry_t.

As with H5P_genclass_t, H5P_mt_list_t is a version of H5P_genlist_t modified to support multi-thread. The major changes from the single thread implementation are outlined as follows:

Like the single thread implementation of property lists, the multi-thread implementation refers to its parent property list class for default versions of all inherited properties, and keeps local copies of all modified or inserted properties. However, the mechanisms are quite different.

First, the modified or inserted properties (instances of H5P_mt_prop_t) are stored on a LFSLL. As with the property list class, there is one instance of H5P_mt_prop_t for each version of the property in question, marked with its creation version, and if appropriate, its delete_version.

Second, when a property list is created, a table of all inherited properties is created, stored in an array of H5P_mt_list_table_entry_t, and sorted first by hash code on the property name, and then by property name.

Each entry in this table contains a base pointer and version, and a current pointer and version. The current pointer and version are initialized to NULL and zero respectively. A base delete version field is also supplied to allow for the deletion of an unmodified, inherited property from the property list – this field is initialized to zero.

The base pointer is set to point to the instance of H5P_mt_prop_t representing the inherited property with its value at the time the new property list was created[19]. Due to the versioning of property list classes, these instances of H5P_mt_prop_t are never modified, and will not be deleted until after all derived property lists that depend on them have been deleted. Thus the lookup of the default values of all inherited properties can be done in O(log n) time, where n is the number of inherited properties.

The base version fields of the table entries are always set to 1 – the initial version of the property list.

Third, when an inherited property is modified, a new instance of H5_mt_prop_t is created with the appropriate value and creation version, and inserted in the LFSLL of modified or inserted properties in sorted order. Then the current pointer and version fields of the appropriate entry in the table of inherited properties are set to the address and version of the new instance of H5P_mt_prop_t respectively. Finally, as per property list classes, the current version of the property list is incremented to make the new value of the property visible.

---

[19] But note that properties on property list classes that have create callbacks must be copied to the new property list. The current design doesn't address this point, which is left pending a better understanding of why this callback exists.

Inserted properties, or modifications of same are handled the same way, only they do not appear in the table of inherited properties, and thus must be found via a search of the LFSLL. Since the number of inserted properties is tracked, this search can be skipped when this number is zero. The presumption here is that inserted properties in property lists are rare – if this presumption proves false, a more efficient lookup mechanism should be retrofitted.

As per property list classes, operations that modify the property list class must increment the curr_version field in issue order. Also per property list classes, multiple operations on the same property must be executed in issue order[20]. Also per property list classes, all read operations on the property list must be directed at a version no greater than this value at the point at which the operation starts. This has the effect of making the read effectively atomic as any subsequent modifications to the property list will not be visible to the read.

In principle, property lists are accessed only via their IDs in the index – within the HDF5 library, code wishing to access a property list should increment the reference count on its ID, use this ID to look up its pointer in the index, and not decrement the property lists ref count until it has discarded its pointer. Similarly, public API calls that receive property list IDs as parameters should increment the reference counts on these IDs on entry, and decrement them on exit. If this is done religiously, and H5I does what it is supposed to, it should be impossible for a property list to be deleted out from under a thread, or for thread to access a property list after its reference count in the index drops to zero[21].

This, of course, presumes that all code that deals with property lists is well behaved. While we may do well in the HDF5 library proper, we have no control over user code, or other VOL connectors for that matter. Further, H5I has a facility for exchanging the void pointers associated with IDs. While I see no reason why this facility should be used with property lists, that doesn't mean it will not be.

The bottom line is that we need some mechanism to exclude threads from property lists while they are in the process of being discarded[22].

---

[20] Avoiding this added bit of complexity would be more difficult with property lists. In addition to inserting a tombstone instance of H5P_mt_prop_t into the LFSLL, and deleting the deletion_version field from H5P_mt_prop_t, it would be necessary to insert sentinel entries for each inherited property in the LFSLL, and modify the lookup table to point to these sentinels instead of the current version. As per property list classes, the expected infrequency of simultaneous modifications to property lists, suggest that it is probably simplest just to serialize these operations, since I see no way of avoiding the requirement that curr_versions increments occur in issue order.

[21] To see this, observe that after an ID's reference count drops to zero, it is removed from the index, and searches on the ID will fail.

[22] We don't need this for property list creation since a new property list is created by a single thread, and does not become visible to other threads until it is inserted in the index.

For now at least, we do this by combining a counter of the number of threads active in the property list with a closing flag in an atomic structure.

When any thread wants to access a property list it must first do an atomic fetch on this structure, and return failure if the closing flag is set. If it isn't, it must increment the active threads count its local copy, overwrite the shared copy with an atomic compare exchange strong, and repeat the procedure if the compare exchange fails. On exiting the property list, it must decrement the active threads count atomically regardless of the value of the closing flag.

When a thread attempts to discard a property list, it must do an atomic fetch on the structure, and throw an assert if the closing flag is set, since we must never have more than one thread attempting to discard a given property list. Otherwise, it sets the closing flag on the local copy, attempts to overwrite the shared copy with an atomic compare exchange strong, and repeats the procedure if the atomic compare exchange strong fails. It must then wait until the number of active threads drops to zero before proceeding with the deletion.

Observe that if the reference counts are managed correctly, and this mechanism works correctly, a free list for instances of H5P_mt_list_t is unnecessary, since all threads are prevented from accessing such an instance once it is marked as closing and all threads have drained.[23]

See the header comments for the definitions of H5P_mt_list_t, and its supporting structures H5P_mt_list_prop_ref_t and H5_mt_list_table_entry_t for further details.

```
/***********************************************************************************
 * struct H5P_mt_list_sptr_t
 *
 * The H5P_mt_list_sptr_t combines a pointer to H5P_mt_list_t with a serial number
 * in a 128 bit package. It is intended to allow instances of H5P_mt_list_t to be
 * linked together in a singly linked list – specifically in a free list.
 *
 * This combination of a pointer and a serial number is needed to prevent ABA
 * bugs.
 *
 * ptr:        Pointer to an instance of H5P_mt_list_t.
 *
 * sn:         Serial number that should be incremented by 1 each time a new
 *             value is assigned to ptr.
 *
 ***********************************************************************************/

typedef H5P_mt_list_sptr_t {

  struct H5P_mt_list_t *  ptr;
  unsigned long long int  sn;
```

[23] All this said, I think it would be prudent to maintain a free list to facilitate useful error detection and reporting. Further, during initial testing, keeping all discarded instances of H5P_mt_list_t on the free list until test completion would have the advantage of ruling out this class of error initially. Due to the subtlety of errors caused by premature freeing or reuse of structures in lock free multi-thread algorithms, avoiding the possibility until the code is otherwise reasonably well tested has worked well for me.

```
} H5P_mt_list_sptr_t;


/****************************************************************************************
 *
 * struct H5P_mt_list_prop_ref_t
 *
 * Struct H5P_mt_list_prop_ref_t is structure designed to contain a pointer to an instance
 * of H5P_mt_prop_t and a version number of a single atomic structure.  This is necessary,
 * as when an entry in the H5P_mt_list_table_entry_t is updated, we need to update both
 * the pointer to an instance H5P_mt_prop_t and the version number in a single atomic
 * operation.
 *
 * This structure is 128 bits, which allows true atomic operation on many (most?) modern
 * CPUs.
 *
 * The structure is used in two contexts:
 *
 * First to point to the instance of H5P_mt_prop_t in the property list class from
 * which the host property list was derived.  In this case, version number should be
 * the initial version of the host property list class.
 *
 * Second, if the default value of the property has been overwritten, to point to an
 * instance of H5P_mt_prop_t in the host property list class's LFSLL of modified or
 * added properties.  In this case, the ver field must match the create_version field
 * of the instance of H5P_mt_prop_t pointed to by ptr.
 *
 * The individual fields of the structure are discussed below:
 *
 * ptr:        Pointer to an instance of H5P_mt_prop_t, or NULL.
 *
 * ver:        Version number of the host property list class at which this
 *             this pointer was set.
 *
 ****************************************************************************************/

typedef struct H5P_mt_list_prop_ref_t {

   H5P_mt_prop_t *     ptr;
   uint64_t            ver;

} H5P_mt_list_prop_ref_t;


/****************************************************************************************
 *
 * struct H5P_mt_list_table_entry_t
 *
 * An array of instances of H5P_mt_list_table_entry_t is used to create a look up
 * table for properties inherited from the parent property list class.
 *
 * chksum:     int64_t containing a 32 bit checksum computed on the name field
 *             below.
 *
 *             Since this field is constant for the life of the instance of
 *             H5P_mt_prop_t, and is set before the instance is visible to more
 *             than one thread, it need not be atomic.
 *
 * name:       Pointer to a dynamically allocated string containing the name of the
 *             property.  This field is not atomic, as the string should be allocated,
 *             and initialized, and the name field set before the instance of
 *             H5P_mt_prop_t is visible to more than one thread.  Since the name
 *             is constant for the life of the instance of H5P_mt_prop_t, this should
 *             be sufficient for thread safety.
```

```
 *
 * base:      Atomic instance of H5P_mt_list_prop_ref_t.  The ptr field points to
 *            the instance of H5P_mt_prop_t in the parent property list class, and
 *            the ver field should contain the initial version number of the
 *            property list.
 *
 *            Note that if the instance of H5P_mt_pro_t in the parent property list class
 *            has a create callback, we must copy the property into the new property list,
 *            and not use the property in the parent property list class as the initial
 *            value of the property.  In this case, base.ptr is set to NULL, and base.ver
 *            set to 0, the copy is inserted into the lock free singly linked list, and
 *            curr.ptr points to the copy of the property until such time as the value of
 *            value of the property is modified.  In this case, curr.ver is initialized to
 *            the initial version of the property list.
 *
 * base_delete_version: Property lists derived from a property list class must not
 *            modify properties in the parent property list class.  Thus they
 *            must maintain their own create and delete versions.  The create
 *            version is simply the initial version of the property list, and
 *            is stored in the base.ver field.  However, if the property is
 *            deleted from the property list, we must have a delete version to
 *            indicate the property list version at which this took place.
 *
 *            The atomic uint64_t base_delete_version exists to serve this purpose.
 *            if the base version of the property has not been deleted, this field
 *            will contain zero. Once set to a non-zero value, it will never change
 *            for the life of the property list.
 *
 *
 * curr:      Atomic instance of H5P_mt_list_prop_ref_t whose ptr and ver fields
 *            must be initialized to NULL and 0 respectively.
 *
 *            If the value of the inherited property is modified, a new instance of
 *            H5P_mt_prop_t is allocated, copying the tag, sentinel, chksum, name,
 *            and callback fields from the most recent version of the property
 *            pointed to by either base.ptr or curr.ptr.
 *
 *            The in_prop_class, and ref_count fields are set to zero, and not used
 *            in property lists.  The create_version is set to the version of the
 *            property list in which the modified version is set, and the
 *            delete_version is set to 0.  The value field is set to point to the
 *            new value of the property, and the new instance of H5P_mt_prop_t is
 *            inserted into the LFSLL of new / modified properties associated with
 *            the host property list.
 *
 *            Next, curr.ptr is set to point to the new instance, and curr.ver is
 *            set to its creation version.  Recall that both of these fields are
 *            set in a single atomic operation.
 *
 *            Finally the version of the host property list is incremented to make
 *            these changes visible.
 *
 ****************************************************************************************/

typedef struct H5P_mt_list_table_entry_t {

   int64_t                        chksum;
   char *                         name;
   _Atomic H5P_mt_list_prop_ref_t     base;
   _Atomic uint64_t               base_delete_version;
   _Atomic H5P_mt_list_prop_ref_t     curr;

} H5P_mt_list_table_entry_t;


/****************************************************************************************
```

```
*
* struct H5P_mt_list_t
*
* Revised version of H5P_genlist_t designed for use in a multi-thread safe version
* of the HDF5 property list module (H5P).
*
* At the conceptual level, a property list class is simply a list of properties -- i.e.
* name value pairs.
*
* When a property list is created, it incorporates a list of properties with default
* values from its parent property list class at the version at which the creation of
* the property list was started.  At this time, the number of properties in the property
* list class is determined and stored in the nprops_inherited field.
*
* This done, an array of H5P_mt_list_table_entry_t of length nprops_inherited is
* allocated, with the base.ptr field pointing to the associated instance of H5P_mt_prop_t
* in the parent property list classes LFSLL of properties, and the base.ver field set
* to the initial version of the property list.  After this array is initialized, it
* is sorted by chksum and then name, and the lkup_table field is set to point to it.
* Observe that this table allows a log n lookup of properties inherited from the
* parent property list class.
*
* If the value of any inherited property is modified, a new instance of H5P_mt_prop_t
* is created with the modified value and creation version, and inserted into the
* property list's LFSLL of properties.  The curr.ptr field of the appropriate entry
* in the lookup table is set to point to it, and the curr.ver field is set to the
* version of the property list at which the modification was made.  Finally, the
* property list's curr_version field is incremented to make this modification visible.
*
* If a new property is added to the property list, it is simply inserted into the
* property list's LFSLL of instances of H5P_mt_prop_t, and the nprops_added field is
* incremented.  On searches, the lookup table is searched first, with the LFSLL being
* searched only if this first search fails, and nprops_added is positive.
*
* As with the multi-thread version of the property list classes, property lists
* maintain back versions of all properties tagged with the property list version
* in which they were created (and possibly deleted).  Unlike property list classes,
* the properties are not reference counted.
*
* All operations on a property list must address a specific version, which must be no
* greater than the current version at the start of the operation. As shall be seen,
* this allows us to make all modifications effectively atomic, which in turn allows
* concurrent operations to occur without the potential for data corruption.
*
* As with property list classes, making modifications to a property list effectively
* atomic is slightly tricky, but can be handled in much the same way.  Since every
* operation on a property list is targeted at a specific version, we can make changes in
* progress effectively invisible since new or modified properties are represented by
* new instances of H5P_mt_prop_t with creation property list versions higher
* than the current version, and thus don't become visible until the property list
* version is incremented to the point that they become visible..
*
* However, as with property list classes, if multiple modifications to the property list
* are in progress simultaneously, there is a race condition between the issue of a new
* version number to be used to tag a modification, and the increment of the property list
* version number when the modification completes.
*
* As with property list classes, this can be usually be handled by waiting to increment
* the version number until the current version number is one less than the issued
* version number.  However, if a modify or insert and a delete on the same property are
* active at the same time, they must be executed in issue order.
*
* Also as per property list classes, modified / new properties are stored on a
* lock free singly linked list (LFSLL) instead of a skip list.  This LFSLL list is
* sorted first by a hash on the property name, second by property name (to allow for
* hash collisions), and finally by creation version in decreasing order.
```

```
*
* Since only new / modified properties are stored on this list, it should be shorter
* than the similar list in property list classes.  More importantly, the latest
* version of each inherited property is pointed to by the appropriate entry in the
* lookup table.  Added properties still require a linear search through the LFSLL.
* If this proves to be a performance issue, we can either keep added entries in a
* different list, or allow the lookup table to be extended when new entries are added.
*
* Property lists are stored in the index, and should only be accessed via their IDs.
* Within HDF5, the reference count on the property list ID should be incremented before
* its pointer is looked up in the index, and should not be decremented until the code
* in question is done with the property list.  If this is rule is followed religiously,
* it should be impossible for a property list to be deleted our from under a thread, or
* for any thread to access a property list after its reference count drops to zero and
* it is removed from the index and discarded.
*
* However, since any failure of this mechanism will be hard to diagnose, an instance
* of H5P_mt_active_thread_count_t is included in H5P_mt_list_t, and must be maintained.
* The protocol for doing this is discussed in the header comment for
* H5P_mt_active_thread_count_t.  In the context of H5P_mt_list_t, a positive thread
* count on discard is an error and should trigger an assertion failure.
*
* Similarly, H5P_mt_list_t contains an instance of H5P_mt_list_sptr_t to support a
* free list that shouldn't be necessary, but which is probably prudent for much the
* same reason.
*
* With this outline of H5P_mt_list_t in hand, we now address individual fields.
*
* tag:        Integer value set to H5P_MT_LIST_TAG when an instance of H5P_mt_list_t
*             is allocated from the heap, and to H5P_MT_LIST_INVALID_TAG just before
*             it is released back to the heap.  The field is used to validate pointers
*             to instances of H5P_mt_list_t,
*
* pclass_ptr:  Pointer to the instance of H5P_mt_class_t from which the property list
*             was derived.
*
*             This field is not atomic, as it is set before this property list is
*             inserted in the index, and thus becomes visible to other threads.
*
* pclass_id:   ID of the instance of H5P_mt_class_t from which the property list
*             was derived.
*
*             This field is not atomic, as it is set before this property list is
*             inserted in the index, and thus becomes visible to other threads.
*
* pclass_version: Version of the parent property list class from which this property
*             list was derived.
*
*             This field is not atomic, as it is set before this property list is
*             inserted in the index, and thus becomes visible to other threads.
*
* plist_id:    ID assigned to this property list.  This field must be atomic, because
*             the instance of H5P_mt_list_t becomes visible to other threads before
*             this field can be set.  That said, once it is set, it should not
*             change for the lifetime of the property list.
*
* curr_version: Atomic uint64_t containing the current version of the property list.
*             This version number is incremented each time a modification to the
*             property list is completed.
*
*             A uint64_t is used, as at present there is no provision for a roll over.
*             Given the relative infrequency of modifications to property lists,
*             64 bits is probably sufficient for all reasonable cases.  However, a roll
*             over must never occur, and an error should be flagged if it does.
*
*             To allow for an undefined deletion version, the curr_version must be
```

```
*          no less than 1.
*
* next_version: Atomic uint64_t containing the version number to be assigned to the
*          next modification of the property list.
*
*          When no modifications to the property list are in progress,
*          next_version should be one greater than curr_version.
*
*          When a modification to a property list begins, it does a fetch
*          and increment on next_version, performs its changes and tags them with
*          the returned version, and finally increments the curr_version.
*
*          Note, that to avoid exposure of partial modifications, increments
*          to curr_version must be executed in next_version issue order.  Thus, a
*          thread that modifies the property list class, must not increment
*          curr_version until its value is one less that the version number it
*          obtained when it started.
*
* lkup_tbl:    Pointer to an array of H5P_mt_list_table_entry_t that permits fast
*          lookup of properties inherited from the parent property list class.
*
*          See the header comment for H5P_mt_list_table_entry_t for further
*          details.
*
* nprops_inherited: The number of properties inherited from the parent property list
*          class, and also the number of entries in the lookup table (lkup_tbl)
*          above.  Note that any or all of these properties may be deleted
*          in an arbitrary version of the property list.
*
* nprops_added: The number of properties added to the property list after its
*          creation.  Note that these properties do not appear in the lkup_tbl,
*          and thus if a search for a property fails in lkup_tbl and nprops_added
*          is positive, the LFSLL pointed to by pl_head (below) must also be
*          searched.
*
* nprops:      Number of properties defined in the current version of the property
*          list.  Note that this value may be briefly inaccurate during property
*          additions or deletions -- if an accurate value is required, the LFSLL
*          pointed to by pl_head must be scanned for the target property list
*          version.
*
* pl_head:     Atomic Pointer to the head of the LFSLL containing the list of modified
*                         or inserted properties (i.e. instances of H5P_mt_prop_t) associated with
*          the property list.  Other than during setup, this field will always point
*          to the first node in the list whose value will be negative infinity.
*
*          Entries in this list are sorted first by a hash on the property name,
*          second by property name (to allow for hash collisions), and finally
*          by creation version in decreasing order.  Other than during setup,
*          The first and last entries in the list list will be sentinel entries
*          with hash values (conceptually) of negative and positive infinity
*          respectively.
*
* log_pl_len:  Number of properties defined in the property list at the
*          current version.  Note that, in general,  this value will not be
*          correct for all versions of the property list class, and will be
*          briefly inaccurate even for the current version during property
*          insertions and deletions.  Thus when an exact value is required,
*          the property list must be scanned for the correct value for the
*          desired version.
*
* phys_pl_len: Number of instances of H5P_mt_prop_t in the property list.  This
*          number includes sentinel nodes, and both current and superseded
*          instances of H5P_mt_prop_t.  Note that this value will be briefly
*          incorrect during property insertions, deletions, and modifications.
*          Modification of a property causes this value to change, as for
```

```
 *          modification, a new instance H5P_mt_prop_t is inserted with the
 *          desired changes and a new creation version.
 *
 * class_init:  True iff the class initialization callback finished successfully.
 *
 *
 * Free list and shutdown management fields:
 *
 * thrd:      Atomic instance of H5P_mt_active_thread_count_t.  This field is used
 *          verify that no threads are active in an instance of H5P_mt_list_t
 *          during the takedown of the structure prior to discard.  See the
 *          header comment on H5P_mt_active_thread_count_t for a discussion of
 *          how its fields must be maintained when a thread wants to access the
 *          host instance of H5P_mt_list_t.
 *
 * fl_next:    Atomic instance of H5P_mt_list_sptr_t – a pointer to H5P_mt_list_t
 *          augmented with a serial number to avoid ABA bugs.  This field is
 *          included to support a free list of instances of H5P_mt_list_t.
 *
 *
 * Statistics:
 *
 * TBD.
 *
 ****************************************************************************************/

#define H5P_MT_LIST_TAG
#define H5P_MT_LIST_INVALID_TAG

typedef struct H5P_mt_list_t {

   uint32_t                  tag;

   H5P_mt_class_t *              pclass_ptr;
   hid_t                     pclass_id;
   uint64_t                  pclass_version;


   _Atomic hid_t                 plist_id;

   _Atomic uint64_t              curr_version;
   _Atomic uint64_t              next_version;

   H5P_mt_list_table_entry_t *      lkup_tbl;

   uint32_t                  nprops_inherited;
   _Atomic uint32_t              nprops_added;
   _Atomic uint32_t              nprops;

   /* List of properties, and related fields */
   H5P_mt_prop_t *               pl_head;
   _Atomic uint32_t              log_pl_len;
   _Atomic uint32_t              phys_pl_len;

   _Atomic bool                  class_init;

   /* shutdown and free list management fields */
   _Atomic H5P_mt_active_thread_count_t thrd;
   _Atomic H5P_mt_list_sptr_t        fl_next;

   /* Stats: */

} H5P_mt_list_t;
```

Public API Function Outlines

In this section, we list all public H5P API calls. Ideally, we would outline the necessary processing for each call in the context of the new design. However, for schedule reasons, the current version only includes outlines for a representative sample of those calls that involve non-trivial operations on the property list class and/or property list data structures.

At first glance, this would seem redundant, as the current implementations of these functions are discussed in considerable detail in Appendix 1. However, as we are re-writing the property list code, it is necessary to review all of these functions to verify that the new design will allow us to implement the existing functionality, and to outline the implementation of these functions in the context of the revised data structures. This section and the subsequent section on internal H5P API calls serves this function.

In these outlines, I have made no particular assumptions as to organization of the code. That said, given the similarity between the internal and external APIs, it will probably be convenient to implement the internal APIs first, and then implement the public APIs using the internal API calls.

Finally, there are a number of relatively subtle proposed semantic changes in some of the APIs, which are noted where appropriate. In my humble opinion, these are improvements, but we may or may not be able to sell them to the HDF group. Given this point, we should retain the ability to duplicate the oddities of the existing implementation.

H5Pclose()

Signature:

```
H5_DLL herr_t H5Pclose(hid_t plist_id);
```

Description:

> Decrement the ref count on the target property list. If it drops to zero, remove it from the index, and delete it.

Outline of Processing

- If the supplied hid_t is H5P_DEFAULT, do nothing and return.
- If the supplied hid_t is not associated with a property list, flag an error and return.
- Otherwise, call H5I_dec_app_ref() which calls H5I__dec_app_ref(), which calls H5I__dec_ref(), and return.

From the perspective of re-implementing H5Pclose() this call to H5I__dec_ref() is where the action is. That function obtains a pointer to instance of H5I_mt_id_info_t associated with the supplied hid_t, and attempts to decrement its reference count.

If this count is greater than one, the reference count is simply decremented, and H5I__dec_ref() returns.

However, if the count is one, the free function associated with the id (which is H5P__close_list_cb() in the current implementation) is called, and (if the free function doesn't fail) the target entry in the index is deleted. Note that H5I ensures that only one call to the free function is active at any point in time for any given entry in the index.[24]

In the current implementation, H5__close_list_cb() simply calls H5P_close() and returns. Assuming we keep this general architecture, H5P_close() will be re-written as follows:

Signature:

```
herr_t H5P_close(H5P_mt_list_t *plist)
```

Outline of Processing:

- Set the closing flag on the target property list, and wait for all threads to drain so we don't delete the property list out from under a thread.

  To do this, atomically load plist→thrd in to a local instance of H5P_mt_active_thread_count_t – call this variable local_thrd.

  If local_thrd.closing is TRUE, flag an error and return.

  Otherwise, set local_thrd.closing to TRUE, and attempt to overwrite plist→thrd with a compare exchange strong. Repeat this process until failure (i.e. some other thread sets the closing flag) or until the compare exchange strong is successful.

- Atomically load list→thrd into local_thrd. If local_thrd.opening is FALSE and local_thrd.count is zero, proceed. Otherwise, wait a bit and try again.

  Since no other thread is allowed to begin any operation on *plist if either the opening or closing flags in plist→thrd are set, and since any thread that operates on *plist (other than the create and discard threads) must increment plist→thrd.count before it does anything, and decrement it when done, once plist→thrd.count drops to zero, we are

[24] That said, we should verify this with sanity checking code in the free_func provided to H5I.

guaranteed that no thread will touch any part of *plist other than the thrd field. Thus we can safely take down the property list and lookup table.

Note that this does not apply to the instance of H5P_mt_list_t itself, which must be kept on a free list until we are sure that no thread retains a pointer to it.

- Test to see if the property list initialization function completed (i.e. plist→class_init == TRUE). If it did, execute the close function of the parent property list class (i.e. plist→pclass→close_func) if it exists, along with those of any property list classes from which the parent property list class was derived (i.e. plist→pclass→parent→close_func, etc).

- Scan the lookup table. For each entry, if the base.ptr field is not NULL[25], call the close function if it exists on the base version of the property on the parent property list class (i.e.call plist→lkup_tbl[i].base→close if it is not NULL). Then atomically decrement the reference count on the base version of the inherited property (plist→lkup_tbl[i].base→ref_count).

- Scan the lock free SLL of instances of H5P_mt_prop_t (plist→pl_head). Neglecting the sentinel entries, if the close callback exists, call it on the associated entry value.

- Decrement the property list reference count on the parent property list class (plist→pclass_ptr→ref_count.pl). In the current code, this is accomplished via a call to H5P__access_class(plist->pclass, H5P_MOD_DEC_LST), which decrements the property list reference count. It then checks to see if the property list reference count and the property list class property list class reference count have dropped to zero, and the property list class has been deleted from the index. If so, it deletes the property list class.

  The re-implementation of the property lists will require a similar call, with the difference that the reference counts maintained by the property list class must be modified in an atomic operation.

  The details of shutting down and discarding a property list are discussed under XXX.

- Clean up the property list data structure:
  - For each entry in the lock free SLL of instances of H5P_mt_prop_t (first element pointed to by plist→pl_head), remove it from the lock free SLL, and discard the string containing the name. The close call should have discarded the value

[25] Recall that if the property has a create callback, the property in the source property list class must be copied into the property list – and in this case base.ptr is NULL.

(plist→value.ptr), so nothing to do there. Then discard the instance of H5P_mt_prop_t itself, being sure to set the tag to something invalid first.[26]

- Discard the lookup table (plist→lkup_tbl) and its associated strings.

- Discard the instance of H5_mt_list_t proper, being sure to set its tag field to something invalid first.[27]

H5Pclose_class()

Signature:

```
H5_DLL herr_t H5Pclose_class(hid_t pclass_id);
```

Description:

Decrement the reference count on the target property list class. If this reference count drops to zero, the class is removed from the index, and is marked as deleted. The class is not actually discarded until both the number of property lists that instantiate it, and the number of classes derived from it drop to zero as well.

Outline of Processing:

H5Pcopy()

Signature:

```
H5_DLL hid_t H5Pcopy(hid_t plist_id);
```

Description:

Duplicate the supplied property list or property list class, insert the duplicate in the appropriate index, and return the associated hid_t.[28]

Outline of Processing:

---

[26] Just freeing the instance should be safe. However, to simplify debugging, it may be useful to put the instance on a free list, and keep it there until the test is done. If it turns our that there is a pointer to the instance of H5P_mt_prop_t out there, and it is de-referenced at some point, this allows us to catch any such errors more gracefully. After all known bugs have been addressed, we can either alter the free list to allow re-use, or simply release instances of H5P_mt_prop_t to the heap immediately.

[27] Unlike instances of H5P_mt_prop_t, we can't free instances of H5P_mt_list_t immediately since it is possible that some thread still has a pointer to it. Thus instances of H5P_mt_list_t must be kept on a free list until we are sure that no thread retains a pointer. Need to develop a heuristic for this.

[28] Note the discrepencies between the documentation on this public API call and the code detailed in the discussion of H5Pcopy() in Appendix 1.

H5Pcopy_prop()

Signature:

```
H5_DLL herr_t H5Pcopy_prop(hid_t dst_id, hid_t src_id, const char *name);
```

Description:

Copy a property from one property list or property list class to another.

Outline of Processing:

H5Pcreate()

Signature:

```
H5_DLL hid_t H5Pcreate(hid_t cls_id);
```

Description:

Create a new property list derived from the property list class indicated by the supplied property list class id, insert it in the index, and return the id associated with the new property list.

Outline of Processing

- Verify that the supplied cls_id is in fact associated with a property list class, and obtain a pointer to the associated instance of H5P_mt_class_t – call this pointer pclass.

- Atomically load pclass→ref_count into a local instance of H5P_mt_class_ref_counts_t – call this local instance local_cls_ref_counts.

  If the pl and plc fields of local_cls_ref_counts are 0 and the deleted flag is TRUE, return failure[29].

  Otherwise, increment local_cls_ref_counts.pl and attempt to overwrite pclass→ref_counts with local_cls_ref_counts using an atomic compare exchange strong.

  Repeat this process until successful, or until it returns failure. Observe that if successful, this operation will prevent the property list class from being deleted out from under us while we are constructing the property list.

[29] This should be impossible, since we incremented the reference count on cls_id. If it happens, it probably means that there are some extra ref count decrements on cls_id elsewhere.

- Allocate an instance of H5P_mt_list_t and set plist to point to it. Do preliminary initialization as follows:

```
H5P_mt_active_thread_count_t init_thrd    = {0, TRUE, FALSE};
H5P_mt_list_sptr_t          init_fl_next = {NULL, 0};

plist→tag = H5P_MT_LIST_TAG;

plist→pclass_ptr = pclass;
plist→pclass_id = cls_id;
plist→pclass_version = atomic_load(&(pclass→curr_version));

atomic_init(&(plist→plist_id), 0);  /* will overwrite this */

atomic_init(&(plist→curr_version), 1);
atomic_init(&(plist→next_version), 2);

plist→lkup_tbl = NULL;  /* will overwrite this */

plist→nprops_inherited = 0; /* will overwrite this */
atomic_init(&(plist→nprops_added), 0);
atomic_init(&(plist→nprops), 0);

plist→pl_head = NULL; /* will overwrite this */
atomic_init(&(plist→log_pl_len), 0);
atomic_init(&(plist→phys_pl_len), 0);

atomic_init(&(plist→class_init), FALSE);

atomic_init(&(plist→thrd), init_thrd);
atomic_init(&(plist→fl_next), init_fl_next);
```

  Also initialize the lock free singly linked list rooted in plist→pl_head and its associated counters. As currently conceived, this list will have sentinel entries at the beginning and end of the list – which is why plist→pl_head is not atomic. Do this now, as it may be necessary to insert entries into the list during the initialization of the lookup table.

  Observe that by setting the plist→thrd_init.opening flag to TRUE, we prevent access to the property list until its initialization is complete.

- Scan the target property list class (plist→pclass) to determine the number of properties that are valid in property list class version plist→pclass_version[30], and store this value in plist→nprops_inherited.

  In passing, increment the atomic ref_count field of each such property in the target property list class[31].

[30] Recall that a property prop in a given property list class is valid for version v of that property list class if prop→create_version <= v, prop→delete_version == 0 or prop→delete_version > v, and the property list contains no property of name == prop→name with create_version greater than prop→create_version and less than or equal to v.

[31] Need a method to ensure that properties are not deleted out from under property lists while they are being initialized. Until that is done, don't prune property list class properties.

- Allocate an array of H5P_mt_list_table_entry_t of length plist→nprops_inherited, and store its base address in the local variable lkup_tbl. We will set plist→lkup_tbl to lkup_table once it is fully initialized. For all I, 0 <= i < plist→nprops_inherited, initialize lkup_tbl[i] as follows:

```
H5P_mt_list_prop_ref_t init_prop_ref = {NULL, 0};

lkup_tbl[i].chksum = 0;
lkup_tbl[i].name = NULL;
atomic_init(&(lkup_tbl[i].base), init_prop_ref);
atomic_init(&(lkup_tbl[i].base_delete_version), 0);
atomic_init(&(lkup_tbl[i].curr), init_prop_ref);
```

- Scan the target property list class (plist→pclass) again. For each property plc_prop in the list that is valid for property list class version plist→pclass_version, select a unique i, 0 <= i < plist→nprops_inherited, and update lkup_tbl[i] as follows:

```
H5P_mt_list_prop_ref_t prop_ref;

lkup_tbl[i].chksum = plc_prop→chksum;
lkup_tbl[i].name = strdup(plc_prop→name);
prop_ref.ptr = plc_prop;
prop_ref.ver = 1;
atomic_store(&(lkup_tbl[i].base), prop_ref);
```

- Sort lkup_tbl first by chksum, and then by name.

- For each i, 0 <= i < plist→nprops_inherited, examine lkup_tbl[i].base.ptr→create.

  If it is NULL, we are done, and can go on to the next value of i.

- If it is not, the create callback is defined, and we must copy the property into the new property list, instead of using the version in the property list class as our initial value of the property. To do this we must:

  - Allocate a new instance of H5P_mt_prop_t, store a pointer to it in prop, and initialize *prop as follows:

```
H5P_mt_list_prop_ref_t plc_prop;
H5P_mt_prop_aptr_t init_prop_aptr = {NULL, 0};
H5P_mt_prop_value_t init_prop_value = {NULL, 0};

atomic_load(&(lkup_tbl[i].base), plc_prop);

prop→tag = H5P_MT_PROP_TAG;

atomic_init(&(prop→next), init_prop_aptr);
prop→sentinel        = FALSE;
prop→in_prop_class    = FALSE;
atomic_init(&(prop→ref_count), 0);

prop→chksum          = plc_prop.ptr→chksum;
```

```
prop→name           = strdup(plc_prop.ptr→name);

atomic_init(&(prop→value), init_prop_value);
atomic_init(&(prop→create_version), 1);
atomic_init(&(prop→delete_version), 0);

prop→callbacks_mt_safe = plc_prop.ptr→callbacks_mt_safe;

prop→create         = plc_prop.ptr→create;
prop→set            = plc_prop.ptr→set;
prop→get            = plc_prop.ptr→get;
prop→encode         = plc_prop.ptr→encode;
prop→decode         = plc_prop.ptr→decode;
prop→del            = plc_prop.ptr→del;
prop→copy           = plc_prop.ptr→copy;
prop→cmp            = plc_prop.ptr→cmp;
prop→close          = plc_prop.ptr→close;
```

  - Allocate a buffer of size strdup(lkup_tbl[i].base.ptr→value.size and store its address in the local variable value.

  - memcopy lkup_tbl[i].base.ptr→value.ptr into the buffer pointed to by value.

  - Call (prop→create)(prop→name, lkup_tbl[i].base.ptr→value.size, value)

  - Set prop→value to contain a pointer to and size of the local variable value as follows:

```
H5P_mt_list_prop_ref_t plc_prop;
H5P_mt_prop_value_t plc_prop_value;
H5P_mt_prop_value_t prop_value;

atomic_load(&(lkup_tbl[i].base), plc_prop);
atomic_load(&(plc_prop.ptr→value), plc_prop_value);
prop_value.ptr = value;
prop_value.size = plc_prop_value.size;
atomic_store(&(prop→value), prop_value);
```

  - Insert prop into the lock free singly linked list of properties pointed to by prop→pl_head.

  - Since we had to copy prop from the source property list class instead of using it as the default value of the property, decrement the reference count on lkup_tbl[i].base.ptr.

  - Finally, set lkup_tbl[i].base to NULL, and set lkup_tbl[i].curr to point to prop. Do this as follows:

```
H5P_mt_list_prop_ref_t prop_ref = {NULL, 0};

atomic_store(&(lkup_tbl[i].base), prop_ref);
prop_ref.ptr = prop;
prop_ref.ver = 1;
atomic_store(&(lkup_tbl[i].curr), prop_ref);
```

- Once lkup_tbl is initialized, set plist→lkup_tbl = lkup_tbl.

At this point, we have done as much initialization a we can before inserting the property list into the property list index. Note that once *plist is in the index, it is accessible to other threads. We use the plist→thrd_init.opening to prevent access until the property list is fully initialized.

- Call plist_id = H5I_register(H5I_GENPROP_LST, plist, TRUE) to insert plist into the property list index, and then call atomic_store(&(plist→plist_id), plist_id) to store the new ID in *plist.

- For all parent property list classes, starting with plist→pclass_ptr and the working up the inheritance tree, call that property list class's create_func if it exists. If any of these calls returns failure, report failure. Otherwise set plist→class_init to TRUE via a call to atomic_store().

- Finally, reset plist→thrd.opening as follows:

```
H5P_mt_active_thread_count_t thrd    = {0, FALSE, FALSE};

atomic_store(&(plist→thrd), thrd);
```

In the event of any failure in the above, clean up the partially created property list (if any), and return failure.

H5Pcreate_class()

Signature:

```
H5_DLL hid_t H5Pcreate_class(hid_t parent, const char *name,
                             H5P_cls_create_func_t create,
                             void *create_data, H5P_cls_copy_func_t copy,
                             void *copy_data, H5P_cls_close_func_t close,
                             void *close_data);
```

Description:

Create a new property list class based on the supplied parent property list class, insert it in the index, and return its id.

Outline of Processing:

- Verify that the supplied name is defined, and fail if it is not.

- If the parent equals H5P_DEFAULT, set the parent_class local variable to NULL.[32]

[32] We may need to disallow H5P_DEFAULT as the parent class. See comment

Otherwise, verify that the supplied parent is in fact associated with a property list class, and obtain a pointer to the associated instance of H5P_mt_class_t and store it in the local variable parent_pclass.

- If parent_pclass is not NULL, attempt to increment parent_pclass→ref_count.plc.[33] To do this, proceed as follows:

  1. Atomically load parent_pclass→ref_count into the local variable parent_ref_count.

  2. If parent_ref_count.pl == 0, parent_ref_count.plc == 0, and parent_ref_count.deleted == TRUE, return and report failure.

  3. Increment parent_ref_count.plc.

  4. Attempt to overwrite parent_pclass→ref_count with parent_ref_count using an atomic compare exchange strong. On failure, return to 1. above. Otherwise proceed.

- Atomically load parent_pclass→version into the local variable target_version.

- Allocate an instance of H5P_mt_class_t and store its pointer in the local variable child_pclass. Then initialize it as follows:

```
H5P_mt_active_thread_count_t init_thrd    = {0, TRUE, FALSE};
H5P_mt_class_ref_counts_t init_ref_count  = {0, 0, FALSE};
H5P_mt_class_sptr_t init_fl_next          = {NULL, 0};

child_pclass→tag                = H5P_MT_CLASS_TAG;
child_pclass→parent_id          = parent;
child_pclass→parent_ptr         = parent_pclass;
child_pclass→parent_version     = target_version;
child_pclass→name               = strdup(name);
atomic_init(&(child_pclass->id), 0);  /* will overwrite this */
child_pclass→type               = H5P_TYPE_USER;

atomic_init(&(child_pclass→curr_version), 1);
atomic_init(&(child_pclass→next_version), 2);

/* List of properties, and related fields */
child_pclass→pl_head            = NULL; /* will overwrite this */
atomic_init(&(child_pclass→log_pl_len), 0);
atomic_init(&(child_pclass→phys_pl_len), 0);

/* reference counts */
atomic_init(&(child_pclass→ref_count), init_ref_count);

/* Callback function pointers & info */
child_pclass→create_func        = create;
child_pclass→create_data        = create_data;
child_pclass→copy_func          = copy;
child_pclass→copy_data          = copy_data;
```

[33] Do we need to increment parent_pclass→thrd.count as well?

```
child_pclass→close_func         = close;
child_pclass→close_data         = close_data;

/* shutdown and free list management fields */
atomic_init(&(child_pclass→thrd), init_thrd);
atomic_init(&(child_pclass→fl_next), init_fl_next);
```

Note that the type of the new property list class is hard coded to H5P_TYPE_USER in the public version of this call. In the internal version, the property list class type is passed in as a parameter to the create property list class function.

- Initialize the lock free singly linked list rooted in child_pclass→pl_head and its associated counters. As currently conceived, this list will have sentinel entries at the beginning and end of the list – which is why child_pclass→pl_head is not atomic.

- Initialize the local variable nprops to zero, and scan the parent property list class (parent_pclass) to look for properties that are valid in parent property list class version new_pclass→parent_version. For each such property found, set the local variable old_prop equal to the address of the associated instance of H5P_mt_prop_t, and proceed as follows:

    - Increment nprops.

    - Verify that old_prop→in_prop_class is true.

    - Atomically increment old_prop→ref_count.

    - Copy *old_prop into the lock free linked list rooted at new_pclass→pl_head. To do this we must:

        - Allocate a new instance of H5P_mt_prop_t, store a pointer to it in the new_prop local variable, and initialize *new_prop as follows:

```
H5P_mt_prop_aptr_t init_prop_aptr = {NULL, 0};
H5P_mt_prop_value_t prop_value = {NULL, 0};

new_prop→tag = H5P_MT_PROP_TAG;

atomic_init(&(new_prop→next), init_prop_aptr);
new_prop→sentinel         = FALSE;
new_prop→in_prop_class    = TRUE;
atomic_init(&(new_prop→ref_count), 0);

new_prop→chksum           = old_prop→chksum;
new_prop→name             = strdup(old_prop→name);

atomic_init(&(new_prop→value), prop_value);
atomic_init(&(new_prop→create_version), 1);
atomic_init(&(new_prop→delete_version), 0);

new_prop→callbacks_mt_safe = old_prop→callbacks_mt_safe;
```

```
new_prop→create        = plc_prop.ptr→create;
new_prop→set           = plc_prop.ptr→set;
new_prop→get           = plc_prop.ptr→get;
new_prop→encode        = plc_prop.ptr→encode;
new_prop→decode        = plc_prop.ptr→decode;
new_prop→del           = plc_prop.ptr→del;
new_prop→copy          = plc_prop.ptr→copy;
new_prop→cmp           = plc_prop.ptr→cmp;
new_prop→close         = plc_prop.ptr→close;
```

- Allocate a buffer of size old_prop→value.size, and store its address in the local variable new_val. Then memcpy old_prop→value.ptr into *new_val.

  If new_prop→copy is not NULL, run the copy callback on new_val :

  ```
  (new_prop->copy)(new_prop->name, old_prop→value.size, new_val)
  ```

  Finally, set prop_value.ptr = new_val, prop_value.size = old_prop→value.size, and then atomically set new_prop→value to prop_val.

- Insert new_prop into the lock free singly linked list rooted at child_pclass→pl_head, and update child_pclass→log_pl_len and child_pclass→phys_pl_len accordingly.

- Verify that child_pclass→log_pl_len == nprops.

- Insert child_pclass into the index by calling

  ```
  H5I_register(H5I_GENPROP_CLS, child_pclass, TRUE)
  ```

  Make note of the returned id, and atomically set child_pclass→id to this value

- Reset the opening flag in child_pclass→thrd. To do this, proceed as follows:

  1. Atomically load child_pclass→thrd into the local variable local_thrd.

  2. Verify that local_thrd.threads == 0, and that local_thrd.opening == TRUE. Note that we don't check the closing flag, as it is possible that we are completing the initialization of *child_pclass just as the associated index is being shut down.

  3. Set local_thrd.opening = FALSE.

  4. Attempt to overwrite child_pclass→thrd with local_thrd using a compare exchange strong. On failure, return to 1. and try again.

- Return child_pclass→id.

H5Pdecode()

Signature:

```
H5_DLL hid_t H5Pdecode(const void *buf);
```

Description:

Given a buffer containing an encoded property list, decode the buffer, construct the property list described in the buffer, insert it in the index, and return the id associated with the decoded property list.

Outline of Processing:

H5Pencode()

Signature:

```
H5_DLL herr_t H5Pencode2(hid_t plist_id, void *buf, size_t *nalloc,
                         hid_t fapl_id);
```

Description:

Encode the indicated property list in the supplied buffer, and return the number of bytes written to buf in *nalloc. If buf is NULL, *nalloc is set to the number of bytes required in buf.

Outline of Processing:

H5Pequal()

Signature:

```
H5_DLL htri_t H5Pequal(hid_t id1, hid_t id2);
```

Description:

Compare two property list or two property list classes and return TRUE if their current versions are identical, and FALSE otherwise.

Outline of Processing:

H5Pexist()

Signature:

```
H5_DLL htri_t H5Pexist(hid_t plist_id, const char *name);
```

Description:

Search for a property of the specified name in the current versions of the property list or property list class associated with the supplied ID. Return TRUE if such a property exists, and FALSE otherwise.

Outline of Processing:

H5Pget()

Signature:

```
H5_DLL herr_t H5Pget(hid_t plist_id, const char *name, void *value);
```

Description:

Get a copy of the value of an existing property in a property list.

Outline of Processing:

Look up the current version of the indicated property in the target property and return a copy of its value in *value. Do this as follows:

- If the supplied plist_id is not associated with a property list, flag an error and return.
- If the supplied name is the empty string, flag an error and return.
- If the supplied value pointer is NULL, flag an error and return.
- Obtain a pointer to the instance of H5P_mt_list_t. Do this as follows, with appropriate error checking added:

```
H5P_mt_list_t plist;

plist = (H5P_genplist_t *)H5I_object_verify(plist_id, H5I_GENPROP_LST);
```

- Increment the active thread count in the target property list so that it can't be deleted out from under us. In passing we check for the case in which the property list is in the process of being created or discarded, and fail in either event.

  To do this, atomically load plist→thrd in to a local instance of H5P_mt_active_thread_count_t – call this variable local_thrd. If either thrd.opening or thrd.closing are TRUE, flag an error and return. Otherwise, increment local_thrd.count and attempt to overwrite plist→thrd with a compare exchange strong. Repeat this process until failure or the compare exchange strong is successful.

- To avoid the possibility of the target property list being modified out from under us, we must first obtain the version from which the read will take place. To do this, define the local variable target_version, and set its value as follows:

  ```
  target_version = atomic_load(&(plist->curr_version));
  ```

- Next, compute the checksum of the supplied name as follows:

  ```
  int64_t chksum;

  chksum = (int64_t)H5_checksum_fletcher32((void *)name, strlen(name));
  ```

  Note the type cast on the return value of H5_checksum_fletcher32(). That function returns a 32 bit value, while we use a 64 bit value for the IDs checksums of property names[34]. Recall that this is for the convenience of our lock free singly linked list, as it allows us to set values for positive and negative infinity that will never be computed as the checksum of a property name.

- Attempt to obtain a pointer to the target property – call this pointer prop. Typically, this search is expedited by the lookup table as follows:

  - Start with a binary search on plist→lkup_tbl[35] for an i such that plist→lkup_tbl[i].chksum == chsksum. If such an i exists, test to see if plist→lkup_tbl[i].name is identical to name.

  - If it is, the search is successful.

  - If it isn't, it is possible that two or more names have hashed to the same value. To test this, look to either side of it to identify any entries in the lookup table with a chksum field equal to chksum. If any such entries exist, compare the associated name fields to see if any match name. If one does, set it equal to its index, and report success. Otherwise fail.

---

34 Recall that this is for the convenience of our lock free singly linked list, as it allows us to set values for positive and negative infinity that will never be computed as the checksum of a property name.

35 Recall that the lookup table is sorted first by chksum, and then by name.

If the search of the lookup table is successful, examine plist→lkup_tbl[i] to try to find prop. This can happen in one of several ways:

- If plist→lkup_tbl[i].base.ptr is not NULL, and plist→lkup_tbl[i].curr.ptr is NULL, the only version of the target property is the default inherited from the source property list class. If plist→lkup_tbl[i].base_delete_version is either zero or greater than target_version, set prop equal to list→lkup_tbl[i].base.ptr. Otherwise, report failure.

- If plist→lkup_tbl[i].base.ptr is NULL, plist→lkup_tbl[i].curr.ptr must be non NULL. Set search_ptr equal to plist→lkup_tbl[i].curr.ptr. Observe that *search_ptr is an entry in the lock free singly linked list whose head is pointed to by plist→pl_head. Observe also that if the target instance of H5P_mt_prop_t exists for the target_version, it must reside in this list since plist→lkup_tbl[i].base.ptr is NULL, and thus the default version of the property was copied into plist.

  Search for the target property as follows:

  1. If search_ptr is NULL, or search_ptr→chksum != chksum, or search_ptr→name is not equal to name, report failure.

  2. If search_ptr→creation_version is greater than target_version, set search_ptr = search_ptr→next.ptr, and goto 1 above.

  3. If search_ptr→delete version is non-zero and less than search_ptr, report failure.

  4. Otherwise, set prop equal to search_ptr.

- The most complex case occurs when both plist→lkup_tbl[i].base.ptr and plist→lkup_tbl[i].curr.ptr are not NULL. In this case, the target property exists in the target_version, it may be resident in either in the lock free singly linked list pointed to by plist→pl_head, or that pointed to by plist→pclass→pl_head. In this later case, the target property (if it exists) must be pointed to by plist→lkup_tbl[i].base.ptr.

  Search for the target property as follows:

  1. If search_ptr is NULL, or search_ptr→chksum != chksum, or search_ptr→name is not equal to name, goto 4.

  2. If search_ptr→creation_version is greater than target_version, set search_ptr = search_ptr→next.ptr, and goto 1 above.

3. If search_ptr→delete version is non-zero and less than search_ptr, report failure. Otherwise, set prop equal to search_ptr.

4. If plist→lkup_tbl[i].base.ver is less than or equal to target_version, and plist→lkup_tbl[i].base_delete_version is zero or greater than target_version, set prop equal to plist→lkup_tbl[i].base.ptr. Otherwise, report failure.

◦ If both plist→lkup_tbl[i].base.ptr and plist→lkup_tbl[i].curr.ptr are NULL, throw an assertion failure.

On the other hand, if the search of the lookup table fails, either the target property doesn't exist, or it was added to the property list, and thus doesn't appear in the lookup table. To handle this case, set search_ptr equal to plist→pl_head, and proceed as follows:

1. If search_ptr is NULL, report failure.

2. If search_ptr→chksum is less than chksum, set search_ptr = search_ptr→next.ptr.

3. If search_ptr→chksum == chksum, and strcmp(search_ptr→name, name) is less than zero, set search_ptr = search_ptr→next.ptr.

4. If search_ptr→chksum == chksum, strcmp(search_ptr→name, name) is zero, and search_ptr→create_version is greater than target_version, set search_ptr = search_ptr→next.ptr.

5. If search_ptr→chksum == chksum, strcmp(search_ptr→name, name) is zero, and search_ptr→create_version is less than or equal to target_version, examine search_ptr→delete_version. If it is zero, or greater than the target version, set prop = search_ptr. Otherwise, report failure.

6. If search_ptr→chksum is greater than chksum, or strcmp(search_ptr→name, name) is greater than zero, report failure.

- Having found prop – the target instance of H5P_mt_prop_t, verify that prop→value.size is positive. Return failure if this is false.[36]

- If prop→get is NULL, memcpy() prop→value.ptr into the buffer pointed to by the value parameter. If prop→get is defined, allocate a buffer tmp of size prop→value.size, memcpy prop→value.ptr into tmp, run

[36] As mentioned in the discussion of H5Pget() in appendix 1, this disagrees with the documentation. Follow the existing code for now, but we must untangle this issue.

(*(prop->get))(plist->plist_id, name, prop→value.size, tmp)

then memcpy tmp into *value and discard tmp. The reason for this circumlocution is not clear, but best to duplicate the current code for now, and optimize later.

- Decrement the active thread count in the target property list.

  To do this, atomically load plist→thrd into a local instance of H5P_mt_active_thread_count_t – call this variable local_thrd. Decrement local_thrd.count and attempt to overwrite plist→thread with a compare exchange strong. Repeat this process until the compare exchange strong is successful.

H5Pget_class()

Signature:

```
H5_DLL hid_t H5Pget_class(hid_t plist_id);
```

Description:

Return an id that maps to the property list class from which the supplied property list was derived.

In the current implementation of this function, the ID returned maps to the version of the property list class from which the property list was derived.

While the rewrite of property lists would permit this, it introduces much potential for confusion as discussed earlier in this document. Instead return the ID of the parent property list class, which may have been modified since the target property list was created.

Outline of Processing:

H5Pget_class_name()

Signature:

```
H5_DLL char *H5Pget_class_name(hid_t pclass_id);
```

Description:

Look up the indicated property list class, allocate a string of appropriate length, copy the class's name into the new string, and return a pointer to it.

Outline of Processing:

H5Pget_class_parent()

Signature:

```
H5_DLL hid_t H5Pget_class_parent(hid_t pclass_id);
```

Description:

Return an id that maps to the property list class from which the supplied property list class was derived.

In the current implementation of this function, the ID returned maps to the version of the property list class from which the property list class was derived.

While the rewrite of property lists would permit this, it introduces much potential for confusion as discussed earlier in this document. Instead return the ID of the parent property list class, which may have been modified since the target property list was created.

Outline of Processing:

H5Pget_nprops()

Signature:

```
H5_DLL herr_t H5Pget_nprops(hid_t id, size_t *nprops);
```

Description:

Given the id of a property list or a property list class, return the number of properties in the same. In the case of property list classes, this is just the number of properties defined in the target property list class, and does not include properties defined in its parents.[37]

Outline of Processing:

[37] Is this sensible in the new implementation? Think on this.

H5Pget_size()

Signature:

```
H5_DLL herr_t H5Pget_size(hid_t id, const char *name, size_t *size);
```

Description:

Look up the named property in the current version of the property list or property list class associated with the supplied id, and return the size of the value of the property in *size.

Outline of Processing:

H5Pinsert2()

Signature:

```
H5_DLL herr_t H5Pinsert2(hid_t plist_id, const char *name, size_t size,
                         void *value, H5P_prp_set_func_t prp_set,
                         H5P_prp_get_func_t prp_get,
                         H5P_prp_delete_func_t prp_del,
                         H5P_prp_copy_func_t prp_copy,
                         H5P_prp_compare_func_t prp_cmp,
                         H5P_prp_close_func_t prp_close);
```

Description:

Create a property as specified by the supplied parameters and insert it in the target property list.

Outline of Processing:

Start with basic sanity checking:

- If the supplied hid_t is not associated with a property list, flag an error and return.
- If the supplied name is the empty string, flag an error and return.
- Verify that either the supplied size and value are 0 and NULL or positive and not NULL. Flag and error and return if this is not the case.
- Lookup the supplied name in the current version of the target property list, and flag an error if it contains a property of the supplied name.
- Obtain a pointer to the instance of H5P_mt_list_t. Do this as follows, with appropriate error checking added:

```
H5P_mt_list_t plist;

plist = (H5P_genplist_t *)H5I_object_verify(plist_id, H5I_GENPROP_LST);
```

- Increment the active thread count in the target property list so that it can't be deleted out from under us. In passing we check for the case in which the property list is in the process of being created or discarded, and fail in either event.

  To do this, atomically load plist→thrd into a local instance of H5P_mt_active_thread_count_t – call this variable local_thrd. If either thrd.opening or thrd.closing are TRUE, flag an error and return. Otherwise, increment local_thrd.count and attempt to overwrite plist→thread with a compare exchange strong. Repeat this process until failure or the compare exchange strong is successful.

- To modify the target property list, we must first obtain the version in which the modification will take place. To do this, define the local variable target_version, and set its value as follows:

  ```
  target_version = atomic_fetch_add(plist->next_version);
  ```

- As mentioned earlier, multiple modifications to a given property must be executed in target_version order, as must increments to plist→version. As simultaneous modifications to a give property list by multiple threads is expected to be rare, just execute all modifications to a given property list in target_version issue order. To do this, atomically load plist→version. If it is one greater than target_version, proceed with the modification. Otherwise, sleep a little and try again.

  Note that we must maintain statistics on both branches to see if the above expectation is correct. If it isn't, we must come up with something better method of enforcing the required serialization

- Allocate a new instance of H5P_mt_prop_t and store its address in the local variable prop. Then initialize *prop as follows:

  ```
  uint32_t chksum;
  H5P_mt_prop_aptr_t init_prop_aptr = {NULL, 0};
  H5P_mt_prop_value_t prop_value = {NULL, 0};

  prop→tag = H5P_MT_PROP_TAG;

  atomic_init(&(prop→next), init_prop_aptr);
  prop→sentinel        = FALSE;
  prop→in_prop_class    = FALSE;
  atomic_init(&(prop→ref_count), 0);

  chksum              = H5_checksum_fletcher32((void *)name, strlen(name));
  prop→chksum          = (int64_t)chksum;
  prop→name           = strdup(plc_prop.ptr→name);

  if ( NULL != value ) {
  ```

```
        prop_value.ptr = malloc(size);
        prop_value.size = size;
        memcpy(prop_value.ptr, size);
        atomic_init(&(prop→value), prop_value);
    }
    atomic_init(&(prop→value), prop_value);
    atomic_init(&(prop→create_version), target_version);
    atomic_init(&(prop→delete_version), 0);

    prop→callbacks_mt_safe = FALSE;

    prop→create        = NULL;
    prop→set           = prp_set;
    prop→get           = prp_get;
    prop→encode        = NULL;
    prop→decode        = NULL;
    prop→del           = prp_del;
    prop→copy          = prp_copy;
    if ( NULL == prp_cmp ) {
        prop_cmp       = &memcmp;
    } else {
        prop→cmp       = prp_cmp;
    }
    prop→close         = prp_close;
```

Callbacks not defined in the H5Pinsert2() API are set to NULL. The callbacks_mt_safe field is initialized to FALSE, since the current API doesn't allow specification of this field. Needless to say, we need to either add an API to do this, or modify the current API.

- Attempt to insert prop into the lock free singly linked list pointed to by plist→pl_head. Fail it plist→pl_head contains an entry of name prop→name if it has not been deleted prior to the target version. Atomically increment the logical and physical length fields in passing

- Atomically increment both plist→nprops and plist→nprops_added.

- Atomically increment plist→version. This makes the new property visible, and completes the insertion process.

- Decrement the active thread count in the target property list.

  To do this, atomically load plist→thrd in to a local instance of H5P_mt_active_thread_count_t – call this variable local_thrd. Decrement local_thrd.count and attempt to overwrite plist→thread with a compare exchange strong. Repeat this process until the compare exchange strong is successful.

H5Pisa_class()

Signature:

```
H5_DLL htri_t H5Pisa_class(hid_t plist_id, hid_t pclass_id);
```

Description:

Determine whether the the supplied property list is a member of the supplied property list class.[38]

Outline of Processing

H5Piterate()

Signature:

```
int H5Piterate(hid_t id, int *idx, H5P_iterate_t iter_func, void *iter_data);
```

Description:

Starting with the *idx'th property, scan the target property list or property list class and call the supplied iter_fcn with the id of the property list or property list class, the name of the property, and the supplied user data.

Outline of Processing

H5Pregister2()

Signature:

```
H5_DLL herr_t H5Pregister2(hid_t cls_id, const char *name, size_t size,
                    void *def_value, H5P_prp_create_func_t create,
                    H5P_prp_set_func_t set, H5P_prp_get_func_t get,
                    H5P_prp_delete_func_t prp_del,
                    H5P_prp_copy_func_t copy,
                    H5P_prp_compare_func_t compare,
                    H5P_prp_close_func_t close);
```

Description:

Insert a new property in a property list class.

If the target class has any directly derived property lists or property list classes, this will result in duplication of the target property list, with the duplicate (with the new property added) replacing the old version in the index.

---

[38] The current semantics of this function are odd – to quote the discussion of same in Appendix 1:

> Here "member" appears to mean that plist_id refers to a property list that is an instance of the property list class referred to by pclass_id **OR** that the immediate parent property list class of the supplied property list is a descendant of the supplied property list class. While this is standard OOP terminology, it may be asking a bit much of our users to know this. Assuming that this interpretation is retained, this point should be made clear in the user level documentation.

We need to decide whether to retain this.

Outline of Processing

H5Premove()

Signature:

```
H5_DLL herr_t H5Premove(hid_t plist_id, const char *name);
```

Description:

Remove the property of the supplied name from the indicated property list.

Outline of Processing

- If the supplied hid_t is not associated with a property list, flag an error and return.

- If the supplied name is the empty string, flag an error and return.

- Lookup the supplied name in the current version of the target property list, and flag an error if it doesn't exist.

- Obtain a pointer to the instance of H5P_mt_list_t. Do this as follows, with appropriate error checking added:

```
H5P_mt_list_t plist;

plist = (H5P_genplist_t *)H5I_object_verify(plist_id, H5I_GENPROP_LST);
```

- Increment the active thread count in the target property list so that it can't be deleted out from under us. In passing we check for the case in which the property list is in the process of being created or discarded, and fail in either event.

  To do this, atomically load plist→thrd in to a local instance of H5P_mt_active_thread_count_t – call this variable local_thrd. If either thrd.opening or thrd.closing are TRUE, flag an error and return. Otherwise, increment local_thrd.count and attempt to overwrite plist→thrd with a compare exchange strong. Repeat this process until failure or the compare exchange strong is successful.

- To modify the target property list, we must first obtain the version in which the modification will take place. To do this, define the local variable target_version, and set its value as follows:

```
target_version = atomic_fetch_add(plist->next_version);
```

- As mentioned earlier, multiple modifications to a given property must be executed in target_version order, as must increments to plist→version. As simultaneous modifications to a given property list by multiple threads is expected to be rare, just execute all modifications to a given property list in target_version issue order. To do this, atomically load plist→version. If it is one greater than target_version, proceed with the modification. Otherwise, sleep a little and try again.

  Note that we must maintain statistics on both branches to see if the above expectation is correct. If it isn't, we must come up with something better method of enforcing the required serialization

- Lookup the target property for the current version of the property list (i.e. plist→version). Fail if it doesn't exist. Otherwise set prop to point to it.

- Verify that prop→delete_version is zero, and atomically set prop→delete_version to target_version.

- Atomically decrement plist→nprops.

- Atomically increment plist→version. This makes the property deletion visible, and completes the deletion process.

- Decrement the active thread count in the target property list.

  To do this, atomically load plist→thrd in to a local instance of H5P_mt_active_thread_count_t – call this variable local_thrd. Decrement local_thrd.count and attempt to overwrite plist→thread with a compare exchange strong. Repeat this process until the compare exchange strong is successful.

H5Pset()

Signature:

    H5_DLL herr_t H5Pset(hid_t plist_id, const char *name, const void *value);

Description:

    Set the value of an existing property in a property list.

Outline of Processing:

Insert a new copy of the indicated property with the specified value in the target property list with a creation version one greater than the current version of the target property list. Do this as follows:

- If the supplied hid_t is not associated with a property list, flag an error and return.

- If the supplied name is the empty string, flag an error and return.

- If the supplied value pointer is NULL, flag an error and return.

- Lookup the supplied name in the current version of the target property list, and flag an error if it doesn't exist.

- Obtain a pointer to the instance of H5P_mt_list_t. Do this as follows, with appropriate error checking added:

```
H5P_mt_list_t plist;

plist = (H5P_genplist_t *)H5I_object_verify(plist_id, H5I_GENPROP_LST);
```

- Increment the active thread count in the target property list so that it can't be deleted out from under us. In passing we check for the case in which the property list is in the process of being created or discarded, and fail in either event.

  To do this, atomically load plist→thrd in to a local instance of H5P_mt_active_thread_count_t – call this variable local_thrd. If either thrd.opening or thrd.closing are TRUE, flag an error and return. Otherwise, increment local_thrd.count and attempt to overwrite plist→thrd with a compare exchange strong. Repeat this process until failure or the compare exchange strong is successful.

- To modify the target property list, we must first obtain the version in which the modification will take place. To do this, define the local variable target_version, and set its value as follows:

```
target_version = atomic_fetch_add(plist->next_version);
```

- As mentioned earlier, multiple modifications to a given property must be executed in target_version order, as must increments to plist→version. As simultaneous modifications to a given property list by multiple threads is expected to be rare, just execute all modifications to a given property list in target_version issue order. To do this, atomically load plist→version. If it is one greater than target_version, proceed with the modification. Otherwise, sleep a little and try again.

  Note that we must maintain statistics on both branches to see if the above expectation is correct. If it isn't, we must come up with something better method of enforcing the required serialization.

- Lookup the target property for the current version of the property list (i.e. plist→version). Fail if it doesn't exist. Otherwise set prop to point to it.

- Allocate a new instance of H5P_mt_prop_t, and store its address in the local variable new_prop. Then initialize *new_prop as follows:

```
H5P_mt_prop_aptr_t init_prop_aptr = {NULL, 0};
H5P_mt_prop_value_t new_prop_value = {NULL, 0};
H5P_mt_prop_value_t old_prop_value = {NULL, 0};

new_prop→tag        = H5P_MT_PROP_TAG;

atomic_init(&(new_prop→next), init_prop_aptr);
new_prop→sentinel        = FALSE;
new_prop→in_prop_class    = FALSE;
atomic_init(&(new_prop→ref_count), 0);

new_prop→chksum          = prop->chksum;
new_prop→name           = strdup(prop.ptr→name);

atomic_load(&(prop→value), old_prop_value);

if ( NULL != prop→set ) {

   new_prop_value.ptr  = malloc(old_prop_value.size);
   new_prop_value.size = old_prop_value.size;

   memcpy(new_prop_value.ptr, value,  new_prop_value.size);

   (*(prop->set))(atomic_load(&(plist->plist_id), name,
                      new_prop_value.size, new_prop_value.ptr);

} else {
   /* This is odd, since it seems that *value is included in the
    * property if set is NULL, but copied if set isn't NULL.  This
    * seems to invite memory leaks.  That said, this seems to be what
    * the current code is doing.  Treat this with suspicion.
    */
   new_prop_value.ptr  = value;
   new_prop_value.size = old_prop_value.size;
}

atomic_init(&(new_prop→value), new_prop_value);
atomic_init(&(new_prop→create_version), target_version);
atomic_init(&(new_prop→delete_version), 0);

new_prop→callbacks_mt_safe = prop→callbacks_mt_safe;

new_prop→create          = prop→create;
new_prop→set            = prop→set;
new_prop→get            = prop→get;
new_prop→encode          = prop→encode;
new_prop→decode          = prop→decode;
new_prop→del            = prop→del;
new_prop→copy           = prop→copy;
new_prop→cmp            = prop→cmp;
new_prop→close          = prop→close;
```

- Insert new_prop into the lock free singly linked list of properties pointed to by prop→pl_head. In passing, atomically increment the log_pl_len and phys_pl_len fields.

- Test to see if name appears in the lookup table. If it is, let i be its index, and update plist→lkup_tbl[i].curr as follows:

```
H5P_mt_list_prop_ref_t new_curr;

new_curr.ptr = new_prop;
new_curr.ver = target_version;

atomic_store(&(plist→lkup_tbl[i].curr), new_curr);
```

- Atomically increment plist→version. This makes the new property value visible, and completes the set process.

- Decrement the active thread count in the target property list.

  To do this, atomically load plist→thrd in to a local instance of H5P_mt_active_thread_count_t – call this variable local_thrd. Decrement local_thrd.count and attempt to overwrite plist→thread with a compare exchange strong. Repeat this process until the compare exchange strong is successful.

H5Punregister()

Signature:

```
H5_DLL herr_t H5Punregister(hid_t pclass_id, const char *name);
```

Description:

Delete a property from the specified property list class.

According to the documentation, this should have no effect on existing property lists. While we will ensure this in the new implementation, examination of the code suggests that this is not what the existing code does.

Outline of Processing:

Existing Private API Function Outlines

In this section, list the existing private API, and outline their processing where the above discussion of public API function does not make this redundant.

Note that we only consider the relatively low level private API calls – there are a large number of private H5P calls built on top of these for individual properties in individual property lists. If we implement the necessary low level routines correctly, these should just work.

H5P_class_isa()

Signature:

```
H5_DLL htri_t H5P_class_isa(const H5P_genclass_t *pclass1,
                    const H5P_genclass_t *pclass2);
```

Description:

Outline of Processing:

H5P_close()

Signature:

```
H5_DLL herr_t H5P_close(H5P_genplist_t *plist);
```

Description:

Outline of Processing:

H5P_copy_plist()

Signature:

```
H5_DLL hid_t  H5P_create_id(H5P_genclass_t *pclass, hbool_t app_ref);
```

Description:

Outline of Processing:

H5P_create_id()

Signature:

```
H5_DLL hid_t  H5P_create_id(H5P_genclass_t *pclass, hbool_t app_ref);
```

Description:

Outline of Processing:

H5P_exist_plist()

Signature:

```
H5_DLL htri_t H5P_exist_plist(const H5P_genplist_t *plist, const char *name);
```

Description:

Outline of Processing:

H5P_get()

Signature:

```
H5_DLL herr_t H5P_get(H5P_genplist_t *plist, const char *name, void *value);
```

Description:

Outline of Processing:

H5P_get_class_name()

Signature:

```
H5_DLL char * H5P_get_class_name(H5P_genclass_t *pclass);
```

Description:

Outline of Processing:

H5P_get_nprops_pclass()

Signature:

```
H5_DLL herr_t H5P_get_nprops_pclass(const H5P_genclass_t *pclass,
                        size_t *nprops, hbool_t recurse);
```

Description:

Outline of Processing:

H5P_insert()

Signature:

```
H5_DLL herr_t H5P_insert(H5P_genplist_t *plist,
                  const char *name,
                  size_t size,
                  void *value,
                  H5P_prp_set_func_t prp_set,
                  H5P_prp_get_func_t prp_get,
                  H5P_prp_encode_func_t prp_encode,
                  H5P_prp_decode_func_t prp_decode,
                  H5P_prp_delete_func_t prp_delete,
                  H5P_prp_copy_func_t prp_copy,
                  H5P_prp_compare_func_t prp_cmp,
                  H5P_prp_close_func_t prp_close);
```

Description:

Outline of Processing:

H5P_peek()

Signature:

```
H5_DLL herr_t H5P_peek(H5P_genplist_t *plist, const char *name, void *value);
```

Description:

Outline of Processing:

H5P_poke()

Signature:

```
H5_DLL herr_t H5P_poke(H5P_genplist_t *plist, const char *name, const void *value);
```

Description:

Outline of Processing:

H5P_remove()

Signature:

```
H5_DLL herr_t H5P_remove(H5P_genplist_t *plist, const char *name);
```

Description:

Outline of Processing:

H5P_set()

Signature:

```
H5_DLL herr_t H5P_set(H5P_genplist_t *plist, const char *name, const void *value);
```

Description:

Outline of Processing:

New Private API Function Outlines

Support for true multi-thread operations requires that we expose the version of property lists to other multi-thread code so as to avoid the possibility that property lists will be changed out from underneath multi-thread safe HDF5 library calls during execution. This requires a number of new versions of existing internal API calls, which are outlined below.

Next Steps

This RFC continues to be a work in progress. Next steps are:

- Finish the census and analysis of the property list and property list class callbacks. Document their function, and limitations on their processing required for multi-thread.
- Update the above data structure definitions to address any issues resulting from the prior task.
- Write the outlines of the public and private multi-thread H5P API calls.
- Circulate for review and update as neccessary.
- If all goes well, design the necessary test code.
- Begin implementation

Appendix 1 – H5P public API calls

After some type and macro definitions, this appendix contains a list of all the public H5P API calls, along with call trees, relevant structure definitions, and descriptions of their processing with particular emphasis on multi-thread safety issues. Apparent bugs / design issues are documented in red. Note that this data was derived by inspection, and thus some errors and/or oversights should be expected.

The list of public API calls was taken from H5Ppublic.h. All the public API calls in this file are decorated with Doxygen code to generate user level documentation on public API calls. I have included this code as it may be a useful addition to my own documentation.

Finally, I have skipped the detailed discussion for one API call that didn't seem interesting, and started skipping the discussions of multi-thread safety issues once it became obvious that a re-implementation was likely.

```
/* Define structure to hold class information */
struct H5P_genclass_t {
   struct H5P_genclass_t *parent;      /* Pointer to parent class */
   char *                 name;        /* Name of property list class */
   H5P_plist_type_t       type;        /* Type of property */
   size_t                 nprops;      /* Number of properties in class */
   unsigned               plists;      /* Number of property lists that have been
                                          created since the last modification to
                                          the class */
   unsigned               classes;     /* Number of classes that have been derived
                                          since the last modification to the class */
   unsigned               ref_count;   /* Number of outstanding ID's open on this
                                          class object */
   hbool_t                deleted;     /* Whether this class has been deleted and is
                                          waiting for dependent classes & proplists
                                          to close */
   unsigned               revision;    /* Revision number of a particular class
                                          (global) */
   H5SL_t               * props;       /* Skip list containing properties */

   /* Callback function pointers & info */
   H5P_cls_create_func_t  create_func; /* Function to call when a property list
                                          is created */
   void *                 create_data; /* Pointer to user data to pass along to
                                          create callback */
   H5P_cls_copy_func_t    copy_func;   /* Function to call when a property list
                                          is copied */
   void *                 copy_data;   /* Pointer to user data to pass along to
                                          copy callback */
   H5P_cls_close_func_t   close_func;  /* Function to call when a property list
                                          is closed */
   void *                 close_data;  /* Pointer to user data to pass along to
                                          close callback */
};


/* Define structure to hold property list information */
```

```
struct H5P_genplist_t {
   H5P_genclass_t *pclass;     /* Pointer to class info */
   hid_t          plist_id;   /* Copy of the property list ID (for use in
                                 close callback) */
   size_t         nprops;     /* Number of properties in class */
   hbool_t        class_init; /* Whether the class initialization callback
                                 finished successfully */
   H5SL_t *       del;        /* Skip list containing names of deleted properties */
   H5SL_t *       props;      /* Skip list containing properties */
};
```

```
/* Generic Property Class ID class */
static const H5I_class_t H5I_GENPROPCLS_CLS[1] = {{
   H5I_GENPROP_CLS,               /* ID class value */
   0,                             /* Class flags */
   0,                             /* # of reserved IDs for class */
   (H5I_free_t)H5P__close_class_cb /* Callback routine for closing objects of this
class */
}};

/* Generic Property List ID class */
static const H5I_class_t H5I_GENPROPLST_CLS[1] = {{
   H5I_GENPROP_LST,              /* ID class value */
   0,                            /* Class flags */
   0,                            /* # of reserved IDs for class */
   (H5I_free_t)H5P__close_list_cb /* Callback routine for closing objects of this
class */
}};

/* Define structure to hold property information */
typedef struct H5P_genprop_t {
   /* Values for this property */
   char *            name;        /* Name of property */
   size_t            size;        /* Size of property value */
   void *            value;       /* Pointer to property value */
   H5P_prop_within_t type;        /* Type of object the property is within */
   hbool_t           shared_name; /* Whether the name is shared or not */

   /* Callback function pointers & info */
   H5P_prp_create_func_t  create; /* Function to call when a property is created */
   H5P_prp_set_func_t     set;    /* Function to call when a property value is set */
   H5P_prp_get_func_t     get;    /* Function to call when a property value is
retrieved */
   H5P_prp_encode_func_t  encode; /* Function to call when a property is encoded */
   H5P_prp_decode_func_t  decode; /* Function to call when a property is decoded */
   H5P_prp_delete_func_t  del;    /* Function to call when a property is deleted */
   H5P_prp_copy_func_t    copy;   /* Function to call when a property is copied */
   H5P_prp_compare_func_t cmp;    /* Function to call when a property is compared */
   H5P_prp_close_func_t   close;  /* Function to call when a property is closed */
} H5P_genprop_t;
```

```
/**
* \ingroup GPLO
*
* \brief Terminates access to a property list
*
* \plist_id
*
* \return \herr_t
*
* \details H5Pclose() terminates access to a property list. All property
*          lists should be closed when the application is finished
*          accessing them. This frees resources used by the property
*          list.
*
* \since 1.0.0
*
*/
H5_DLL herr_t H5Pclose(hid_t plist_id);
```

```
H5Pclose()
 +-H5I_get_type()
 +-H5I_dec_app_ref()
    +-H5I__dec_app_ref()
       +-H5I__dec_ref()
       |  +-H5I__find_id()
       |  |  +- …
       |  +-(type_info->cls->free_func)((void *)info->object, request)
       |  |  ||
       |  | H5P__close_list_cb() // in this case
       |  |  +-H5P_close()
       |  |     | // iterate through the plist class close functions.
       |  |     +-(tclass->close_func)(plist->plist_id, tclass→close_data);
       |  |     |
       |  |     +-H5SL_create() // create seen list
       |  |     |
       |  |     | // iterate through properties running the close function
       |  |     | // on each property and inserting its name in the seen list
       |  |     +-H5SL_count(plist->props)
       |  |     +-H5SL_first(plist->props)
       |  |     +-(H5P_genprop_t *)H5SL_item(curr_node)
       |  |     +-(tmp->close)(tmp->name, tmp->size, tmp→value)
       |  |     +-H5SL_insert(seen, tmp->name, tmp->name)
       |  |     +-H5SL_next(curr_node)
       |  |     |
       |  |     | // repeat the above process iterating through the parent
       |  |     | // plist classes.
       |  |     +-H5SL_first(tclass→props)
       |  |     +-(H5P_genprop_t *)H5SL_item(curr_node);
       |  |     +-H5SL_search(seen, tmp->name)
       |  |     +-H5SL_search(plist->del, tmp->name)
       |  |     +-H5MM_malloc()
       |  |     +-H5MM_memcpy()
       |  |     +-(tmp->close)(tmp->name, tmp->size, tmp_value)
       |  |     +-H5MM_xfree()
       |  |     +-H5SL_insert(seen, tmp->name, tmp->name)
       |  |     +-H5SL_next(curr_node);
       |  |     |
       |  |     +-H5P__access_class(plist→pclass, H5P_MOD_DEC_LST)
       |  |     |  +-H5MM_xfree()
       |  |     |  +-H5SL_destroy()
       |  |     |  |  +- …
       |  |     |  +-H5P__access_class(par_class, H5P_MOD_DEC_CLS)
```

```
|   |      |     + …
|   |      +-H5SL_close(seen)
|   |      +-H5SL_destroy(plist->del, H5P__free_del_name_cb, NULL);
|   |      |  +- … // eventually
|   |      |       H5P__free_del_name_cb()
|   |      |        +-H5MM_xfree()
|   |      +-H5SL_destroy(plist->props, H5P__free_prop_cb, &make_cb);
|   |      |  +- … // eventually
|   |      |        H5P__free_prop_cb()
|   |      |         | // tprop→close is not called in this case because *make_cb
|   |      |         | // is FALSE.
|   |      |         +-(tprop->close)(tprop->name, tprop->size, tprop->value);
|   |      |         |  +- … // property specific – may not exist
|   |      |         +-H5P__free_prop(tprop);
|   |      |            +-H5MM_xfree()
|   |      |            +-H5FL_FREE()
|   |      +-H5FL_FREE(H5P_genplist_t, plist);
|   |      +-H5SL_close(seen);
|  +-H5I__remove_common()
+-H5I__find_id()
   +- …
```

In a nutshell:

Decrement the ref count on the target property list. If it drops to zero, remove it from the index, and delete it.

In greater detail:

If the supplied hid_t is H5P_DEFAULT, **H5Pclose()** does nothing and returns.

If the supplied hid_t is not associated with a property list, H5Pclose() flags an error and returns. Otherwise, it calls H5I_dec_app_ref() and returns.

H5I_dec_app_ref() is basically a pass through. It preforms some sanity checks, and the calls H5I__dec_app_ref(id, H5_REQUEST_NULL), and returns whatever value H5I__dec_app_ref() returns.

H5I__dec_app_ref() calls H5I__dec_ref() to decrement the regular ref count on the target. If H5I__dec_ref() returns a positive value (indicating that the regular reference count has not been decremented to zero), the function calls H5I__find_id() to obtain a pointer to the instance of H5I_id_info_t associated with the ID. This in hand, the function decrements the application reference count. The function returns either the value return by H5I__dec_ref() (if it is non-positive), or the application reference count after it has been decremented.

H5I__dec_ref() first calls H5I__find_id() to obtain a pointer (info) to the instance of H5I_id_info_t associated with the target index entry.

If info->count is greater that one, it decrements that value, and returns it to the caller.

If info→count is one, it accesses the H5I_type_info_array_g global to look up the pointer to the instance of H5I_type_info_t associated with the target, calls type_info→free_func() (which will be H5P__close_list_cb() in this case) to free info→object, calls H5I__remove_common() to remove *info from the index, and returns 0.

H5P_close_list_cb() calls H5P_close(), flags an error if it fails, and returns.

H5P_close() is a lengthy, and involved function – it performs the following actions:

- Test to see if the property list initialization function completed (i.e. plist→class_init == TRUE). If it did, execute the close function of the parent property list class (i.e. plist→pclass→close_func) if it exists, along with those of any property list classes from which the parent property list class was derived (i.e. plist→pclass→parent→close_func, etc).

- Create a skip list – call it "seen". This is used to track the property list entries that have been encountered in the subsequent scan of the target property list, and the property list class(es) from which the property list was derived. This list is then used to ensure that the close callback for each property encountered is only called the first time it is seen.

- Scan the target property list (i.e. the plist→props skip list). It should contain only properties that have been modified from their default values. For each such property, add it to the "seen" list, and call its close callback if it exists.

- Scan the list of properties in the parent property list class. For each property, test to see if it appears in either the seen list, or the target property list's deleted list (plist→del). If it does not, add the property to the seen list, and call its close function if it exists.

  If the parent property list class has a parent property list class, repeat the process on the parent until the root of the property list class hierarchy is reached.

- Call H5P__access_class(plist->pclass, H5P_MOD_DEC_LST) to decrement the parent property list class's dependent property list count.

- Free the skip lists associated with the target property list (plist→del and plist→props) via calls to H5SL_destroy(plist->del, H5P__free_del_name_cb, NULL) and H5SL_destroy(plist->props, H5P__free_prop_cb, &make_cb) respectively.

  Note that H5P__free_del_name_cb just discards the string containing the name of the discarded property. H5P__free_prop_cb can call the close callback for the target property, but does not in this case because *make_cb is zero (i.e. FALSE).

- Finally, call H5FL_FREE() to discard the base plist structure (an instance of H5P_genplist_t – see above for definition).

Multi-thread safety concerns:

Skip list implementation is not thread safe.

Property List classes, property lists, and properties are all subject to simultaneous access and / or modification – with the obvious potential for corruption.

```
/**
* \ingroup GPLOA
*
* \brief Closes an existing property list class
*
* \plistcls_id{plist_id}
*
* \return \herr_t
*
* \details H5Pclose_class() removes a property list class from the library.
*          Existing property lists of this class will continue to exist,
*          but new ones are not able to be created.
*
* \since 1.4.0
*
*/
H5_DLL herr_t H5Pclose_class(hid_t plist_id);

H5Pclose_class()
 +-H5I_get_type()
 +-H5I_dec_app_ref()
    +-H5I__dec_app_ref()
       +-H5I__dec_ref()
       |  +-H5I__find_id()
       |  |  +- …
       |  +-(type_info->cls->free_func)((void *)info->object, request)
       |  |  ||
       |  | H5P__close_class_cb() // in this case
       |  |  +-H5P__close_class()
       |  |     +-H5P__access_class(pclass, H5P_MOD_DEC_REF)
       |  |        +-H5MM_xfree()
       |  |        +-H5SL_destroy(pclass->props, H5P__free_prop_cb, &make_cb)
       |  |        |  +- … // eventually
       |  |        |   H5P__free_prop_cb()
       |  |        |    +-(tprop->close)(tprop->name, tprop->size, tprop->value);
       |  |        |    |  +- … // property specific - may not exist
       |  |        |    +-H5P__free_prop(tprop);
       |  |        |       +-H5MM_xfree()
       |  |        |       +-H5FL_FREE()
       |  |        +-H5FL_FREE()
       |  |        +-H5P__access_class(par_class, H5P_MOD_DEC_CLS);
       |  |           +- … // note recursion
       |  +-H5I__remove_common()
       +-H5I__find_id()
          +- …
```

In a nutshell:

Decrement the reference count on the target property list class. If this reference count drops to zero, the class is removed from the index, and is marked as deleted. The class is not actually discarded until both the number of property lists that instantiate it, and the number of classes derived from it drop to zero as well.

In greater detail:

If the supplied hid_t is not associated with a property list class, **H5Pclose_class()** flags an error and returns. Otherwise, it calls H5I_dec_app_ref() and returns.

H5I_dec_app_ref() is basically a pass through. It preforms some sanity checks, and the calls H5I__dec_app_ref(id, H5_REQUEST_NULL), and returns whatever value H5I__dec_app_ref() returns.

H5I__dec_app_ref() calls H5I__dec_ref() to decrement the regular ref count on the target. If H5I__dec_ref() returns a positive value (indicating that the regular reference count has not been decremented to zero), the function calls H5I__find_id() to obtain a pointer to the instance of H5I_id_info_t associated with the ID. This in hand, the function decrements the application reference count. The function returns either the value return by H5I__dec_ref() (if it is non-positive), or the application reference count after it has been decremented.

H5I__dec_ref() first calls H5I__find_id() to obtain a pointer (info) to the instance of H5I_id_info_t associated with the target index entry.

If info->count is greater that one, it decrements that value, and returns it to the caller.

If info→count is one, it accesses the H5I_type_info_array_g global to look up the pointer to the instance of H5I_type_info_t associated with the target, calls type_info→free_func() (which will be H5P__close_class_cb() in this case) to free info→object, calls H5I__remove_common() to remove *info from the index, and returns 0.

H5P_close_class_cb() calls H5P__close_class(), flags an error if it fails, and returns.

H5P__close_class() calls H5P__access_class(pclass, H5P_MOD_DEC_REF), flags an error if it fails, and returns.

H5P__access_class() maintains counts of the number of property lists and/or property list classes that depend on the target property list class.

In this case it is called with the H5P_MOD_DEC_REF op code (actually an instance of H5P_class_mod_t enumerated type), which directs it to decrement pclass→ref_count and set pclass→deleted to TRUE if the ref count has dropped to zero.

Regardless of op code, it also checks to see if:

(pclass->deleted && pclass->plists == 0 && pclass->classes == 0)

If so, it:

- frees all class properties without calling the associated close callbacks,

- discards the skip list that contained the class properties,
- discards the instance of H5P_genclass_t that represented the property list class, and
- calls H5P__access_class(par_class, H5P_MOD_DEC_CLS) on the parent of the discarded class (if there was one).

The call to H5P__access_class(par_class, H5P_MOD_DEC_CLS) will decrement par_class→classes, before the check of:

(par_class->deleted && par_class->plists == 0 && par_class->classes == 0)

is run on the parent class – possibly resulting in its deletion as well.

Multi-thread safety concerns:

Skip list implementation is not thread safe.

Property List classes, and their associated properties are all subject to simultaneous access and / or modification – with the obvious potential for corruption.

```
/**
* \ingroup GPLO
*
* \brief Copies an existing property list to create a new property list
*
* \plist_id
*
* \return \hid_t{property list}
*
* \details H5Pcopy() copies an existing property list to create a new
*          property list. The new property list has the same properties
*          and values as the original property list.
*
* \since 1.0.0
*
*/
H5_DLL hid_t H5Pcopy(hid_t plist_id);

H5Pcopy()
 +-H5I_get_type() // verify they is either H5I_GENPROP_LST or H5I_GENPROP_CLS
 +-H5I_object()
 |  + …
 +-H5P_copy_plist() // if a property list
 |  +-H5FL_CALLOC()
 |  +-H5SL_create()
 |  |  +- …
 |  |
 |  | // copy the deleted list into the copy.  Also duplicate the seen list
 |  +-H5SL_count()
 |  |  +- …
 |  +-H5SL_first()
 |  |  +- …
 |  +-H5MM_xstrdup()
 |  +-H5SL_insert()
 |  |  + …
 |  +-H5SL_next()
 |  |  + …
 |  |
 |  | // copy the properties from plist->props
 |  +-H5SL_first()
 |  |  +- …
 |  +-H5SL_item()
 |  |  +- …
 |  +-H5P__dup_prop()
 |  |  +-H5FL_MALLOC()
 |  |  +-H5MM_memcpy()
 |  |  +-H5MM_xstrdup()
 |  +-(new_prop->copy)(new_prop->name, new_prop->size, new_prop→value)
 |  |  +- …  // property specific
 |  +-H5P__add_prop()
 |  |  +-H5SL_insert()
 |  |     +- …
 |  +-H5P__free_prop() // error recovery
 |  |  +- …
 |  +-H5SL_insert()
 |  |  +- …
 |  +-H5SL_next()
 |  |  +- …
 |  |
 |  | // copy properties from parent property list class(es) if required.
 |  +-H5SL_first()
 |  |  +- …
```

```
|  +-H5SL_item()
|  |  +- …
|  +-H5SL_search()
|  |  +- …
|  +-H5P__do_prop_cb1(plist->props, tmp, tmp->copy)
|  |  +-H5MM_malloc()
|  |  +-cb(prop->name, prop->size, tmp_value) // cb == tmp→copy
|  |  | +- // property specific
|  |  +-H5P__dup_prop(prop, H5P_PROP_WITHIN_LIST)
|  |  |  +-H5FL_MALLOC()
|  |  |  +-H5MM_memcpy()
|  |  |  +-H5MM_xstrdup()
|  |  |  +-H5MM_xfree() // error cleanup
|  |  |  +-H5FL_FREE()  // error cleanup
|  |  +-H5MM_memcpy()
|  |  +-H5P__add_prop()
|  |  |  +-H5SL_insert()
|  |  |     +- …
|  |  +-H5MM_xfree () // error cleanup
|  |  +-H5P__free_prop()
|  |  +-H5P__free_prop()
|  |     +-H5MM_xfree()
|  |     +-H5FL_FREE()
|  +-H5SL_insert()
|  |  +- …
|  +-H5SL_next()
|  |  +- …
|  |
|  +-H5P__access_class(new_plist->pclass, H5P_MOD_INC_LST)
|  |  +-H5MM_xfree()    // can't happen in this case
|  |  +-H5SL_destroy()  // can't happen in this case
|  |  |  +- …
|  |  +-H5P__access_class(par_class, H5P_MOD_DEC_CLS) // can't happen in this case
|  |     + …
|  |
|  +-H5I_register(H5I_GENPROP_LST, new_plist, app_ref)
|  |  +- …
|  |
|  | // tclass is initialized as plist→parent, and then scans up the inheritance tree
|  +-(tclass->copy_func)(new_plist_id, old_plist->plist_id,
|  |  |                  old_plist→pclass→copy_data)
|  |  +- … // class specific
|  |
|  +-H5I_remove() // error cleanup
|     +- …
|
+-H5P__copy_pclass // if a property class
|  +-H5P__create_class()
|  |  +-H5FL_CALLOC()
|  |  +-H5MM_xstrdup()
|  |  +-H5SL_create()
|  |  |  +- …
|  |  +-H5P__access_class(par_class, H5P_MOD_INC_CLS)
|  |  |  +-H5MM_xfree()
|  |  |  +-H5SL_destroy()
|  |  |  |  +- …
|  |  |  +-H5P__access_class(par_class, H5P_MOD_DEC_CLS) // can't happen in this case
|  |  |     + …
|  |  +-H5MM_xfree() // error cleanup
|  |  +-H5SL_destroy()
|  |     +- … // eventually
|  |          H5P__free_prop_cb()
```

```
|   |            +-(tprop->close)(tprop->name, tprop->size, tprop->value);
|   |            |  +- … // property specific - may not exist
|   |            +-H5P__free_prop(tprop);
|   |               +-H5MM_xfree()
|   |               +-H5FL_FREE()
|   |
|   +-H5SL_first()
|   |  +- …
|   +-H5P__dup_prop()
|   |  +- // see above
|   +-H5P__add_prop()
|   |  +- // see above
|   +-H5SL_next()
|   |  +- …
|   +-H5P__close_class() // error recovery
|      +-H5P__access_class(pclass, H5P_MOD_DEC_REF)
|         +-H5MM_xfree()
|         +-H5SL_destroy()
|         |  +- …
|         +-H5P__access_class(par_class, H5P_MOD_DEC_CLS)
|            +- …
+-H5I_register()   // if a property class
|  +- …
+-H5P__close_class() // error cleanup on H5I_register()
   +- // see above
```

In a nutshell:

Duplicate the supplied property list or property list class, insert the duplicate in the appropriate index, and return the associated hid_t.

It is interesting to note that the user documentation only discusses copying property lists – is there a reason for this?

Also, the duplicate property list need not be an exact duplicate, as its property list will contain any properties that have been deleted from the base property list. These added properties are copied from the parent property list class(es).

Is this a feature or a bug?

In greater detail:

H5Pcopy() verifies that the supplied hid_t refers to either a property list or a property list class, and verifies that the target of the ID actually exists. If any of these checks fail, H5Pcopy() fails.

H5Pcopy() then checks to see if the supplied id is that of a property list, or a property list class.

If it is a property list, it calls H5P_copy_plist(), flags an error it it fails, and returns.

If it is a property list class, if first calls H5P_copy_pclass(). If that function succeeds, it then calls H5I_register() to insert the new property list class in appropriate index. If H5I_register fails, it calls H5P_close_class() to discard the new property list class before returning. See H5Pclose_class() above for details on H5P_close_class().

COPY PROPERTY LIST CASE:

H5P_copy_plist() first verifies that the supplied old_plist pointer is not NULL, and then allocates a new instance of H5P_gen_plist_t and stores its address in new_plist.

It then sets new_plist→pclass = old_plist→pclass – which is to say that the new property list will be derived from the same property list class as the old property list.

It then initializes:

```
new_plist→nprops = 0;           // the plist is empty to begin with
new_plist→class_init = FALSE;  // until the class callback completes
```

and calls H5SL_create() to create the new_plist→props and new_plist→del skip lists. Both of these lists are empty to begin with.

Similarly, it creates the "seen" skip list used to track properties that have already been seen and thus had their copy callbacks invoked – thus avoiding invoking the copy callback again on the property of the same name in a parent property list class.

It then copies the contents of the old_plist→del skip list into the new_plist→del skip list. The del skip lists contain the names of properties that have been deleted from old_plist.

It then scans through old_plist→props and performs the following operations on each property it encounters:

Duplicate the property via a call to H5P__dup_prop() (specifically, new_prop = H5P_dup_prop(old_prop, H5P_PROP_WITHIN_LIST).

> In this context, **H5P__dup_prop()** allocates a new instance of H5P_genprop_t (new_prop), copies the image of *old_prop into it, and (if old_prop→shared_name is FALSE), duplicates the string pointed to by old_prop→name and stored its address in new_prop->name. It then duplicates the buffer pointed to by old_prop→value (if it exists) storing the address of the duplicate in new_prop→value, and returns the address of the new instance of H5P_genprop_t
>
> Note that the name would be shared if the property was copied from a class – which can't happen in this case.

Call the copy callback for the property if it exists (specifically, (new_prop->copy)(new_prop->name, new_prop->size, new_prop→value), calling H5P__free_prop() if this copy fails.

Insert the new property into new_plist→props via a call to H5P__add_prop(), again calling H5P__free_prop() if the insertion fails. **H5P__add_prop()** simply calls H5SL_insert() to perform the insertion.

Insert the name of the new property into the "seen" skip list.

Increment nseen.

Increment new_plist→nprops.

H5P_copy_plist() then scans through the properties of the parent property list class (i.e. tclass->props), and then for any property list classes that the parent property list class may be derived from. For each such property (prop) that is not in the "seen" list, the following operations are performed:

1. Test to see if prop→copy is defined. If it is, duplicate the property and insert it into new_plist→props. Do this via a call to H5P__do_prop_cb1(new_plist->props, prop, prop→copy). In this context, **H5P__do_prop_cb1()** does the following:

    - Allocate a buffer (tmp_value) of size equal to the size of the value of the property (prop→size),

    - memcopy the value of the property (prop→value) into tmp_value,

    - Call (prop→copy)(prop→name, prop→size, tmp_value),
      *** need spec on what copy is supposed to do ***

    - Call H5P__dup_prop(prop, H5P_PROP_WITHIN_LIST). In this context, **H5P__dup_prop()** allocates a new instance of H5P_genprop_t, copies the image of *prop into it, and (if prop→shared_name is FALSE), duplicates the string pointed to by old_prop→name and stored its address in new_prop->name. It then duplicates the buffer pointed to by prop→value (if it exists) storing the address of the duplicate in pcopy->value, and returns the address of the new instance of H5P_genprop_t

    - Memcpy tmp_value into new_prop→value.

    - Call H5P__add_prop() to insert the new property into new_slist→props.

    - Discard tmp_value.

◦ On failure, call H5P__free_prop() to discard the copy of the property.

Observe that while all properties in old_plist→props are copied into new_plist→props, only properties that have copy callbacks are copied from parent property list class(es) into new_plist→props.

2. Add the name of the new property to the "seen" skip list

3. Increment nseen

4. Increment new_plist→nprops

After the scan of the parent property list(s), H5P_copy_plist() increments the number of property lists derived from the parent property list class via a call to H5P__access_class(new_plist->pclass, H5P_MOD_INC_LST). In this context, H5P__access_class() increments pclass→plists and returns.

It then registers the new property list via a call to H5I_register(H5I_GENPROP_LST, new_plist, app_ref) where app_ref is a boolean parameter passed into H5P_copy_plist, and stores the new id in new_plist->plist_id.

It then calls the pclass→copy_func() for all parent property list classes for which the call exists. If any of these calls fail, it calls H5I_remove(new_plist→plist_id) and flags an error. If all succeed, it sets new_plist→class_init to TRUE.

Finally, it sets the return value to the id of the new property list, and discards the "seen" skip list.

On failure, the new property list is discarded if it exists via a call to H5P_close(new_plist) – see discussion of H5Pclose() above for details of H5P_close().

COPY PROPERTY LIST CLASS CASE:

H5P_copy_pclass() first calls:

```
H5P__create_class(old_pclass→parent, old_pclass->name,
                  old_pclass->type, old_pclass→create_func,
                  old_pclass→create_data, old_pclass→copy_func,
                  old_pclass->copy_data, old_pclass→close_func,
                  old_pclass->close_data)
```

to create a new instance H5P_genclass_t. The pointer to the new instance is stored in new_pclass. In addition to allocating the the new instance, **H5P__create_pclass()** also:

Sets new_pclass→parent = old_pclass→parent

Duplicates the string pointed to by old_class→name, and sets new_pclass→name equal to the address of the duplicate string.

Initializes other fields of new_pclass as follows:

```
new_pclass->type        = old_pclass->type;
new_pclass->nprops      = 0;     /* No properties initially */
new_pclass->plists      = 0;     /* No property lists of this class yet */
new_pclass->classes     = 0;     /* No classes derived from this class yet */
new_pclass->ref_count   = 1;     /* This is the first ref to the new class */
new_pclass->deleted     = FALSE; /* Not deleted yet... :-) */
new_pclass->revision    = H5P_GET_NEXT_REV; /* Get a rev num for the class */
new_pclass->create_func = old_pclass->create_func;
new_pclass->create_data = old_pclass->create_data;
new_pclass->copy_func   = old_pclass->copy_func;
new_pclass->copy_data   = old_pclass->copy_data;
new_pclass->close_func  = old_pclass->close_func;
new_pclass->close_data  = old_pclass->close_data;
```

Allocates the properties skip list and set new_pclass→props to point to it.

Increments new_pclass→parent→classes via a call to H5P__access_class().

This done, H5P__copy_pclass() scans the property list (old_pclass→props) and does the following with each property found:

1. Call new_prop = H5P__dup_prop(old_prop, H5P_PROP_WITHIN_CLASS) to create a duplicate of the property.

   In this context, **H5P__dup_prop()** allocates a new instance of H5P_genprop_t, copies the image of *old_prop into it, and duplicates the string pointed to by old_prop→name and stored its address in new_prop→name. Note that the unconditional duplication of the name is forced by the H5P_PROP_WITHIN_CLASS flag. It then duplicates the buffer pointed to by old_prop→value (if it exists) storing the address of the duplicate in new_prop->value, and returns the address of the new instance of H5P_genprop_t

   Note that unlike the copy property list case, the property specific copy call is not invoked.

2. Insert the new_prop into the new_class→props skip list via a call to H5P__add_prop(new_pclass→props, new_prop).

3. Increment new_class→nprops

After the scan of the old_pclass→props completes, the function sets its return value to new_pclass and returns.

On failure, H5P__close_class(new_pclass) is called to cleanup. See discussion of H5Pclose_class() above for a description of this call.

Multi-thread safety concerns:

Skip list implementation is not thread safe.

Property list, property list classes, and their associated properties are all subject to simultaneous access and / or modification – with the obvious potential for corruption.

```
/**
* \ingroup GPLOA
*
* \brief Copies a property from one list or class to another
*
* \param[in] dst_id Identifier of the destination property list or class
* \param[in] src_id Identifier of the source property list or class
* \param[in] name Name of the property to copy
*
* \return \herr_t
*
* \details H5Pcopy_prop() copies a property from one property list or
*          class to another.
*
*          If a property is copied from one class to another, all the
*          property information will be first deleted from the destination
*          class and then the property information will be copied from the
*          source class into the destination class.
*
*          If a property is copied from one list to another, the property
*          will be first deleted from the destination list (generating a
*          call to the close callback for the property, if one exists)
*          and then the property is copied from the source list to the
*          destination list (generating a call to the copy callback for
*          the property, if one exists).
*
*          If the property does not exist in the class or list, this
*          call is equivalent to calling H5Pregister() or H5Pinsert() (for
*          a class or list, as appropriate) and the create callback will
*          be called in the case of the property being copied into a list
*          (if such a callback exists for the property).
*
* \since 1.6.0
*
*/
H5_DLL herr_t H5Pcopy_prop(hid_t dst_id, hid_t src_id, const char *name);

H5Pcopy_prop()
 +-H5I_get_type()
 |  +- …
 +-H5P__copy_prop_plist()
 |  +-H5I_object()
 |  |  +- …
 |  |
 |  | // property exists in destination property list
 |  +-H5P__find_prop_plist()
 |  |  +-H5SL_search()
 |  |     +- …
 |  +-H5P_remove()
 |  |  +-H5P__do_prop(plist, name, H5P__del_plist_cb, H5P__del_pclass_cb, NULL)
 |  |     +-H5SL_search()
 |  |     |  +- …
 |  |     |  // if the target property is in the property list
 |  |     +-(*plist_op)(plist, name, prop, udata) // H5P__del_plist_cb in this case
 |  |     |  // if the target property is in the property class
 |  |     +-(*pclass_op)(plist, name, prop, udata) // H5P__del_pclass_cb in this case
 |  +-H5P__dup_prop()
 |  |  +-H5FL_MALLOC()
 |  |  +-H5MM_memcpy()
 |  |  +-H5MM_xstrdup()
 |  |  +-H5MM_xfree() // error cleanup
```

```
|  |  +-H5FL_FREE()  // error cleanup
|  +-(new_prop->copy)(new_prop->name, new_prop->size, new_prop→value)
|  |  +- …  // property specific
|  +-H5P__add_prop()
|  |  +-H5SL_insert()
|  |     +- …
|  |
|  | // property doesn't exist in the destination property list
|  +-H5P__find_prop_plist()
|  |  +- // see above
|  +-H5P__create_prop()
|  |  +-H5FL_MALLOC()
|  |  +-H5MM_xstrdup()
|  |  +-H5MM_malloc()
|  |  +-H5MM_memcpy()
|  |  +-H5MM_xfree()  // error cleanup
|  |  +-H5FL_FREE()   // error cleanup
|  +-(new_prop->create)(new_prop->name, new_prop->size, new_prop→value)
|  |  +- … // prop. specific
|  +-H5P__add_prop()
|  |  +- // see above
|  |
|  | // error cleanup
|  +-H5P__free_prop() // error cleanup
|     +-H5MM_xfree()
|     +-H5FL_FREE()
|
+-H5P__copy_prop_pclass()
   +-H5I_object()
   |  +- …
   +-H5P__find_prop_pclass()
   |  +-H5SL_search()
   |     +- …
   +-H5P__exist_pclass()
   |  +-H5SL_search()
   |     +- …
   +-H5P__unregister()
   |  +-H5SL_search()
   |  |  +- …
   |  +-H5SL_remove()
   |  |  +- …
   |  +-H5P__free_prop()
   |     +-H5MM_xfree()
   |     +-H5FL_FREE()
   +-H5P__register()
   |  | // duplicate class if required
   |  +-H5P__create_class()
   |  |  +-H5FL_CALLOC()
   |  |  +-H5MM_xstrdup()
   |  |  +-H5SL_create()
   |  |  |  +- …
   |  |  +-H5P__access_class(par_class, H5P_MOD_INC_CLS)
   |  |  |  +-H5MM_xfree()
   |  |  |  +-H5SL_destroy()
   |  |  |  |  +- …
   |  |  |  +-H5P__access_class(par_class, H5P_MOD_DEC_CLS) // can't happen
   |  |  |     + …                                          // in this case
   |  |  +-H5MM_xfree() // error cleanup
   |  |  +-H5SL_destroy()
   |  |     +- … // eventually
   |  |          H5P__free_prop_cb()
   |  |           +-(tprop->close)(tprop->name, tprop->size, tprop->value);
   |  |           |  +- … // property specific – may not exist
```

```
|   |            +-H5P__free_prop(tprop);
|   |               +-H5MM_xfree()
|   |               +-H5FL_FREE()
|  +-H5SL_first()
|   |  +- …
|  +-H5P__dup_prop()
|   |  +- // see above
|  +-H5P__add_prop()
|   |  +- // see above
|  +-H5SL_next()
|   |  +- …
|   |
|  +-H5P__register_real()
|   |  +-H5SL_search()
|   |  |  +- …
|   |  +-H5P__create_prop()
|   |  |  +- // see above
|   |  +-H5P__add_prop()
|   |  |  +- // see above
|   |  +-H5P__free_prop()
|   |     +- // see above
|  +-H5P__close_class() // error recovery
|     +-H5P__access_class(pclass, H5P_MOD_DEC_REF)
|        +-H5MM_xfree()
|        +-H5SL_destroy()
|        |  +- …
|        +-H5P__access_class(par_class, H5P_MOD_DEC_CLS)
|           +- …
+-H5I_subst()
|  +- …
+-H5P__close_class()
   +- // see above
```

In a nutshell:

Copy a property from one property list or property list class to another.

In greater detail:

H5Pcopy_prop() first verifies that the objects referred to by the source and destination ids are either both property lists, or both property list classes. The function flags an error and returns if this is not the case.

Note: No test to verify that source and destination are distinct. Subsequent review indicates that unique ids need not refer to unique property list classes. So far this doesn't seem to be the case for property lists.

If both source and destination are property lists, the function then calls:

```
H5P__copy_prop_plist(dst_id, src_id, name)
```

and returns success if that function succeeds, and failure if it fails. Otherwise, if both source and destination are property list classes, the function calls:

```
H5P__copy_prop_pclass(dst_id, src_id, name)
```

again returning success if the function succeeds and failure if it fails.

PROPERTY LIST CASE:

H5P__copy_prop_plist() first makes calls to H5I_object() to obtain pointers to the source and destination property lists (src_plist and dst_plist) (Note: no test for distinctness).

This done, the function calls H5P__find_prop_plist(dst_plist, name), which calls H5SL_search() to see if the target property already exists in dst_plist.

If it does, H5P__copy_prop_plist():

1. Calls H5P__remove(dst_plist, name) to remove the target property from the destination plist.

   H5P__remove() is mostly a pass through function. It does some sanity checking, and then calls

   ```
   H5P__do_prop(dst_plist, name, H5P__del_plist_cb, H5P__del_pclass_cb, NULL)
   ```

   In this context, H5P__do_prop() first searches dst_plist→del to see if the target property has already been deleted – and flags an error if it has been.

   It then tries to find the named property in dst_plist→props via the call

   ```
   prop = H5SL_search(dst_plist→props, name).
   ```

   If successful, it calls

   ```
   H5P__del_plist_cb(dst_plist, name, prop, NULL)
   ```

   and returns success or failure if this call succeeds or fails.

   If this search fails, H5P__do_prop() searches the parent property list class(es) for the target property, and if successful calls:

   ```
   H5P__del_pclass_cb(dst_plist, name, prop, NULL)
   ```

   and again returns success or failure if this call succeeds or fails.

If the searches of the supplied property list and its parent property list class(es) fail, H5P__do_prop() returns failure.

H5P__del_plist_cb() calls the properties delete callback

```
(*(prop->del))(plist->plist_id, name, prop->size, prop→value)
```

(What does the delete callback use the plist_id for? MT issues? Possible re-entry into H5I?)

if it exists, duplicates the property name string and inserts it into the dst_plist→del skip list, deletes the property from the dst_plist→props skip list, frees it, and decrements dst_plist→nprops.

H5P__del_pclass_cb() is similar to H5P__del_plist_cb() but subtly different.

Like H5P__del_plist_cb() it calls the properties delete callback if it exists. However, before it does so, it duplicates *(prop→value), and passes a pointer to this duplicate as the final parameter to the properties delete callback.   After that, it duplicates the property name string and inserts it into the dst_plist→del skip list, and decrements dst_plist→nprops. Note, however, that since the target property is not in the dst_plist→props skip list, it doesn't attempt to remove it.

2. Calls prop = H5P__find_prop_plist(src_plist, name) to get a pointer to the source property.

   H5P__find_prop_plist() first searches src_plist→del to see if the property has been deleted, and returns an error if it has been.

   It then searches src_plist→props, and returns a pointer to the target property if this search is successful.

   If this search fails, it searches for the property in the parent property list class(es), and returns a pointer to the target property if this search succeeds.

   If neither search succeeds, it returns an error.

3. Calls new_prop = H5P__dup_prop(prop, H5P_PROP_WITHIN_LIST). **H5P__dup_prop()** allocates a new instance of H5P_genprop_t (new_prop), copies the image of *prop into it, and (if prop→shared_name is FALSE), duplicates the string pointed to by prop→name and stored its address in new_prop→name, duplicates the buffer pointed to by prop→value (if it exists) storing the address of the duplicate in new_prop→value, and returns the address of the new instance of H5P_genprop_t

4. Calls the property copy callback if it exists – (new_prop->copy)(new_prop->name, new_prop->size, new_prop→value).

5. Calls H5P__add_prop(dst_plist->props, new_prop) to insert the new property in the destination property list. **H5P__add_prop()** does this via a call to

   H5SL_insert(dst_plist->props, new_prop, new_prop->name)

6. Increments dst_plist→nprops.

If, on the other hand, the target property doesn't already exist in the destination property list, H5P__copy_prop_plist() proceeds as follows:

1. Call prop = H5P__find_prop_plist(src_plist, name) to obtain a pointer to the target property. See discussion above.

2. Create the new property via the call

```
new_prop = H5P__create_prop(prop->name, prop->size,
                            H5P_PROP_WITHIN_LIST, prop→value,
                            prop->create, prop->set, prop->get,
                            prop->encode, prop→decode,
                            prop->del, prop->copy, prop->cmp,
                            prop→close)
```

   After some sanity checks, **H5P__create_prop()** allocates a new instance of H5P_genprop_t (new_prop), duplicates *(prop→name) and sets new_prop→name to point to it. Similarly, if prop→value is not NULL, it duplicates *(prop→value), and sets new_prop→value to point to the copy.

   In this context, it initializes the remaining fields of *new_prop from its parameters as follows:

```
new_prop→shared_name = FALSE
new_prop→size        = prop→size;
new_prop→type        = H5P_PROP_WITHIN_LIST;
new_prop→create      = prop→create;
new_prop→set         = prop→set;
new_prop→get         = prop→get;
new_prop→encode      = prop→encode;
new_prop→decode      = prop→decode;
new_prop→del         = prop→del;
new_prop→copy        = prop→copy;
new_prop→cmp         = prop→cmp;
new_prop→close       = prop→close;
```

While I don't think it can happen in this case, if prop→cmp were NULL, new_prop→cmp would be set to &memcmp.

On success, H5P__create_prop() returns the pointer to the new instance of H5P_genprop_t (i.e. new_prop).

3. Call the property creation callback if it exists:

```
(new_prop->create)(new_prop->name, new_prop→size,
                   new_prop→value)
```

4. Insert the new property into the dst_plist via the call

```
H5P__add_prop(dst_plist->props, new_prop)
```

5. Increment dst_plist→nprops

Whichever path is taken, H5P__copy_prop_plist() then returns. On error, *new_prop is discarded via a call to H5P__free_prop() prior to return.

PROPERTY LIST CLASS CASE:

H5P__copy_prop_pclass() first makes calls to H5I_object() to obtain pointers to the source and destination property list classes (src_pclass and dst_pclass) (No test for distinctness).

It then obtains a pointer to the target property in the source property list class via the call:

```
prop = H5P__find_prop_pclass(src_pclass, name)
```

which is essentially a pass through to H5SL_search(src_pclass→props, name).

H5P__copy_prop_pclass() then calls

```
H5P__exist_pclass(dst_pclass, name)
```

to determine whether the destination property list class already contains a property of the target name.

H5P__exist_class() does this by first running H5SL_search(dst_class→props, name). If this search fails, it repeats the process on dst_class→parent→props, and so on up the inheritance tree until a property of the specified name is found, or there are no further parent property list classes. It returns TRUE if such a property is found, and FALSE otherwise.

If H5P__exist_pclass() returns TRUE, H5P__copy_prop_pclass() calls

```
H5P__unregister(dst_pclass, name)
```

to remove the target property from the destination property list class.

H5P__unregister() calls

```
prop = H5SL_search(pclass->props, name)
```

to obtain a pointer to the target property, calls

```
H5SL_remove(pclass->props, prop→name)
```

to remove it from pclass→props, calls

```
H5P__free_prop(prop)
```

to free it, decrements pclass→nprops, sets pclass->revision equal to the global variable H5P_next_rev, increments H5P_next_rev, and returns.

Note: H5P__unregister() operates only on dst_pclass, but H5P__exist_pclass() will return TRUE if either dst_pclass or any property list class that dst_pclass is derived from contains a property of the target name. Thus it appears that it is possible for H5P__exist_pclass() to return TRUE, and for H5P__unregister() to fail because it is unable to find the target property. This appears to be a bug.

After removing the target property from the destination property list class, if necessary, H5P__copy_prop_pclass() saves a copy of the pointer dst_pclass in old_dst_pclass, and then calls

```
H5P__register(&dst_pclass, name, prop->size, prop->value,
              prop->create, prop->set, prop->get,
              prop->encode, prop->decode, prop->del,
              prop->copy, prop->cmp, prop→close)
```

The objective of **H5P__register()** is to insert the new property into the target property list. However, there is a problem if there are any extant property lists, or property list classes derived from dst_pclass. Specifically, there appears to be no method for updating the derived property lists and property list classes for changes to *dst_pclass. Instead, H5P__register() duplicates *dst_pclass, inserts the new property into the duplicate, and returns a pointer to the duplicate. The duplicate later replaces the earlier version of *dst_class in the index. The original version of *dst_pclass is then is only accessible via the parent pointers in its derived property lists and property list classes (not quite – as shall be seen H5Pget_class() can insert the old version of the property list class into the index, albeit with a new id. See discussion of H5Pget_class() for further details.). It is retained until both its plists (number of derived property lists) and classes (number of derived property list classes) fields drop to zero – at which point it is discarded.

With this background, we return to a detailed discussion of H5P__register().

H5P__register first tests to see if either dst_pclass→plists or dst_class→classes is positive. If either is, it calls

```
  new_pclass = H5P__create_class(dst_pclass→parent,
                                 dst_pclass→name,
                                 dst_pclass→type,
                                 dst_pclass→create_func,
                                 dst_pclass→create_data,
                                 dst_pclass→copy_func,
                                 dst_pclass->copy_data,
                                 dst_pclass→close_func,
                                 dst_pclass→close_data)
```

After some sanity checks, **H5P__create_class()** allocates a new instance of H5P_genclass_t, and stores its address in new_pclass. It then duplicates the string containing the name of the new property list class (dst_class→name in this case) and set new_pclass→name equal to the duplicate string.

H5P__create_class() then calls H5SL_create() to create the skip list used to store properties in the property list class, sets new_pclass→nprops to point to it, and then initializes the remaining fields of *new_pclass as follows (in this case):

```
new_pclass→parent      = dst_pclass→parent;
new_pclass->type        = dst_class->type;
new_pclass->nprops      = 0;            /* No properties initially */
new_pclass->plists      = 0;            /* No properties lists of this class yet */
new_pclass->classes     = 0;            /* No classes derived from this class yet */
new_pclass→ref_count    = 1;           /* This is the first reference to *new_class */
new_pclass->deleted     = FALSE;        /* Not deleted yet... :-) */
new_pclass->revision    = H5P_next_rev++;   /* Get a revision number for the class */
new_pclass->create_func = dst_pclass→create_func;
new_pclass->create_data = dst_pclass→create_data;
new_pclass->copy_func   = dst_pclass→copy_func;
new_pclass->copy_data   = dst_pclass→copy_data;
new_pclass->close_func  = dst_pclass→close_func;
new_pclass->close_data  = dst_pclass→close_data;
```

where H5P_next_rev is a global variable used to assign unique revision numbers to property list classes. Finally, H5P__create_class() calls

```
  H5P__access_class(new_pclass→parent, H5P_MOD_INC_CLS)
```

to increment new_pclass→parent→classes, and returns new_pclass.

With *new_pclass created, H5P__register must now populate it with copies of all properties that appear in dst_pclass. It does this by scanning dst_pclass→props, and performing the following operations on each property (old_prop) encountered:

1. Duplicate the property with a call to

   ```
   new_prop = H5P__dup_prop(old_prop, H5P_PROP_WITHIN_CLASS)
   ```

   In this context, **H5P__dup_prop()** allocates a new instance of H5P_genprop_t, copies the image of *old_prop into it, duplicates the string pointed to by old_prop→name and stored its address in new_prop→name duplicates the buffer pointed to by prop→value (if it exists) storing the address of the duplicate in pcopy→value, and returns the address of the new instance of H5P_genprop_t.

   Note that the unconditional duplication of the name is forced by the H5P_PROP_WITHIN_CLASS flag.

2. Insert new_prop into new_pclass via the call

   ```
   H5P__add_prop(new_pclass->props, new_prop)
   ```

3. Increment new_pclass->nprops

This done, H5P__register sets dst_pclass = new_pclass, completing processing for the dst_pclass→plists or dst_class→classes positive case. Note that the copy of dst_pclass made earlier is used later to detect whether a copy of dst_pclass has been created after H5P__register returns.

Whether either dst_pclass→plists or dst_class→classes is positive or not, H5P__register() next calls

```
H5P__register_real(dst_pclass, name, prop→size, prop→value,
                   prop→create, prop→set, prop→get, prop→encode,
                   prop→decode, prop→delete, prop→copy,
                   prop→cmp, prop→close)
```

Recall that dst_pclass may be either the original, or the duplicate at this point.

After some sanity checks, **H5P__register_real()** creates the new property via the call:

```
new_prop = H5P__create_prop(name, prop→size,
                            H5P_PROP_WITHIN_CLASS, prop→value,
                            prop→create, prop→set, prop→get,
                            prop→encode, prop→decode,
                            prop→delete, prop→copy,
                            prop→cmp, prop→close)
```

See the PROPERTY LIST CASE above for a detailed discussion of H5P__create_prop(). Note that in this case, new_prop→type is set to H5P_PROP_WITHIN_CLASS instead of H5P_PROP_WITHIN_LIST.

After new_prop is created, H5P__register_real() inserts it into dst_class via a call to

```
H5P__add_prop(dst_pclass->props, new_prop)
```

increments dst_pclass→nprops, sets dst_pclass→revision = H5P_next_rev, increments that global integer, and returns. Recall that dst_pclass may now point to a duplicate with the new property added.

Assuming no errors, H5P__register() returns immediately after H5P__register_real() returns.

After H5P__register() returns, H5P__copy_prop_pclass() compares dst_pclass with old_dst_pclass – the copy it made just before calling H5P__register(). If the two don't match, it must replace old_dst_pclass with dst_pclass in the index. It does this with the call

```
H5I_subst(dst_id, dst_pclass)
```

and then calls

```
H5P__close_class(old_dst_pclass)
```

which decrements old_dst_pclass→ref_count, and may delete it. See discussion in H5Pclose_class() above for further details.

Multi-thread safety concerns:

Skip list implementation is not thread safe.

Property list, property list classes, and their associated properties are all subject to simultaneous access and / or modification – with the obvious potential for corruption.

H5P_next_rev is a global used to assign unique ids. Mutual exclusion is required.

```
/**
* \ingroup GPLO
*
* \brief Creates a new property list as an instance of a property list class
*
* \plistcls_id{cls_id}
*
* \return \hid_t{property list}
*
* \details H5Pcreate() creates a new property list as an instance of
*          some property list class. The new property list is initialized
*          with default values for the specified class. The classes are as
*          follows:
*
* <table>
*   <tr>
*     <th>Class Identifier</th>
*     <th>Class Name</th>
*     <th>Comments</th>
*   </tr>
*   <tr>
*     <td>#H5P_ATTRIBUTE_CREATE</td>
*     <td>attribute create</td>
*     <td>Properties for attribute creation</td>
*   </tr>
*   <tr>
*     <td>#H5P_DATASET_ACCESS</td>
*     <td>dataset access</td>
*     <td>Properties for dataset access</td>
*   </tr>
*   <tr>
*     <td>#H5P_DATASET_CREATE</td>
*     <td>dataset create</td>
*     <td>Properties for dataset creation</td>
*   </tr>
*   <tr>
*     <td>#H5P_DATASET_XFER</td>
*     <td>data transfer</td>
*     <td>Properties for raw data transfer</td>
*   </tr>
*   <tr>
*     <td>#H5P_DATATYPE_ACCESS</td>
*     <td>datatype access</td>
*     <td>Properties for datatype access</td>
*   </tr>
*   <tr>
*     <td>#H5P_DATATYPE_CREATE</td>
*     <td>datatype create</td>
*     <td>Properties for datatype creation</td>
*   </tr>
*   <tr>
*     <td>#H5P_FILE_ACCESS</td>
*     <td>file access</td>
*     <td>Properties for file access</td>
*   </tr>
*   <tr>
*     <td>#H5P_FILE_CREATE</td>
*     <td>file create</td>
*     <td>Properties for file creation</td>
*   </tr>
*   <tr>
*     <td>#H5P_FILE_MOUNT</td>
*     <td>file mount</td>
```

```
*     <td>Properties for file mounting</td>
*   </tr>
*   <tr valign="top">
*     <td>#H5P_GROUP_ACCESS</td>
*     <td>group access</td>
*     <td>Properties for group access</td>
*   </tr>
*   <tr>
*     <td>#H5P_GROUP_CREATE</td>
*     <td>group create</td>
*     <td>Properties for group creation</td>
*   </tr>
*   <tr>
*     <td>#H5P_LINK_ACCESS</td>
*     <td>link access</td>
*     <td>Properties governing link traversal when accessing objects</td>
*   </tr>
* <tr>
*     <td>#H5P_LINK_CREATE</td>
*     <td>link create</td>
*     <td>Properties governing link creation</td>
*   </tr>
*   <tr>
*     <td>#H5P_OBJECT_COPY</td>
*     <td>object copy</td>
*     <td>Properties governing the object copying process</td>
*   </tr>
*   <tr>
*     <td>#H5P_OBJECT_CREATE</td>
*     <td>object create</td>
*     <td>Properties for object creation</td>
*   </tr>
*   <tr>
*     <td>#H5P_STRING_CREATE</td>
*     <td>string create</td>
*     <td>Properties for character encoding when encoding strings or
*       object names</td>
*   </tr>
*   <tr>
*     <td>#H5P_VOL_INITIALIZE</td>
*     <td>vol initialize</td>
*     <td>Properties for VOL initialization</td>
*   </tr>
* </table>
*
* This property list must eventually be closed with H5Pclose();
* otherwise, errors are likely to occur.
*
* \version 1.12.0 The #H5P_VOL_INITIALIZE property list class was added
* \version 1.8.15 For each class, the class name returned by
*                 H5Pget_class_name() was added.
*                 The list of possible Fortran values was updated.
* \version 1.8.0 The following property list classes were added at this
*                release: #H5P_DATASET_ACCESS, #H5P_GROUP_CREATE,
*                #H5P_GROUP_ACCESS, #H5P_DATATYPE_CREATE,
*                #H5P_DATATYPE_ACCESS, #H5P_ATTRIBUTE_CREATE
*
*
* \since 1.0.0
*
*/
H5_DLL hid_t H5Pcreate(hid_t cls_id);
```

```
H5Pcreate()
 +-H5I_object_verify()
 |  +- …
 +-H5P_create_id()
    +-H5P__create()
    |  +-H5FL_CALLOC()
    |  +-H5SL_create()
    |  |  +- …
    |  +-H5SL_first()
    |  |  +- …
    |  +-H5SL_item()
    |  |  +- …
    |  +-H5SL_search()
    |  |  +- …
    |  +-H5P__do_prop_cb1(plist->props, tmp, tmp->create)
    |  |  +-H5MM_malloc()
    |  |  +-cb(prop->name, prop->size, tmp_value) // cb == tmp→create
    |  |  +-H5P__dup_prop(prop, H5P_PROP_WITHIN_LIST)
    |  |  |  +-H5FL_MALLOC()
    |  |  |  +-H5MM_memcpy()
    |  |  |  +-H5MM_xstrdup()
    |  |  |  +-H5MM_xfree() // error cleanup
    |  |  |  +-H5FL_FREE()  // error cleanup
    |  |  +-H5MM_memcpy()
    |  |  +-H5P__add_prop()
    |  |  |  +-H5SL_insert()
    |  |  |     +- …
    |  |  +-H5MM_xfree () // error cleanup
    |  |  +-H5P__free_prop()
    |  |  +-H5P__free_prop()
    |  |     +-H5MM_xfree()
    |  |     +-H5FL_FREE()
    |  +-H5SL_insert()
    |  |  +- …
    |  +-H5SL_next()
    |  |  +- …
    |  +-H5P__access_class(plist->pclass, H5P_MOD_INC_LST)
    |  |   // in this case, increment the plist count in the pclass
    |  |
    |  +-H5SL_close() // error cleanup
    |  |  +- …
    |  +-H5SL_destroy() // error cleanup
    |  |  +- …
    |  +-H5SL_close()  // error cleanup
    |  |  +- …
    |  +-H5FL_FREE() // error cleanup
    |
    +-H5I_register()
    |  +- …
    +-(tclass->create_func)(plist_id, tclass→create_data) // class specific,
    |                                                     // may not exist
    +-H5I_remove() // error cleanup only
    |  +- …
    +-H5P_close() // error cleanup only
       +- // see H5Pclose() above
```

In a nutshell:

Create a new property list derived from the property list class indicated by the supplied id.

In greater detail:

H5Pcreate() first calls

```
pclass = H5I_object_verify(cls_id, H5I_GENPROP_CLS)
```

to obtain a pointer to the source property list class, calls

```
plist_id = H5P_create_id(pclass, TRUE)   // discussed below
```

to create the new property list and then returns plist_id.

H5P_create_id() starts by calling

```
plist = H5P__create(pclass)  // discussed below
```

to create the new property list, and then inserts it into the index with the callback

```
plist_id = H5I_register(H5I_GENPROP_LST, plist, app_ref)
```

Note that app_ref is TRUE in this case. H5P_create_id() then sets plist→plist_id = plist_id, and then scans up the property list class inheritance tree (i.e. tclass = plist→parent, tclass = plist→parent→parent, etc), calling the create function:

```
(tclass->create_func)(plist_id, tclass->create_data)
```

whenever it exists. This done, it sets pclass→class_init = TRUE and returns plist_id.

H5P__create() is the main routine for creating a new property list. It starts by allocating a new instance of H5P_genplist_t and seting plist to point to it. This done, it sets:

```
plist→pclass     = pclass;
plist→nprops     = 0;
plist→class_init = FALSE;
```

and creates the

```
plist→props
plist→del
```

skip lists with calls to H5SL_create(). The "seen" skip list is also created.

The next step is to populate the new property list. Do this by scanning the property lists of the parent property list class(es) and copying properties into the new property list. This is complicated by two factors:

First, only properties with create functions are copied into plist->props.

Second, in cases where pclass has one or more parent property list classes (i.e. pclass→parent != NULL), it appears to be possible that two or more of these property list classes will contain properties of the same name. This is handled by first scanning the property list of pclass, then pclass→parent (if it exists), and so forth, and for any given name, only coping the first property of that name encountered (and then only if its create function is defined). The "seen" skip list is used to track the names of properties already been considered for copying into the new property list.

For each property (prop) selected by this method, **H5P__create()** does the following:

1. Test to see if the property creation callback exists (i.e. prop→create != NULL). If it does, it calls

   ```
   H5P__do_prop_cb1(plist->props, prop, prop->create)
   ```

   H5P__do_prop_cb1() duplicates the property, and inserts it into plist→props. In this context, it does the following:

   - Allocate a buffer (tmp_value) of size equal to the size of the value of the property (prop→size),

   - memcopy the value of the property (prop→value) into tmp_value,

   - Call (prop→create)(prop→name, prop→size, tmp_value),
     *** need spec on what create is supposed to do ***

   - Call H5P__dup_prop(prop, H5P_PROP_WITHIN_LIST). In this context, **H5P__dup_prop()** allocates a new instance of H5P_genprop_t (new_prop), copies the image of *prop into it, sets new_prop→shared_name = TRUE, duplicates the buffer pointed to by prop→value (if it exists) storing the address of the duplicate in new_prop→value, and returns the address of the new instance of H5P_genprop_t.

   - Memcopy tmp_value into new_prop→value.

   - Call H5P__add_prop() to insert the new property into new_slist→props.

   - Discard tmp_value.

- On failure, call H5P__free_prop() to discard the copy of the property.

  Again, note that only properties that have create callbacks are copied from parent property list class(es) into plist→props.

  After H5P__do_prop_cb1() returns, increment pclass→nprops.

2. Regardless of whether prop→create != NULL, add prop→name to the "seen" skip list.

After populating plist→props, H5P__create() calls

```
H5P__access_class(plist->pclass, H5P_MOD_INC_LST)
```

which in this context increments plist→pclass→plists.

Finally, H5P__create() returns plist.

Multi-thread safety concerns:

Skip list implementation is not thread safe.

Property list, property list classes, and their associated properties are all subject to simultaneous access and / or modification – with the obvious potential for corruption.

```
/**
* \ingroup GPLOA
*
* \brief Creates a new property list class
*
* \plistcls_id{parent}
* \param[in] name        Name of property list class to register
* \param[in] create      Callback routine called when a property list is
*                        created
* \param[in] create_data Pointer to user-defined class create data, to be
*                        passed along to class create callback
* \param[in] copy        Callback routine called when a property list is
*                        copied
* \param[in] copy_data   Pointer to user-defined class copy data, to be
*                        passed along to class copy callback
* \param[in] close       Callback routine called when a property list is
*                        being closed
* \param[in] close_data  Pointer to user-defined class close data, to be
*                        passed along to class close callback
*
* \return \hid_t{property list class}
*
* \details H5Pcreate_class() registers a new property list class with the
*          library. The new property list class can inherit from an
*          existing property list class, \p parent, or may be derived
*          from the default "empty" class, NULL. New classes with
*          inherited properties from existing classes may not remove
*          those existing properties, only add or remove their own class
*          properties. Property list classes defined and supported in the
*          HDF5 library distribution are listed and briefly described in
*          H5Pcreate(). The \p create, \p copy, \p close functions are called
*          when a property list of the new class is created, copied, or closed,
*          respectively.
*
*          H5Pclose_class() must be used to release the property list class
*          identifier returned by this function.
*
* \since 1.4.0
*
*/
H5_DLL hid_t H5Pcreate_class(hid_t parent, const char *name,
                             H5P_cls_create_func_t create,
                             void *create_data, H5P_cls_copy_func_t copy,
                             void *copy_data, H5P_cls_close_func_t close,
                             void *close_data);
```

```
H5Pcreate_class()
 +-H5I_get_type()
 |  + …
 +-H5I_object()
 |  +- …
 +-H5P__create_class()
 |  +-H5FL_CALLOC()
 |  +-H5MM_xstrdup()
 |  +-H5SL_create()
 |  |  +- …
 |  +-H5P__access_class(par_class, H5P_MOD_INC_CLS)
 |  |  +-H5MM_xfree()
 |  |  +-H5SL_destroy()
 |  |  |  +- …
 |  |  +-H5P__access_class(par_class, H5P_MOD_DEC_CLS) // can't happen in this case
```

```
|   |      + …
|   +-H5MM_xfree() // error cleanup
|   +-H5SL_destroy()
|      +- … // eventually
|           H5P__free_prop_cb()
|             +-(tprop->close)(tprop->name, tprop->size, tprop->value);
|             |  +- … // property specific - may not exist
|             +-H5P__free_prop(tprop);
|                +-H5MM_xfree()
|                +-H5FL_FREE()
+-H5I_register()
|   +- …
+-H5P__close_class() // error cleanup
   +-H5P__access_class(pclass, H5P_MOD_DEC_REF)
      +-H5MM_xfree()
      +-H5SL_destroy()
      |   +- …
      +-H5P__access_class(par_class, H5P_MOD_DEC_CLS)
         +- // see above
```

In a nutshell:

Create a new property list class based on the supplied parent property list class, insert it in the index, and return its id.

In greater detail:

After input validations, **H5Pcreate_class()** checks to see if the parent property list class id is H5P_DEFAULT. If it is, it sets par_class = NULL. Otherwise it calls H5I_object to obtain a pointer to the parent class, and sets par_class to this value. (Note: a NULL par_class will trigger an assertion failure in H5P__create_class() in debug builds, but does not seem to trigger an error in production builds. This is probably a bug.)

This done, H5Pcreate_class() calls

```
pclass = H5P__create_class(par_class, name, H5P_TYPE_USER,
                           cls_create, create_data,
                           cls_copy, copy_data,
                           cls_close, close_data)
```

to create the new property list class.

H5P__create_class() is discussed in detail in the “COPY PROPERTY LIST CLASS” case of the discussion of H5Pcopy() above. Briefly, it creates a property list class with no properties and no derived property lists or property list classes, that is otherwise a duplicate of the parent property list class, and returns a pointer to it. In passing, it also increments dst_pclass->parent->classes.

When H5P__create_class() returns, H5Pcreate_class() calls

```
H5I_register(H5I_GENPROP_CLS, pclass, TRUE)
```

to insert the new property list class into the index, and returns the new property list class id.

Multi-thread safety concerns:

Skip list implementation is not thread safe.

Property list classes, and their associated properties are all subject to simultaneous access and / or modification – with the obvious potential for corruption.

```
/**
* \ingroup GPLO
*
* \brief Decodes property list received in a binary object buffer and
*          returns a new property list identifier
*
* \param[in] buf Buffer holding the encoded property list
*
* \return \hid_tv{object}
*
* \details Given a binary property list description in a buffer, H5Pdecode()
*          reconstructs the HDF5 property list and returns an identifier
*          for the new property list. The binary description of the property
*          list is encoded by H5Pencode().
*
*          The user is responsible for passing in the correct buffer.
*
*          The property list identifier returned by this function should be
*          released with H5Pclose() when the identifier is no longer needed
*          so that resource leaks will not develop.
*
* \note Some properties cannot be encoded and therefore will not be available
*       in the decoded property list. These properties are discussed in
*       H5Pencode().
*
* \since 1.10.0
*
*/
H5_DLL hid_t H5Pdecode(const void *buf);
```

```
H5Pdecode()
 +-H5P__decode()
    +-H5P__new_plist_of_type()
    |  +-H5I_object()
    |  |  +- …
    |  +-H5P_create_id()
    |     +-H5P__create()
    |     |  +-H5FL_CALLOC()
    |     |  +-H5SL_create()
    |     |  |  +- …
    |     |  +-H5SL_first()
    |     |  |  +- …
    |     |  +-H5SL_item()
    |     |  |  +- …
    |     |  +-H5SL_search()
    |     |  |  +- …
    |     |  +-H5P__do_prop_cb1(plist->props, tmp, tmp->create)
    |     |  |  +-H5MM_malloc()
    |     |  |  +-cb(prop->name, prop->size, tmp_value) // cb == tmp→create
    |     |  |  +-H5P__dup_prop(prop, H5P_PROP_WITHIN_LIST)
    |     |  |  |  +-H5FL_MALLOC()
    |     |  |  |  +-H5MM_memcpy()
    |     |  |  |  +-H5MM_xstrdup()
    |     |  |  |  +-H5MM_xfree() // error cleanup
    |     |  |  |  +-H5FL_FREE()  // error cleanup
    |     |  |  +-H5MM_memcpy()
    |     |  |  +-H5P__add_prop()
    |     |  |  |  +-H5SL_insert()
    |     |  |  |     +- …
    |     |  |  +-H5MM_xfree () // error cleanup
    |     |  |  +-H5P__free_prop()
    |     |  |  +-H5P__free_prop()
```

```
|     |  |     +-H5MM_xfree()
|     |  |     +-H5FL_FREE()
|     |  +-H5SL_insert()
|     |  |  +- …
|     |  +-H5SL_next()
|     |  |  +- …
|     |  +-H5P__access_class(plist->pclass, H5P_MOD_INC_LST)
|     |  |   // in this case, increment the plist count in the pclass
|     |  |
|     |  +-H5SL_close() // error cleanup
|     |  |  +- …
|     |  +-H5SL_destroy() // error cleanup
|     |  |  +- …
|     |  +-H5SL_close()  // error cleanup
|     |  |  +- …
|     |  +-H5FL_FREE() // error cleanup
|     |
|     +-H5I_register()
|     |  +- …
|     +-(tclass->create_func)(plist_id, tclass→create_data) // class specific,
|     |                                                     // may not exist
|     +-H5I_remove() // error cleanup only
|     |  +- …
|     +-H5P_close() // error cleanup only
|        +- // see H5Pclose() above
|
+-H5I_object()
|  +- …
+-HDstrlen()
+-H5P__find_prop_plist()
|  +-
+-H5MM_realloc()
+-(prop->decode)((const void **)&p, value_buf)
|  +- // property specific
|
+-H5P_poke()
|  +-H5P__do_prop(plist, name, H5P__poke_plist_cb,
|     |           H5P__poke_pclass_cb, &udata)
|     +-H5SL_search()
|     |  +- …
|     +-H5P__poke_plist_cb()
|     |  +-H5MM_memcpy()
|     +-H5P__poke_pclass_cb()
|        +-H5P__dup_prop(prop, H5P_PROP_WITHIN_LIST)
|        |  +-H5FL_MALLOC()
|        |  +-H5MM_memcpy()
|        |  +-H5MM_xstrdup()
|        |  +-H5MM_xfree() // error cleanup
|        |  +-H5FL_FREE()  // error cleanup
|        +-H5MM_memcpy()
|        +-H5P__add_prop()
|        |  +-H5SL_insert()
|        |     +- …
|        +-H5P__free_prop() // error cleanup
|             +-H5MM_xfree()
|             +-H5FL_FREE()
|
+-H5MM_xfree()
+-H5I_dec_ref() // error cleanup
   +- …
```

In a nutshell:

Given a buffer containing an encoded property list, decode the buffer, construct the property list described in the buffer, insert it in the index, and return the id associated with the decoded property list.

Note that only property lists derived from HDF5 defined property list classes are supported. This is not mentioned in the user documentation.

In greater detail:

H5Pdecode() simply calls H5P__decode() with the supplied buffer, and returns whatever H5P__decode() returns, flagging an error in passing if H5P__decode() fails.

H5P__decode() first reads the encoding version number from the first byte of the supplied buffer and fails if it is not H5P_ENCODE_VERS (currently defined to be zero).

It then loads the type of the property list (another byte) and fails if

type <= H5P_TYPE_USER or type >=H5P_TYPE_MAX_TYPE.

Observe that this disallows user defined property lists. H5P_TYPE_USER and H5P_TYPE_MAX_TYPE are both members of the enumerated type H5P_plist_type_t, whose definition is reproduced below:

```
typedef enum H5P_plist_type_t {
   H5P_TYPE_USER              = 0,
   H5P_TYPE_ROOT              = 1,
   H5P_TYPE_OBJECT_CREATE     = 2,
   H5P_TYPE_FILE_CREATE       = 3,
   H5P_TYPE_FILE_ACCESS       = 4,
   H5P_TYPE_DATASET_CREATE    = 5,
   H5P_TYPE_DATASET_ACCESS    = 6,
   H5P_TYPE_DATASET_XFER      = 7,
   H5P_TYPE_FILE_MOUNT        = 8,
   H5P_TYPE_GROUP_CREATE      = 9,
   H5P_TYPE_GROUP_ACCESS      = 10,
   H5P_TYPE_DATATYPE_CREATE   = 11,
   H5P_TYPE_DATATYPE_ACCESS   = 12,
   H5P_TYPE_STRING_CREATE     = 13,
   H5P_TYPE_ATTRIBUTE_CREATE  = 14,
   H5P_TYPE_OBJECT_COPY       = 15,
   H5P_TYPE_LINK_CREATE       = 16,
   H5P_TYPE_LINK_ACCESS       = 17,
   H5P_TYPE_ATTRIBUTE_ACCESS  = 18,
   H5P_TYPE_VOL_INITIALIZE    = 19,
   H5P_TYPE_MAP_CREATE        = 20,
   H5P_TYPE_MAP_ACCESS        = 21,
   H5P_TYPE_REFERENCE_ACCESS  = 22,
   H5P_TYPE_MAX_TYPE
```

```
} H5P_plist_type_t;
```

If the property list type is in range, H5P__decode() calls

```
plist_id = H5P__new_plist_of_type(type)
```

to create a new property list of the specified type, insert it in the index, and return its id. Given plist_id, H5P_decode() obtains a pointer to the new property list (plist) via a call to H5I_object(). It populates *plist by executing the following loop until the supplied buffer is exhausted.

- Test to see if the buffer is exhausted, and exit the loop if it is.

- Set name to point to the string in the buffer containing the next property name.

- Call

  ```
  prop = H5P__find_prop_plist(plist, name)
  ```

  to obtain a pointer to the existing property in the newly created property list of the specified name. Since plist is a newly created property list, prop should have the default value.

- The value_buf is a buffer that is provided to prop→decode(), and must be of length greater than or equal to prop→size. Check to see if the current value_buf is large enough, and if not, realloc() it to the required size, and make note of the new size in value_buf_size.

- Call

  ```
  prop->decode)((const void **)&p, value_buf)
  ```

  where p is a pointer to the current location in the supplied buffer. In addition to loading the value into value_buf, prop→decode() must update p to reflect the number of bytes read from the buffer.

- Call

  ```
  H5P_poke(plist, name, value_buf)
  ```

  to insert the value into the named property in the property list.

On exiting this loop, free value_buf, and return plist_id.

On error, test to see if the new property list has been created, and if so, discard it via a call to H5I_dec_ref().

After some sanity checking (which includes specifically disallowing property lists derived from user created property list classes) **H5P__new_plist_of_type()** runs a switch statement to map the supplied instance of H5P_plist_type_t to the id of the associated property list class. These ids are read from the appropriate global variable, and stored in the local variable class_id.

H5P__new_plist_of_type() then calls H5I_object() to obtain a pointer (pclass) to the source data set class, and then calls

```
ret_value = H5P_create_id(pclass, TRUE)
```

which creates the desired property list class, inserts it into the index, and returns the new id, which H5P__rew_plist_of_type() returns.

H5P_create_id() is discussed at length In H5Pcreate() above – please see the discussion of that API call for details. Briefly, it creates a new property list of the supplied property list class, populates it with default values, inserts it in the index, and returns the id of the new property list class.

H5P__find_prop_plist() first searches the deleted list (prop→del) for the supplied name, and flags an error and returns if the search is successful.

Failing that, it searches plist→props for a property of the supplied name, and returns a pointer to the target instance of H5P_genprop_t if the search succeeds.

Failing that, it searches the property lists of its parent property list class(es), starting with plist→parent→props, and then up the list of parents until it either finds a property of the supplied name – in which case it returns a pointer to the target instance of H5P_genprop_t, or it runs out of parents – in which case it returns NULL and flags an error.

All searches are done via calls to H5SL_search()

H5P_poke() allocates an instance of H5P_prop_set_ud_t (definition below)

```
/* Typedef for property list set/poke callbacks */
typedef struct {
   const void *value; /* Pointer to value to set */
} H5P_prop_set_ud_t;
```

and initializes the instance (udata) as follows:

```
udata.value = value;
```

This done, it calls

```
H5P__do_prop(plist, name, H5P__poke_plist_cb, H5P__poke_pclass_cb, &udata)
```

and returns.

H5P__do_prop(plist, name, H5P__poke_plist_cb, H5P__poke_pclass_cb, &udata) first searches plist→del for the specified name, and fails if it is found.

It then tries to find the named property in dst_plist→props via the call

```
prop = H5SL_search(dst_plist→props, name).
```

If successful, it calls

```
H5P__poke_plist_cb(plist, name, prop, udata)
```

and returns success or failure if this call succeeds or fails.

If this search fails, H5P__do_prop() searches the parent property list class(es) for the target property, and if successful calls:

```
H5P__poke_pclass_cb(dst_plist, name, prop, udata)
```

and again returns success or failure if this call succeeds or fails.

If the searches of the supplied property list and its parent property list class(es) fail, H5P__do_prop() returns failure.

H5P__poke_plist_cb() simply memcpy's the supplied buffer (udata→value) into prop→value and returns.

After some sanity checks, **H5P__poke_pclass_cb()** calls

```
pcopy = H5P__dup_prop(prop, H5P_PROP_WITHIN_LIST)
```

to duplicate the named property, memcpy()s udata.value into pcopy→value, and then calls

```
H5P__add_prop(plist->props, pcopy)
```

to insert the modified property into plist before returning.

In this context, **H5P__dup_prop()** allocates a new instance of H5P_genprop_t (pcopy), copies the image of *prop into it, sets pcopy→shared_name = TRUE, duplicates the buffer pointed to by prop→value (if it exists) storing the address of the duplicate in pcopy→value, and returns the address of the new instance of H5P_genprop_t.

H5P__add_prop() simply calls H5SL_insert() to add pcopy to plist→props.

Multi-thread safety concerns:

```
/**
* \ingroup GPLO
*
* \brief Encodes the property values in a property list into a binary
*        buffer
*
* \plist_id
* \param[out] buf    Buffer into which the property list will be encoded.
*                    If the provided buffer is NULL, the size of the
*                    buffer required is returned through \p nalloc; the
*                    function does nothing more.
* \param[out] nalloc The size of the required buffer
* \fapl_id
*
* \return \herr_t
*
* \details H5Pencode2() encodes the property list \p plist_id into the
*          binary buffer \p buf, according to the file format setting
*          specified by the file access property list \p fapl_id.
*
*          If the required buffer size is unknown, \p buf can be passed
*          in as NULL and the function will set the required buffer size
*          in \p nalloc. The buffer can then be created and the property
*          list encoded with a subsequent H5Pencode2() call.
*
*          If the buffer passed in is not big enough to hold the encoded
*          properties, the H5Pencode2() call can be expected to fail with
*          a segmentation fault.
*
*          The file access property list \p fapl_id is used to
*          control the encoding via the \a libver_bounds property
*          (see H5Pset_libver_bounds()). If the \a libver_bounds
*          property is missing, H5Pencode2() proceeds as if the \a
*          libver_bounds property were set to (#H5F_LIBVER_EARLIEST,
*          #H5F_LIBVER_LATEST). (Functionally, H5Pencode1() is identical to
*          H5Pencode2() with \a libver_bounds set to (#H5F_LIBVER_EARLIEST,
*          #H5F_LIBVER_LATEST).)
*          Properties that do not have encode callbacks will be skipped.
*          There is currently no mechanism to register an encode callback for
*          a user-defined property, so user-defined properties cannot
*          currently be encoded.
*
*          Some properties cannot be encoded, particularly properties that
*          are reliant on local context.
*
*      \b Motivation:
*       This function was introduced in HDF5-1.12 as part of the \a H5Sencode
*       format change to enable 64-bit selection encodings and a dataspace
*       selection that is tied to a file.
*
* \since 1.12.0
*
*/
H5_DLL herr_t H5Pencode2(hid_t plist_id, void *buf, size_t *nalloc, hid_t fapl_id);

H5Pencode2()
 +-H5I_object_verify()
 |  +- …
 +-H5CX_set_apl()
 |  +-  //
 +-H5P__encode()
    +-H5P__iterate_plist(plist, enc_all_prop, &idx, H5P__encode_cb, &udata)
```

```
+-H5SL_create()
|  +- …
|
| // Note different callback functions in two invocations
| // of H5SL_iterate().
|
+-H5SL_iterate(plist->props, H5P__iterate_plist_cb, &udata_int)
|   |
|   | // in this case - note that arg names have been changed
|   | // for clarity
|  +-H5P__iterate_plist_cb(prop, name, udata_int)
|     +-H5P__cmp_plist_cb(id, prop→name, udata)
|        +-H5P_exist_plist(udata->plist2, prop->name)
|        |  +-H5SL_search()
|        |     +- …
|        +-H5P__find_prop_plist(udata->plist2, prop->name)
|        |  +-H5SL_search()
|        |     +- …
|        +-H5P__encode_cb(prop, prop2)
|           +-HDstrlen()
|           +-HDstrcpy()
|           +-prop->encode(prop->value, udata->pp, &prop_value_len)
|              +- ??? // property specific
|
+-H5SL_iterate(plist->props, H5P__iterate_plist_pclass_cb, &udata_int)
|   |
|   | // in this case - note that arg names have been changed
|   | // for clarity
|  +-H5P__iterate_plist_pclass_cb(prop, name, udata_int)
|     +_H5SL_search()
|     |  +- …
|     +-H5P__iterate_plist_cb(prop, name, udata_int)
|        +-H5P__encode_cb(id, prop→name, udata)
|           +- // see above
|
+-H5SL_close()
   +- …
```

In a nutshell:

Encode the indicated property list in the supplied buffer, and return the number of bytes written to buf in *nalloc. If buf is NULL, *nalloc is set to the number of bytes required in buf.

In greater detail:

H5Pencode2() calls H5I_object_verify() to obtain a pointer (plist) to the target property list, and then calls

```
H5CX_set_apl(&fapl_id, H5P_CLS_FACC, H5I_INVALID_HID, TRUE)
```

to setup the context – in particular to make the FAPL available to the next call. This done, H5Pencode2() calls

```
H5P__encode(plist, TRUE, buf, nalloc)
```

and returns the result.

Note: there appears to be no mechanism to use *nalloc to detect or prevent buffer overruns when buf is not NULL.

After some sanity checking, **H5P__encode()** tests to see if the buf parameter is NULL. If it is, it sets the local encode variable to FALSE. Otherwise, encode is TRUE. Similarly, it sets the local variable p equal to buf. p is used to point to the next location to write in buf.

If encode is TRUE, H5P__encode sets the first two bytes of buf equal to H5P_ENCODE_VERS and the type of the property list to be encoded, and updates p according. Regardless of the value of encode, the local variable encode_size (used to accumulate the number of bytes written / required in the buffer) is set to 2

It then initializes udata, an instance of H5P_enc_iter_ud_t (definition below)

```
/* Typedef for iterator when encoding a property list */
typedef struct {
   hbool_t encode;        /* Whether the property list should be encoded */
   size_t *enc_size_ptr;  /* Pointer to size of encoded buffer */
                          // Note: the above comment is inaccurate
                          // in this case.  *enc_size_ptr is used to
                          // accumulate the number of bytes that are
                          // (or would be) written to the buffer.
   void ** pp;            /* Pointer to encoding buffer pointer */
} H5P_enc_iter_ud_t;
```

as follows

```
/* Initialize user data for iteration callback */
udata.encode       = encode;
udata.enc_size_ptr = &encode_size;
udata.pp           = (void **)&p;
```

H5P__encode() next sets the local variable idx equal to zero and calls

```
H5P__iterate_plist(plist, TRUE, &idx, H5P__encode_cb, &udata)
```

to encode the properties in plist, After H5P__iterate_plist() returns, it sets the last byte in the buffer to zero. As mentioned above, the number of bytes either written to the buffer (if encode is TRUE) or that would be written (if encode is FALSE) is maintained in the local variable encode_size. Just before H5P__encode() returns, it sets *nalloc = encode_size.

H5P__iterate_plist() first creates the “seen” skip list that is used to track the names of properties that have already been seen in the scan of the properties.

It then initializes udata_int, which is an instance of H5P_iter_plist_ud_t (definition below)

```
/* Typedef for property list iterator callback */
typedef struct {
   H5P_iterate_int_t     cb_func;      /* Iterator callback */
   void *                udata;        /* Iterator callback pointer */
   const H5P_genplist_t *plist;        /* Property list pointer */
   H5SL_t *              seen;         /* Skip list to hold names of
                                          properties already seen */
   int *                 curr_idx_ptr; /* Pointer to current iteration
                                          index */
   int                   prev_idx;     /* Previous iteration index */
} H5P_iter_plist_ud_t;
```

as follows:

```
/* Set up iterator callback info */
udata_int.plist        = plist;
udata_int.cb_func      = cb_func; // H5P__encode_cb() in this case
udata_int.udata        = udata;   // the udata initialized by H5P__encode()
udata_int.seen         = seen;
udata_int.curr_idx_ptr = &curr_idx; // curr_idx is a local integer
                                    // that is initialized to zero
udata_int.prev_idx     = *idx;      // *idx is zero in this case
```

This done, H5P__iterate_plist() calls

```
H5SL_iterate(plist->props, H5P__iterate_plist_cb, &udata_int)
```

After this call returns, H5P__iterate_plist() tests to see if the iter_all_prop parameter is TRUE (which it is in this case). If it is, the function scans the property list(s) of the parent property list class(es) staring with plist→parent->props via the calls

```
H5SL_iterate(tclass->props, H5P__iterate_plist_pclass_cb, &udata_int)
```

where tclass is the parent property list class currently under scan. Note that it breaks out of this scan of the parent property list class(es) if an error is return by H5SL_iterate().

Before returning, H5P__iterate_plist() sets *idx equal to *(udata_int.curr_idx) and frees the "seen" skip list.

H5SL_iterate() simply walks the skip list, calling the supplied call back function on the contents of each node until it either reaches the end of the list, or the supplied callback returns a non-zero value. In this case, it calls either

```
H5P__iterate_plist_cb(prop, name, udata_int)
```

or

```
H5P__iterate_plist_pclass_cb(prop, name, udata_int)
```

depending on which invocation in H5P__iterate_plist() we are looking at. Here, prop is a pointer to the instance of H5P_getprop_t, and name is the name of the property pointed to by prop.

After some sanity checks, **H5P__iterate_plist_cb()** tests to see if

```
*(udata_int→curr_idx_ptr) >= udata_int→prev_idx
```

if it is, H5P__iterate_plist_cb() calls:

```
ret_value = (*udata_int->cb_func)(prop, udata_int→udata)
```

or, in this case

```
ret_value =  H5P__encode_cb(prop, udata_int→udata)
```

If the above test is false, or if ret_value is non-zero, H5P__iterate_plist_cb() increments *(udata_int→cur_idx_ptr) and adds name to the "seen" skip list (udata_int→seen) prior to returning.

In reading the above, recall that udata→prev_idx is zero in this case, thus the effect is to call H5P__encode_cb() on each property in the target property list.

Finally, note that udata_int→udata is (in this case) the instance of H5P_enc_iter_ud_t allocated on the stack of H5P__encode() and initilized by that function.

If prop→encode is NULL, **H5P__encode_cb()** is a NO-OP.

Otherwise, H5P__encode_cb() calls strlen() to determine the length of the name of the supplied property. If udata→encode is TRUE, the function calls strcpy() to copy the property name into the buffer starting at udata→pp, and updates *(udata→pp) to point to the next available byte in the buffer. Regardless of the value of udata→encode, it also adds the length of property name to *(udata→enc_size_ptr).

Also regardless of the value of udata→encode, H5P__encode_cb() sets the local variable prop_value_len = 0, calls

(prop->encode)(prop->value, udata->pp, &prop_value_len)

adds prop_value_len to *(udata→enc_size_ptr) and returns.

Note: This unconditional call to prop→encode() implies that encode functions will not write to a NULL, and will update *(udata→pp) if it is not NULL Further, the decode code makes the assumption that value length is prop→size. When this is not the case, it seems that the encode and decode functions must conceal this.

Backing up a bit, **H5P__iterate_plist_pclass_cb()** first checks to see if the supplied name is in either the "seen" skip list (udata_int→seen) or the properties deleted skip list (prop→del). If it isn't, H5P__iterate_plist_pclass_cb() calls

```
ret_value = H5P__iterate_plist_cb(prop, name, udata_int)
```

and returns ret_value. See above for a discussion of H5P__iterate_plist_cb.

Multi-thread safety concerns:

```
/**
* \ingroup GPLOA
*
* \brief Compares two property lists or classes for equality
*
* \param[in] id1 First property object to be compared
* \param[in] id2 Second property object to be compared
*
* \return \htri_t
*
* \details H5Pequal() compares two property lists or classes to determine
*          whether they are equal to one another.
*
*          Either both \p id1 and \p id2 must be property lists or both
*          must be classes; comparing a list to a class is an error.
*
* \since 1.4.0
*
*/
H5_DLL htri_t H5Pequal(hid_t id1, hid_t id2);

H5Pequal()
 +-H5I_get_type()
 |  +- …
 +-H5I_object()
 |  +- …
 |
 | // parameter names have been changed for clarity
 +-H5P__cmp_plist(plist1, plist2, &cmp_ret)  // if plists
 |  +-H5P__iterate_plist(plist1, TRUE, &idx, H5P__cmp_plist_cb, &udata)
 |  |  +-H5SL_create()
 |  |  |  +- …
 |  |  |
 |  |  | // Note different callback functions in two invocations
 |  |  | // of H5SL_iterate().
 |  |  |
 |  |  +-H5SL_iterate(plist->props, H5P__iterate_plist_cb, &udata_int)
 |  |  |  |
 |  |  |  | // in this case - note that arg names have been changed for clarity
 |  |  |  +-H5P__iterate_plist_cb(prop, name, udata_int)
 |  |  |     +-H5P__cmp_plist_cb(id, prop→name, udata)
 |  |  |        +-H5P_exist_plist(udata->plist2, prop->name)
 |  |  |        |  +-H5SL_search()
 |  |  |        |     +- …
 |  |  |        +-H5P__find_prop_plist(udata->plist2, prop->name)
 |  |  |        |  +-H5SL_search()
 |  |  |        |     +- …
 |  |  |        +-H5P__cmp_prop(prop, prop2)
 |  |  |           +-prop->cmp(prop->value, prop2->value, prop→size)
 |  |  |              +- ??? // property specific
 |  |  |
 |  |  +-H5SL_iterate(plist->props, H5P__iterate_plist_pclass_cb, &udata_int)
 |  |  |  |
 |  |  |  | // in this case - note that arg names have been changed for clarity
 |  |  |  +-H5P__iterate_plist_pclass_cb(prop, name, udata_int)
 |  |  |     +_H5SL_search()
 |  |  |     |  +- …
 |  |  |     +-H5P__iterate_plist_cb(prop, name, udata_int)
 |  |  |        +-H5P__cmp_plist_cb(id, prop→name, udata)
 |  |  |           +- // see above
 |  |  |
```

```
 |  |  +-H5SL_close()
 |  |     +- …
 |  +-H5P__cmp_class(plist1->pclass, plist2->pclass)
 |     +-strcmp()
 |     +-H5SL_first()
 |     |  +- …
 |     +-H5SL_item()
 |     |  +- …
 |     +-H5P__cmp_prop()
 |     |  +- // see above
 |     +-H5SL_next()
 |        +- …
 |
 | // parameter names have been changed for clarity
 +-H5P__cmp_class(pclass1, pclass2)
    +- // see above
```

In a nutshell:

Compare two property list or two property list classes and return TRUE if they are identical, and FALSE otherwise.

In greater detail:

H5Pequal() first verifies that either both the supplied ids refer to property list classes, or both refer to property lists. It flags an error and returns if this is not the case.

It then calls H5I_object() to obtain pointers to both property lists or property list classes, and in passing verifies that both exist.

If both of the supplied ids refer to property lists, it calls

```
H5P__cmp_plist(plist1, plist2, &cmp_ret)
```

sets ret_value to TRUE *cpm_ret = 0, and FALSE otherwise and then returns.

Alternatively, if both of the supplied ids refer to property list classes, H5Pequal() calls

```
H5P__cmp_class(pclass1, pclass2)
```

and then returns TRUE if that function returns 0, and FALSE otherwise.

PROPERTY LIST CASE:

H5P__cmp_plist() first checks to see if plist1→nprops is either greater than or less than plist2→nprops, sets *cmp_ret to either +1 or -1 if this is the case and returns SUCCEED.

If the nprops fields are equal, it checks to see if plist1→class_init is either greater than or less than pclist2→class_init, again setting *cpm_ret to either +1 or -1 and returning SUCCEED if this is the case.

If neither of these tests demonstrate inequality, H5P__cmp_plist initializes udata, an instance of H5P_plist_cmp_ud_t (definition below)

```
/* Typedef for property list comparison callback */
typedef struct {
   const H5P_genplist_t *plist2;    /* Pointer to second property list */
   int                  cmp_value; /* Value from property comparison */
} H5P_plist_cmp_ud_t;
```

as follows:

```
/* Set up iterator callback info */
udata.cmp_value = 0;
udata.plist2    = plist2;
```

This done, H5P__cmp_plist() calls

```
ret_value = H5P__iterate_plist(plist1, TRUE, &idx,
                               H5P__cmp_plist_cb, &udata)
```

Here, idx is a local integer that is initialized to zero.

If ret_value is negative, H5P__cmp_plist() flags an error and returns. If ret_value is positive, it sets *cm_ret = udata.cmp_value. Otherwise, it calls

```
*cmp_ret = H5P__cmp_class(plist1->pclass, plist2->pclass)
```

and returns SUCCEED if cmp_ret is non-zero.

Finally, if none of these tests have demonstrated inequality, it sets *cmp_ret = 0 and returns SUCCEED.

H5P__iterate_plist() is discussed in detail in the description of H5Piterate() below. Thus, only a brief outline that highlights the differences is shown here, and the reader is directed to H5Piterate() for further details.

The only significant difference between the calls to H5P__iterate_plist() here and in H5Piterate() are the call back and udata parameters, which are reflected in the initialization of udata_int as shown below:

```
/* Set up iterator callback info */
udata_int.plist        = plist1;
```

```
udata_int.cb_func      = cb_func;   // H5P__cmp_plist_cb() in this case
udata_int.udata        = udata;     // the udata initialized by
                                    // H5P__cmp_plist()
udata_int.seen         = seen;
udata_int.curr_idx_ptr = &curr_idx; // curr_idx is a local integer
                                    // that is initialized to zero
udata_int.prev_idx     = *idx;
```

While these differences have no effect on the processing of H5P__iterate_plist() proper, they become key further down the calling tree.

This initialization done, H5P__iterate_plist calls:

```
ret_value = H5SL_iterate(plist1->props, H5P__iterate_plist_cb, &udata_int)
```

to scan the plist1→props, and, if H5SL_iterate() returns zero (indicating no differences found in this case) and if the iter_all_props parameter is TRUE (which it is in this case) it goes on to scan the property lists of the parent property list classes via (possibly) repeated calls to:

```
ret_value = H5SL_iterate(tclass->props, H5P__iterate_plist_pclass_cb,
                         &udata_int)
```

where tclass is the parent property list class currently being scanned. P5P__iterate_plist() breaks out of the scan and returns if ret_value is non-zero – which indicates (in this case) that either an error or a difference has been found.

H5SL_iterate() simply walks the skip list, calling the supplied call back function on the contents of each node until it either reaches the end of the list, or the supplied callback returns a non-zero value. In this case, it calls either

```
H5P__iterate_plist_cb(prop, name, udata_int)
```

or

```
H5P__iterate_plist_pclass_cb(prop, name, udata_int)
```

depending on which invocation in H5P__iterate_plist() we are looking at. Here, prop is a pointer to the instance of H5P_getprop_t, and name is the name of the property pointed to by prop.

H5P__iterate_plist_cb() is also discussed under H5Piterate(), and the reader is directed there for a detailed discussion. In this context, H5P__iterate_plist_cb() simply calls:

```
ret_value = (*udata_int->cb_func)(prop, udata_int→udata)
```

or, in this case

```
ret_value =  H5P__cmp_plist_cb(prop, udata_int→udata)
```

and returns ret_value. Other than maintaining the “seen” skip list, the remaining processing of the function is irrelevant to property list comparison.

H5P__cmp_plist_cb() starts by calling

```
prop2_exist = H5P_exist_plist(udata→plist2, prop→name)
```

to see whether plist2 contains a property of the same name as the target property.

If prop2_exist is TRUE, H5P__cmp_plist_cb() calls

```
prop2 = H5P__find_prop_plist(udata->plist2, prop->name)
```

to obtain a pointer to the target property in plist2, and then calls

```
udata->cmp_value = H5P__cmp_prop(prop, prop2)
```

to compare the two properties. If udata→cmp_value is not zero, H5P__cmp_plist_cb() returns H5_ITER_STOP.

If prop2_exist is FALSE, H5P__cmp_plist_cb() sets udata→cmp_value to 1 and again returns H5_ITER_STOP.

H5P_exist_plist() first searches for the supplied property name in the deleted list (plist→del), and returns FALSE if this search is successful.

It then searches plist→props, returning TRUE if this search succeeds.

If the search of plist→props fails, it searches the parent property list class(es), starting with plist→pclass→props and works its way up until either the search succeeds – in which case it returns TRUE – or it runs out of parent property list classes – in which case it returns FALSE.

All searches are done via calls to H5SL_search().

H5P__find_prop_plist() is essentially the same as H5P_exist_plist(), save that it returns a pointer to the target property on success, and flags an error if either the target property has been deleted or if the search fails.

H5P__cmp_prop() does a field by field comparison of the two instances of H5P_genprop_t pointed to by the prop1 and prop2 parameters. The names are compared via strcmp() and the values via the cmp call back. The function returns zero if all fields are identical, and either +1 or -1 if a difference is detected.

Backing up a bit, **H5P__iterate_plist_pclass_cb()** first checks to see if the supplied name is in either the "seen" skip list (udata_int→seen) or in the properties deleted skip list (prop→del). If it isn't, H5P__iterate_plist_pclass_cb() calls

```
ret_value = H5P__iterate_plist_cb(prop, name, udata_int)
```

and returns ret_value. See above for a discussion of H5P__iterate_plist_cb.

Backing up even further, **H5P__cmp_class()**, compares the supplied instances of H5P_genclass_t. If the revision fields match, the function assumes that the rest of the structures are identical, and returns zero.

Otherwise, H5P__cmp_class() does a field by field comparison of the instances of H5P_genclass_t (Note that parent fields are not compared). Names are compared via strcmp(). The contents of the property lists are compared by stepping through both property lists entry by entry, and calling H5P__cmp_prop() to compare the properties. Since skip lists sort their entries, this test appears to be correct.

H5P__cmp_class() returns zero if no differences are found, and either +1 or -1 otherwise.

PROPERTY LIST CLASS CASE:

The property list class case is handled by a call to H5P__cmp_class() – see above.

```
/**
* \ingroup GPLOA
*
* \brief Queries whether a property name exists in a property list or
*        class
*
* \param[in] plist_id   Identifier for the property list or class to query
* \param[in] name       Name of property to check for
*
* \return \htri_t
*
* \details  H5Pexist() determines whether a property exists within a
*           property list or class.
*
* \since 1.4.0
*
*/
H5_DLL htri_t H5Pexist(hid_t plist_id, const char *name);

H5Pexist(id, name)
 +-H5I_get_type()
 |  +- …
 +-H5I_object()
 |  +- …
 +-H5P_exist_plist()
 |  +-H5SL_search()
 |     +- …
 +-H5P__exist_pclass()
   +-H5SL_search()
       +- …
```

In a nutshell:

Search for a property of the specified name in the property list or property list class associated with the supplied ID. Return TRUE if such a property exist, and FALSE otherwise.

In greater detail:

H5Pexists() first calls H5I_get_type() to verify that the supplied id refers to either a property list or a property list class. It then calls H5I_object() to get a pointer to the property list or class.

If it is property list, it calls H5P_exist_plist(plist, name) and returns the result.

If it is a property list class, it calls H5P__exist_pclass(pclass, name) and again returns the result.

H5P_exist_plist(plist, name), first searches plist→del for the target name and returns FALSE if the search succeeds.

Failing that, it searches plist→props for a property of the supplied name, and returns TRUE if the seach succeeds.

Failing that, it searches the property list classes of its parent property list class(es), starting with plist→parent→props, and then works its way up the list of parents until it either finds a property of the supplied name – in which case it returns TRUE, or it runs out of parents – in which case it returns FALSE.

All searches are done via calls to H5SL_search()

H5P__exist_pclass(pclass_name) first searches pclass→props for a property of the supplied name, and returns TRUE if the search succeeds.

Failing that, it searches the property lists of its parent property list class(es), starting with pclass→parent→props, and then works its way up the list of parents until it either finds a property of the supplied name – in which case it returns TRUE, or it runs out of parents – in which case it returns FALSE.

Again, all searches are done via calls to H5SL_search().

Thread safety concerns:

```
/**
* \ingroup GPLOA
*
* \brief Queries the value of a property
*
* \plist_id
* \param[in]  name  Name of property to query
* \param[out] value Pointer to a location to which to copy the value of
*                   the property
*
* \return \herr_t
*
* \details H5Pget() retrieves a copy of the value for a property in a
*          property list. If there is a \p get callback routine registered
*          for this property, the copy of the value of the property will
*          first be passed to that routine and any changes to the copy of
*          the value will be used when returning the property value from
*          this routine.
*
*          This routine may be called for zero-sized properties with the
*          \p value set to NULL. The \p get routine will be called with
*          a NULL value if the callback exists.
*
*          The property name must exist or this routine will fail.
*
*          If the \p get callback routine returns an error, \ value will
*          not be modified.
*
* \since 1.4.0
*
*/
H5_DLL herr_t H5Pget(hid_t plist_id, const char *name, void *value);

H5Pget()
 +-H5I_object_verify()
 |  +- …
 +-H5P_get()
    +-H5P__do_prop(plist, name, H5P__get_cb, H5P__get_cb, &udata)
       +-H5SL_search()
       |  +- …
       |  // In this case, the same callback is provided for both the
       |  // plist_op and pclass_op parameters – hence simplifying the
       |  // call tree in this case.
       +-H5P__get_cb(plist, name, prop, udata)
          +-H5MM_malloc()
          +-H5MM_memcpy()
          +-(*(prop->get))(plist->plist_id, name, prop->size, tmp_value)
             +- …
```

In a nutshell:

Lookup the named property in the specified property list and copy its value into the supplied buffer.

In greater detail:

H5Pget() validates input, calls H5I_object_verify() to obtain a pointer (plist) to the supplied property, calls

```
H5P_get(plist, name, value)
```

and returns success or failure depending the result.

H5P_get() is essentially a pass through. It sets udata.value = value (here, udata is an instance of H5P_prop_get_ud_t), calls

```
H5P__do_prop(plist, name, H5P__get_cb, H5P__get_cb, &udata)
```

and returns success or failure depending on its result.

In this context, **H5P__do_prop()** first searches plist→del to see if the target property has already been deleted – and flags an error if it has been.

It then tries to find the named property, first in plist→props via the call

```
prop = H5SL_search(dst_plist→props, name),
```

and if that is unsuccessful, via similar calls on the props fields of the parent property list class(es) starting with plist→parent, and working its way up.

If H5P__do_prop() is unable to find the target property, the function returns failure.

If it succeeds, it calls

```
H5P__get_cb(dst_plist, name, prop, udata)
```

and returns success or failure depending on whether this call succeeds or fails.

Conceptually, **H5P__get_cb()** copies the value of the target property into the supplied buffer *(udata→value). If the property has a get callback, it calls it on this buffer before returning it to the caller.

The actual implementation is a bit more complex, as outlined below.

H5P__get_cb() first verifies that prop→size > 0, and fails if this is not the case.

Note that this disagrees with the user documentation that states "This routine may be called for zero-sized properties with the value set to NULL. The get routine will be called with a NULL value if the callback exists." As H5Pget() will also fail if the value parameter is NULL, this appears to be a documentation bug (at best).

H5P__get_cb() then checks to see if prop→get is NULL.

If it is, H5P__get_cb() simply calls

```
H5MM_memcpy(udata->value, prop->value, prop->size)
```

and returns. If the get callback is defined, H5P__get_cb() duplicates prop→value and saves the duplicate's address in tmp_value. It then calls

```
(*(prop->get))(plist->plist_id, name, prop->size, tmp_value)
```

on the duplicate, copies *tmp_value into *value via

```
H5MM_memcpy(udata->value, tmp_value, prop->size)
```

discards *tmp_value, and returns. The comments suggest that this done to avoid corrupting prop→value if prop→get fails.

Multi-thread safety concerns:

```
/**
*\ingroup GPLO
*
* \brief Returns the property list class identifier for a property list
*
* \plist_id
*
* \return \hid_t{property list class}
*
* \details H5Pget_class() returns the property list class identifier for
*          the property list identified by the \p plist_id parameter.
*
*          Note that H5Pget_class() returns a value of #hid_t type, an
*          internal HDF5 identifier, rather than directly returning a
*          property list class. That identifier can then be used with
*          either H5Pequal() or H5Pget_class_name() to determine which
*          predefined HDF5 property list class H5Pget_class() has returned.
*
*          A full list of valid predefined property list classes appears
*          in the description of H5Pcreate().
*
*          Determining the HDF5 property list class name with H5Pequal()
*          requires a series of H5Pequal() calls in an if-else sequence.
*          An iterative sequence of H5Pequal() calls can compare the
*          identifier returned by H5Pget_class() to members of the list of
*          valid property list class names. A pseudo-code snippet might
*          read as follows:
*
*          \code
*          plist_class_id = H5Pget_class (dsetA_plist);
*
*          if H5Pequal (plist_class_id, H5P_OBJECT_CREATE) = TRUE;
*              [ H5P_OBJECT_CREATE is the property list class    ]
*              [ returned by H5Pget_class.                       ]
*
*          else if H5Pequal (plist_class_id, H5P_DATASET_CREATE) = TRUE;
*              [ H5P_DATASET_CREATE is the property list class.  ]
*
*          else if H5Pequal (plist_class_id, H5P_DATASET_XFER) = TRUE;
*              [ H5P_DATASET_XFER is the property list class.    ]
*
*          .
*          .   [ Continuing the iteration until a match is found. ]
*          .
*          \endcode
*
*          H5Pget_class_name() returns the property list class name directly
*          as a string:
*
*          \code
*          plist_class_id = H5Pget_class (dsetA_plist);
*          plist_class_name = H5Pget_class_name (plist_class_id)
*          \endcode
*
*          Note that frequent use of H5Pget_class_name() can become a
*          performance problem in a high-performance environment. The
*          H5Pequal() approach is generally much faster.
*
* \version 1.6.0 Return type changed in this release.
* \since 1.0.0
*
```

```
*/
H5_DLL hid_t H5Pget_class(hid_t plist_id);

H5Pget_class(plist_id)
 +-H5I_object_verify()
 |   +- …
 +-H5P_get_class(plist)
 |   +-
 +-H5P__access_class(pclass, H5P_MOD_INC_REF)
 |   +-H5MM_xfree()
 |   +-H5SL_destroy()
 |   |   +- …
 |   +-H5P__access_class(par_class, H5P_MOD_DEC_CLS)
 |      +- // see above
 +-H5I_register(H5I_GENPROP_CLS, pclass, TRUE)
 |   +-
 +-H5P__close_class() // error cleanup
     +-H5P__access_class(pclass, H5P_MOD_DEC_REF)
        +- // see above
```

In a nutshell:

Return an id that maps to the property list class from which the supplied property list was derived.

In greater detail:

H5Pget_class() first calls H5I_object_verify to obtain a pointer to the target property list (plist). It then calls H5P_get_class() to obtain a pointer to the target property list's parent property list class (pclass = plist→pclass).

This done, H5Pget_class() calls **H5P__access_class(pclass, H5P_MOD_INC_REF)**. This has the effect of setting pclass→deleted back to FALSE if it was TRUE, and incrementing pclass→ref_count.

H5Pget_class() then calls H5I_register() to insert the property list class into the index.

Note that can result in a given property list class having multiple ids in the index. Further, if the parent property list class has been modified since plist was created, this will cause the previous version of the property list class to be inserted into the index – resulting in multiple property list classes of the same name but different structure.

This behavior is not discussed in the user documentation.

If H5I_register() fails, H5Pget_class() calls H5P__close_class(pclass) to undo the prior call to H5P__access_class().

Thread safety concerns:

```
/**
 * \ingroup GPLOA
 *
 * \brief Retrieves the name of a class
 *
 * \plistcls_id{pclass_id}
 *
 * \return Returns a pointer to an allocated string containing the class
 *         name if successful, and NULL if not successful.
 *
 * \details H5Pget_class_name() retrieves the name of a generic property
 *          list class. The pointer to the name must be freed by the user
 *          with a call to H5free_memory() after each successful call.
 *
 *          <table>
 *           <tr>
 *            <th>Class Name (class identifier) Returned</th>
 *            <th>Property List Class</th>
 *            <th>Expanded Name of the Property List Class</th>
 *            <th>The Class Identifier Used with H5Pcreate</th>
 *            <th>Comments</th>
 *           </tr>
 *           <tr>
 *            <td>attribute create</td>
 *            <td>acpl</td>
 *            <td>Attribute Creation Property List</td>
 *            <td>H5P_ATTRIBUTE_CREATE</td>
 *            <td> </td>
 *           </tr>
 *           <tr>
 *            <td>dataset access</td>
 *            <td>dapl</td>
 *            <td>Dataset Access Property List</td>
 *            <td>H5P_DATASET_ACCESS</td>
 *            <td> </td>
 *           </tr>
 *           <tr>
 *            <td>dataset create</td>
 *            <td>dcpl</td>
 *            <td>Dataset Creation Property List</td>
 *            <td>H5P_DATASET_CREATE</td>
 *            <td> </td>
 *           </tr>
 *           <tr>
 *            <td>data transfer</td>
 *            <td>dxpl</td>
 *            <td>Data Transfer Property List</td>
 *            <td>H5P_DATASET_XFER</td>
 *            <td> </td>
 *           </tr>
 *           <tr>
 *            <td>datatype access</td>
 *            <td> </td>
 *            <td> </td>
 *            <td>H5P_DATATYPE_ACCESS</td>
 *            <td>This class can be created, but there are no properties
 *                in the class currently.
 *            </td>
 *           </tr>
 *           <tr>
 *            <td>datatype create</td>
 *            <td> </td>
```

```
*             <td> </td>
*             <td>H5P_DATATYPE_CREATE</td>
*             <td>This class can be created, but there
*                 are no properties in the class currently.</td>
*            </tr>
*            <tr>
*             <td>file access</td>
*             <td>fapl</td>
*             <td>File Access Property List</td>
*             <td>H5P_FILE_ACCESS</td>
*             <td> </td>
*            </tr>
*            <tr>
*             <td>file create</td>
*             <td>fcpl</td>
*             <td>File Creation Property List</td>
*             <td>H5P_FILE_CREATE</td>
*             <td> </td>
*            </tr>
*            <tr>
*             <td>file mount</td>
*             <td>fmpl</td>
*             <td>File Mount Property List</td>
*             <td>H5P_FILE_MOUNT</td>
*             <td> </td>
*            </tr>
*            <tr>
*             <td>group access</td>
*             <td> </td>
*             <td> </td>
*             <td>H5P_GROUP_ACCESS</td>
*             <td>This class can be created, but there
*                 are no properties in the class currently.</td>
*            </tr>
*            <tr>
*             <td>group create</td>
*             <td>gcpl</td>
*             <td>Group Creation Property List</td>
*             <td>H5P_GROUP_CREATE</td>
*             <td> </td>
*            </tr>
*            <tr>
*              <td>link access</td>
*              <td>lapl</td>
*              <td>Link Access Property List</td>
*              <td>H5P_LINK_ACCESS</td>
*              <td> </td>
*            </tr>
*            <tr>
*             <td>link create</td>
*             <td>lcpl</td>
*             <td>Link Creation Property List</td>
*             <td>H5P_LINK_CREATE</td>
*             <td> </td>
*            </tr>
*            <tr>
*             <td>object copy</td>
*             <td>ocpypl</td>
*             <td>Object Copy Property List</td>
*             <td>H5P_OBJECT_COPY</td>
*             <td> </td>
*            </tr>
*            <tr>
```

```
*             <td>object create</td>
*             <td>ocpl</td>
*             <td>Object Creation Property List</td>
*             <td>H5P_OBJECT_CREATE</td>
*             <td> </td>
*            </tr>
*            <tr>
*             <td>string create</td>
*             <td>strcpl</td>
*             <td>String Creation Property List</td>
*             <td>H5P_STRING_CREATE</td>
*             <td> </td>
*            </tr>
*           </table>
*
* \since 1.4.0
*
*/
H5_DLL char *H5Pget_class_name(hid_t pclass_id);

H5Pget_class_name(pclass_id)
 +-H5I_object_verify()
 |   +- …
 +-H5P_get_class_name(pclass)
    +-H5MM_xstrdup(pclass->name)
```

In a nutshell:

Look up the indicated property list class, allocate a string of appropriate length, copy the class's name into the new string, and return a pointer to it.

In greater detail:

Later

Multi-thread safety concerns:

```
/**
* \ingroup GPLOA
*
* \brief Retrieves the parent class of a property class
*
* \plistcls_id{pclass_id}
*
* \return \hid_t{parent class object}
*
* \details H5Pget_class_parent() retrieves an identifier for the parent
*          class of a property class.
*
* \since 1.4.0
*
*/
H5_DLL hid_t H5Pget_class_parent(hid_t pclass_id);
```

```
H5Pget_class_parent(pclass_id)
 +-H5I_object_verify()
 |  +- …
 +-H5P_get_class_parent(pclass)
 |  +-
 +-H5P__access_class(parent, H5P_MOD_INC_REF)
 |  +-H5MM_xfree()
 |  +-H5SL_destroy()
 |  |  +- …
 |  +-H5P__access_class(par_class, H5P_MOD_DEC_CLS)
 |     +- // see above
 +-H5I_register(H5I_GENPROP_CLS, pclass, TRUE)
 |  +-
 +-H5P__close_class() // error cleanup
    +-H5P__access_class(pclass, H5P_MOD_DEC_REF)
       +- // see above
```

In a nutshell:

Return an id that maps to the property list class from which the supplied property list class was derived.

In greater detail:

H5Pget_class_parent() first calls H5I_object_verify to obtain a pointer to the target property list class (pclass). It then calls H5P_get_class_parent() to obtain a pointer to the target property list class's parent property list class (parent = pclass→parent).

This done, H5Pget_class_parent() calls H5P__access_class(parent, H5P_MOD_INC_REF). This has the effect of setting parent→deleted back to FALSE if it was TRUE, and incrementing parent→ref_count.

H5Pget_class() then calls H5I_register() to insert the parent property list class into the index.

Note that can result in a given property list class having multiple ids in the index. Further, if the parent property list class has been modified since pclass was created, this will cause the previous version of the property list class to be inserted into the index – resulting in multiple property list classes of the same name but different structure.

This behavior is not discussed in the user documentation.

If H5I_register() fails, H5Pget_classparent() calls H5P__close_class(pclass) to undo the prior call to H5P__access_class().

Thread safety concerns:

```
/**
* \ingroup GPLOA
*
* \brief  Queries the number of properties in a property list or class
*
* \param[in]  id     Identifier for property object to query
* \param[out] nprops Number of properties in object
*
* \return \herr_t
*
* \details H5Pget_nprops() retrieves the number of properties in a
*          property list or property list class.
*
*          If \p id is a property list identifier, the current number of
*          properties in the list is returned in \p nprops.
*
*          If \p id is a property list class identifier, the number of
*          registered properties in the class is returned in \p nprops.
*
* \since 1.4.0
*
*/
H5_DLL herr_t H5Pget_nprops(hid_t id, size_t *nprops);
```

```
H5Pget_nprops()
 +-H5I_get_type()
 +-H5I_object()
 |  +- …
 +-H5P__get_nprops_plist(plist, nprops)
 |
 +-H5P_get_nprops_pclass(pclass, nprops, FALSE)
```

In a nutshell:

Given the id of a property list or a property list class, return the number of properties in same. In the case of property list classes, this is just the number of properties defined in the target property list class, and does not include properties defined in its parents.

In greater detail:

After verifying that the supplied id refers to either a property list, or a property list class, and that the target property list or property list class actually exists, **H5Pget_nprops()** first calls either H5P__get_nprops_plist() or H5P_get_nprops_pclass() to obtain the number of properties in the target, and returns this value in *nprops.

H5P__get_nprops_plist() simply sets *nprops = plist→nprops and returns.

If its “recurse” parameter is FALSE (as it is in this case) **H5P_get_nprops_pclass()** also just sets *nprops = pclass→nprops. However, if “recurse” is TRUE, it also walks the list of parent of pclass, and returns the sum of the nprops fields of pclass and all of its parents.

Thread safety concerns:

```
/**
* \ingroup GPLOA
*
* \brief Queries the size of a property value in bytes
*
* \param[in]  id   Identifier of property object to query
* \param[in]  name Name of property to query
* \param[out] size Size of property in bytes
*
* \return  \herr_t
*
* \details H5Pget_size() retrieves the size of a property's value in
*          bytes. This function operates on both property lists and
*          property classes.
*
*          Zero-sized properties are allowed and return 0.
*
* \since 1.4.0
*
*/
H5_DLL herr_t H5Pget_size(hid_t id, const char *name, size_t *size);
```

```
H5Pget_size()
 +-H5I_get_type(id)
 |  +- …
 +-H5I_get_object(id)
 |  +- …
 +-H5P__get_size_plist(plist, name, size)
 |  +-H5P__find_prop_plist(plist, name)
 |     +-H5SL_search()
 |        +- …
 +-H5P__get_size_pclass(pclass, name, size)
    +-H5P__find_prop_pclass(pclass, name)
       +-H5SL_search()
          +- …
```

In a nutshell:

Look up the named property in the property list or property list class associated with the supplied id, and return the size of the value of the property in *size.

In greater detail:

After verifying that the supplied id refers to either a property list class or a property list, **H5Pget_size()** verifies that the target property list class or property list exists, and then calls either

```
H5P__get_size_plist(plist, name, size)
```

or

```
H5P__get_size_pclass(pclass, name, size)
```

to load the size of the target properties value in *size, and returns.

H5P__get_size_plist() calls

```
prop = H5P__find_prop_plist(plist, name)
```

to obtain a pointer to the target property. If this is successful, it sets *size = prop→size and returns.

H5P__find_prop_plist() first searches the deleted list (prop→del) for the supplied name, and flags an error and returns if the search is successful.

Failing that, it searches plist→props for a property of the supplied name, and returns a pointer to the target instance of H5P_genprop_t if the search succeeds.

Failing that, it searches the property lists of its parent property list class(es), starting with plist→parent→props, and then up the list of parents until it either finds a property of the supplied name – in which case it returns a pointer to the target instance of H5P_genprop_t, or it runs out of parents – in which case it returns NULL and flags an error.

All searches are done via calls to H5SL_search()

H5P__get_size_pclass() calls

```
prop = H5P__find_prop_pclass(pclass, name)
```

to obtain a pointer to the target property. If this is successful, it sets *size = prop→size and returns.

H5P__find_prop_pclass() searches its property list (pclass→props) via a call to H5SL_search() for a property of the supplied name, and returns a pointer to the target instance of H5P_genprop_t if successful. If the search fails, it flags an error and returns NULL.

Note that H5P__find_prop_pclass() does not search the property lists of its parent property list class(es) for the target property if it doesn't exist in pclass→props. This makes H5Pset_size() in-congruent with similar calls and is probably a bug.

Multi-thread safety concerns:

```
/**
* \ingroup GPLOA
*
* \brief Registers a temporary property with a property list
*
* \plist_id
* \param[in] name    Name of property to create
* \param[in] size    Size of property in bytes
* \param[in] value   Initial value for the property
* \param[in] set     Callback routine called before a new value is copied
*                    into the property's value
* \param[in] get     Callback routine called when a property value is
*                    retrieved from the property
* \param[in] prp_del Callback routine called when a property is deleted
*                    from a property list
* \param[in] copy    Callback routine called when a property is copied
*                    from an existing property list
* \param[in] compare Callback routine called when a property is compared
*                    with another property list
* \param[in] close   Callback routine called when a property list is
*                    being closed and the property value will be disposed
*                    of
*
* \return \herr_t
*
* \details H5Pinsert2() creates a new property in a property
*          list. The property will exist only in this property list and
*          copies made from it.
*
*          The initial property value must be provided in \p value and
*          the property value will be set accordingly.
*
*          The name of the property must not already exist in this list,
*          or this routine will fail.
*
*          The \p set and \p get callback routines may be set to NULL
*          if they are not needed.
*
*          Zero-sized properties are allowed and do not store any data
*          in the property list. The default value of a zero-size
*          property may be set to NULL. They may be used to indicate the
*          presence or absence of a particular piece of information.
*
*          The \p set routine is called before a new value is copied
*          into the property. The #H5P_prp_set_func_t callback function
*          is defined as follows:
*          \snippet this H5P_prp_cb2_t_snip
*
*
*          The parameters to the callback function are defined as follows:
*          <table>
*           <tr>
*            <td>\ref hid_t \c prop_id</td>
*            <td>IN: The identifier of the property list being
*                modified</td>
*           </tr>
*           <tr>
*            <td>\Code{const char * name}</td>
*            <td>IN: The name of the property being modified</td>
*           </tr>
*           <tr>
*            <td>\Code{size_t size}</td>
```

```
*            <td>IN: The size of the property in bytes</td>
*           </tr>
*           <tr>
*             <td>\Code{void * value}</td>
*             <td>IN: Pointer to new value pointer for the property
*                 being modified</td>
*           </tr>
*          </table>
*
*          The \p set routine may modify the value pointer to be set and
*          those changes will be used when setting the property's value.
*          If the \p set routine returns a negative value, the new property
*          value is not copied into the property and the \p  set routine
*          returns an error value. The \p set routine will be called for
*          the initial value.
*
*          \b Note: The \p set callback function may be useful to range
*          check the value being set for the property or may perform some
*          transformation or translation of the value set. The \p get
*          callback would then reverse the transformation or translation.
*          A single \p get or \p set callback could handle multiple
*          properties by performing different actions based on the
*          property name or other properties in the property list.
*
*          The \p get routine is called when a value is retrieved from
*          a property value. The #H5P_prp_get_func_t callback function
*          is defined as follows:
*
*          \snippet this H5P_prp_cb2_t_snip
*
*          The parameters to the above callback function are:
*
*          <table>
*           <tr>
*            <td>\ref hid_t \c prop_id</td>
*            <td>IN: The identifier of the property list being queried</td>
*           </tr>
*           <tr>
*            <td>\Code{const char * name}</td>
*            <td>IN: The name of the property being queried</td>
*           </tr>
*           <tr>
*            <td>\Code{size_t  size}</td>
*            <td>IN: The size of the property in bytes</td>
*           </tr>
*           <tr>
*            <td>\Code{void *  value}</td>
*            <td>IN: The value of the property being returned</td>
*           </tr>
*          </table>
*
*          The \p get routine may modify the value to be returned from
*          the query and those changes will be preserved. If the \p get
*          routine returns a negative value, the query routine returns
*          an error value.
*
*          The \p prp_del routine is called when a property is being
*          deleted from a property list. The #H5P_prp_delete_func_t
*          callback function is defined as follows:
*
*          \snippet this H5P_prp_cb2_t_snip
*
*          The parameters to the above callback function are:
```

```
*
*          <table>
*           <tr>
*            <td>\ref hid_t \c prop_id</td>
*            <td>IN: The identifier of the property list the property is
*                being deleted from</td>
*           </tr>
*           <tr>
*            <td>\Code{const char * name}</td>
*            <td>IN: The name of the property in the list</td>
*           </tr>
*           <tr>
*            <td>\Code{size_t size}</td>
*            <td>IN: The size of the property in bytes</td>
*           </tr>
*           <tr>
*            <td>\Code{void * value}</td>
*            <td>IN: The value for the property being deleted</td>
*           </tr>
*          </table>
*
*          The \p prp_del routine may modify the value passed in, but the
*          value is not used by the library when the \p prp_del routine
*          returns. If the \p prp_del routine returns a negative value,
*          the property list \p prp_del routine returns an error value but
*          the property is still deleted.
*
*          The \p copy routine is called when a new property list with
*          this property is being created through a \p copy operation.
*
*          The #H5P_prp_copy_func_t callback function is defined as follows:
*
*          \snippet this H5P_prp_cb1_t_snip
*
*          The parameters to the above callback function are:
*          <table>
*           <tr>
*            <td>\Code{const char * name}</td>
*            <td>IN: The name of the property being copied</td>
*           </tr>
*           <tr>
*            <td>\Code{size_t size}</td>
*            <td>IN: The size of the property in bytes</td>
*           </tr>
*           <tr>
*            <td>\Code{void * value}</td>
*            <td>IN/OUT: The value for the property being copied</td>
*           </tr>
*          </table>
*
*          The \p copy routine may modify the value to be set and those
*          changes will be stored as the new value of the property. If the
*          \p copy routine returns a negative value, the new property value
*          is not copied into the property and the copy routine returns an
*          error value.
*
*          The \p compare routine is called when a property list with this
*          property is compared to another property list with the same
*          property.
*
*          The #H5P_prp_compare_func_t callback function is defined as
*          follows:
*
```

```
*          \snippet this H5P_prp_compare_func_t_snip
*
*          The parameters to the callback function are defined as follows:
*
*          <table>
*           <tr>
*            <td>\Code{const void * value1}</td>
*            <td>IN: The value of the first property to compare</td>
*           </tr>
*           <tr>
*            <td>\Code{const void * value2}</td>
*            <td>IN: The value of the second property to compare</td>
*           </tr>
*           <tr>
*            <td>\Code{size_t size}</td>
*            <td>IN: The size of the property in bytes</td>
*           </tr>
*          </table>
*
*          The \p compare routine may not modify the values. The \p compare
*          routine should return a positive value if \p value1 is greater
*          than \p value2, a negative value if \p value2 is greater than
*          \p value1 and zero if \p value1 and \p value2 are equal.
*
*          The \p close routine is called when a property list with this
*          property is being closed.
*
*          The #H5P_prp_close_func_t callback function is defined as follows:
*          \snippet this H5P_prp_cb1_t_snip
*
*          The parameters to the callback function are defined as follows:
*
*          <table>
*           <tr>
*            <td>\Code{const char * name}</td>
*            <td>IN: The name of the property in the list</td>
*           </tr>
*           <tr>
*            <td>\Code{size_t size}</td>
*            <td>IN: The size of the property in bytes</td>
*           </tr>
*           <tr>
*            <td>\Code{void * value}</td>
*            <td>IN: The value for the property being closed</td>
*           </tr>
*          </table>
*
*          The \p close routine may modify the value passed in, the
*          value is not used by the library when the close routine
*          returns. If the \p close routine returns a negative value,
*          the property list \p close routine returns an error value
*          but the property list is still closed.
*
*          \b Note: There is no \p create callback routine for temporary
*          property list objects; the initial value is assumed to
*          have any necessary setup already performed on it.
*
* \since 1.8.0
*
*/
H5_DLL herr_t H5Pinsert2(hid_t plist_id, const char *name, size_t size,
                         void *value, H5P_prp_set_func_t prp_set,
                         H5P_prp_get_func_t prp_get,
```

```
                         H5P_prp_delete_func_t prp_del,
                         H5P_prp_copy_func_t prp_copy,
                         H5P_prp_compare_func_t prp_cmp,
                         H5P_prp_close_func_t prp_close);

H5Pinsert2(plist_id, name, size, value, prp_set, prp_get, prp_del, prp_copy,
 |          prp_cmp, prp_close)
 +-H5I_object_verify()
 |  +- …
 +-H5P_insert(plist, name, size, value, prp_set, prp_get, NULL, NULL,
    |          prp_delete, prp_copy, prp_cmp, prp_close)
    +-H5SL_search()
    |  +- …
    +-H5SL_remove()
    |  +- …
    +-H5MM_xfree()
    +-H5P__create_prop()
    |  +-H5FL_MALLOC()
    |  +-H5MM_xstrdup()
    |  +-H5MM_malloc()
    |  +-H5MM_memcpy()
    |  +-H5MM_xfree()  // error cleanup
    |  +-H5FL_FREE()   // error cleanup
    +-H5P__add_prop()
    +-H5P__add_prop()
    |  +-H5SL_insert()
    |     +- …
    +-H5P__free_prop() // error cleanup
        +-H5MM_xfree()
        +-H5FL_FREE()
```

In a nutshell:

Create a property as specified by the supplied parameters and insert it in the target property list.

In greater detail:

H5Pinsert2() calls H5I_object_verify() to obtain a pointer (plist) to the target property list. After some sanity checking, it then calls

```
    H5P_insert(plist, name, size, value, prp_set, prp_get, NULL,
                      NULL, prp_delete, prp_copy, prp_cmp, prp_close)
```

and returns whatever H5P_insert() returns.

H5P_insert() proceeds as follows:

- Search plist→props for name. If this search is successful, a property of the supplied name already exists. In this case, H5P_insert() flags an error and returns.

- Search the deleted list (plist→del) for the supplied name. If it is found, remove it from the deleted list.

  If the supplied name doesn't appear in the deleted list, search the parent property list class(es) for the supplied name. If it is found, a property of the supplied name already exists. In this case as well, H5P_insert() flags an error and returns.

- Create the new property via the call:

  ```
  new_prop = H5P__create_prop(name, size, H5P_PROP_WITHIN_LIST,
                              value, NULL, prp_set, prp_get,
                              prp_encode, prp_decode,
                              prp_delete. prp_copy, prp_cmp,
                              prp_close)
  ```

  After some sanity checks, **H5P__create_prop()** allocates a new instance of H5P_genprop_t (new_prop), duplicates *name and sets new_prop→name to point to it. Similarly, if value is not NULL, it duplicates *value, and sets new_prop→value to point to the copy.

  It initializes the remaining fields of *new_prop from its parameters as follows:

  ```
  new_prop→shared_name = FALSE
  new_prop→size        = size;
  new_prop→type        = H5P_PROP_WITHIN_LIST;
  new_prop→create      = NULL;
  new_prop→set         = prp_set;
  new_prop→get         = prp_get;
  new_prop→encode      = prp_encode;
  new_prop→decode      = prp_decode;
  new_prop→del         = prp_delete;
  new_prop→copy        = prp_copy;
  new_prop→cmp         = prp_cmp;
  new_prop→close       = prp_close;
  ```

  If prp_cmp is NULL, new_prop→cmp is set to &memcmp.

- Insert the new property in plist via the call

  ```
  H5P__add_prop(plist->props, new_prop)
  ```

- Increment plist→nprops.

And then return. In the event of error, H5P_insert() tests for the existence of *new_prop and discards it if found prior to return.

Multi-thread safety concerns:

```
/**
* \ingroup GPLOA
*
* \brief Determines whether a property list is a member of a class
*
* \plist_id
* \plistcls_id{pclass_id}
*
* \return \htri_t
*
* \details H5Pisa_class() checks to determine whether the property list
*          \p plist_id is a member of the property list class
*          \p pclass_id.
*
* \see H5Pcreate()
*
* \since  1.6.0
*
*/
H5_DLL htri_t H5Pisa_class(hid_t plist_id, hid_t pclass_id);

H5Pisa_class(plist_id, pclass_id)
 +-H5I_get_type()
 |  +- …
 +-H5P_isa_class(plist_id, pclass_id)
    +-H5I_object_verify()
    |  +- …
    |
    | // pclass1 == plist→pclass, pclass2 == pclass
    +-H5P_class_isa(pclass1, pclass2)
       |
       +-H5P__cmp_class(pclass1, pclass2)
       |  +-strcmp()
       |  +-H5SL_first()
       |  |  +- …
       |  +-H5SL_item()
       |  |  +- …
       |  +-H5P__cmp_prop(prop1, prop2)
       |  |  +-prop->cmp(prop1->value, prop2->value, prop1→size)
       |  |     +- ??? // property specific
       |  +-H5SL_next()
       |     +- …
       |
       +-H5P_class_isa(pclass1->parent, pclass2)
          +- // recursive call – see above
```

In a nutshell:

Determine whether the the supplied property list is a member of the supplied property list class.

Note: Here “member” appears to mean that plist_id refers to a property list that is an instance of the property list class referred to by pclass_id **OR** that the immediate parent property list class of the supplied property list is a descendant of the supplied property list class. While this is standard OOP terminology, it may be asking a bit much of our users to know this. Assuming that this interpretation is retained, this point should be made clear in the user level documentation.

In more detail:

H5Pisa_class() verifies that plist_id and pclass_id refer to a property list and a property list class respectively, calls:

```
H5P_isa_class(plist_id, pclass_id)
```

and returns whatever that function returns.

H5P_isa_class() calls H5I_object_verify() to obtain pointers (plist and pclass2) to the supplied property list and property class. To make the following discussion easier to follow, define pclass1 = plist→pclass.

Finally, H5P_isa_class() calls

```
H5P_class_isa(pclass1, pclass2)
```

and returns whatever that function returns. In what follows, keep in mind that H5P_isa_class() and H5P_class_isa() are two very different functions, and that pclass1 is the parent property list class of the supplied property list, and that pclass2 is the property list class supplied by the user.

H5P_class_isa(pclass1, pclass2) first calls

```
H5P__cmp_class(pclass1, pclass2)
```

and returns TRUE if H5P__cmp_class() returns 0. For any other return value, H5P_class_isa() sets pclass1 = pclass1→parent. If pclass1 is NULL, H5P_class_isa() returns FALSE. Otherwise, H5P_class_isa() makes the recursive call:

```
H5P_class_isa(pclass1, pclass2)
```

and returns whatever the recursive call returns. The effect of this recursive call is to compare all the property list classes from which pclass1 is derived to pclass2 and return TRUE if and only if one of these property list classes is identical to pclass2.

H5P__cmp_class(), compares the supplied instances of H5P_genclass_t. If the revision fields match, the function assumes that the rest of the structures are identical, and returns zero.

Otherwise, H5P__cmp_class() does a field by field comparison of the instances of H5P_genclass_t (Note that parent fields are not compared). Names are compared via strcmp(). The contents of the property lists are compared by stepping through both property lists entry by entry, and calling H5P__cmp_prop() to compare the properties. Since skip lists sort their entries, this test appears to be correct.

H5P__cmp_class() returns zero if no differences are found, and either +1 or -1 otherwise.

H5P__cmp_prop() does a field by field comparison of the two instances of H5P_genprop_t pointed to by the prop1 and prop2 parameters. The names are compared via strcmp() and the values via the cmp call back. The function returns zero if all fields are identical, and either +1 or -1 if a difference is detected.

Multi-thread safety concerns:

```
/**
* \ingroup GPLOA
*
* \brief Iterates over properties in a property class or list
*
* \param[in]     id  Identifier of property object to iterate over
* \param[in,out] idx Index of the property to begin with
* \param[in]     iter_func  Function pointer to function to be called
*                with each property iterated over
* \param[in,out] iter_data  Pointer to iteration data from user
*
* \return On success: the return value of the last call to \p iter_func if
*         it was non-zero; zero if all properties have been processed.
*         On Failure, a negative value
*
* \details H5Piterate() iterates over the properties in the property
*          object specified in \p id, which may be either a property
*          list or a property class, performing a specified operation
*          on each property in turn.
*
*          For each property in the object, \p iter_func and the
*          additional information specified below are passed to the
*          #H5P_iterate_t operator function.
*
*          The iteration begins with the \p idx-th property in the
*          object; the next element to be processed by the operator
*          is returned in \p idx. If \p idx is NULL, the iterator
*          starts at the first property; since no stopping point is
*          returned in this case, the iterator cannot be restarted if
*          one of the calls to its operator returns non-zero.
*
*          The operation \p iter_func receives the property list or class
*          identifier for the object being iterated over, \p id, the
*          name of the current property within the object, \p name,
*          and the pointer to the operator data passed in to H5Piterate(),
*          \p iter_data.
*
*          H5Piterate() assumes that the properties in the object
*          identified by \p id remain unchanged through the iteration.
*          If the membership changes during the iteration, the function's
*          behavior is undefined.
*
* \since 1.4.0
*
*/
H5_DLL int H5Piterate(hid_t id, int *idx, H5P_iterate_t iter_func,
                      void *iter_data);

H5Piterate(id, idx, iter_func, iter_data)
 +-H5I_get_type(id)
 |  +- …
 +-H5I_object(id)
 |  +- …
 |
 | // iter_func and iter_data stored in udata
 |
 +-H5P__iterate_plist((H5P_genplist_t *)obj, TRUE, idx, H5P__iterate_cb, &udata)
 |  +-H5SL_create()
 |  |  +- …
 |  |
 |  | // Note different callback functions in two invocations of H5SL_iterate().
 |  |
```

```
|  +-H5SL_iterate(plist->props, H5P__iterate_plist_cb, &udata_int)
|  |  |
|  |  | // in this case - note that arg names have been changed for clarity
|  |  +-H5P__iterate_plist_cb(prop, name, udata)
|  |     +-H5P__iterate_cb(prop, iter_data)
|  |     |   +-*iter_func(id, prop→name, iter_data) // user supplied function
|  |     |       +- …
|  |     +-H5SL_insert()
|  |
|  +-H5SL_iterate(plist->props, H5P__iterate_plist_pclass_cb, &udata_int)
|  |  |
|  |  | // in this case - note that arg names have been changed for clarity
|  |  +-H5P__iterate_plist_pclass_cb(prop, name, udata)
|  |     +_H5SL_search()
|  |     |  +- …
|  |     +-H5P__iterate_plist_cb(prop, name, udata)
|  |        +-H5P__iterate_cb(prop, iter_data)
|  |        |   +-*iter_func(id, prop→name, iter_data) // user supplied function
|  |        |       +- …
|  |        +-H5SL_insert()
|  |
|  +-H5SL_close()
|     +- …
|
+-H5P__iterate_pclass(pclass, idx, H5P__iterate_cb, &udata)
   +-H5SL_iterate(pclass->props, H5P__iterate_pclass_cb, &udata_int)
   |  |
   |  | // in this case - note that arg names have been changed for clarity
   |  +-H5P__iterate_pclass_cb(prop, name, udata)
   |     +-*iter_func(prop, iter_data) // user supplied function
   |         +- …
   +-H5SL_close()
```

In a nutshell:

Starting with the *idx'th property, scan the target property list or property list class and call the supplied iter_fcn with the id of the property list or property list class, the name of the property, and the supplied user data.

In greater detail:

After verifying that the supplied id exists, and references either a property list class or a property list, **H5Piterate()** initializes an instance of H5P_iter_ud_t (definition shown below):

```
typedef struct {
   H5P_iterate_t iter_func; /* Iterator callback */
   hid_t         id;        /* Property list or class ID */
   void *        iter_data; /* Iterator callback pointer */
} H5P_iter_ud_t;
```

(udata) as follows:

```
udata.iter_func = iter_func;
udata.id        = id;
udata.iter_data = iter_data;
```

With udata initialized, H5Piterate() calls either

```
H5P__iterate_plist((H5P_genplist_t *)obj, TRUE, (idx ? idx : &fake_idx),
                   H5P__iterate_cb, &udata)
```

if id refers to a property list, or

```
H5P__iterate_pclass((H5P_genclass_t *)obj, (idx ? idx : &fake_idx),
                    H5P__iterate_cb, &udata)
```

if id refers to a property list class. Note that fake_idx is an integer that is initialized to zero. It allows the function to handle a NULL idx pointer gracefully.

In either case, H5Piterate() returns whatever value is returned.

THE PROPERTY LIST CASE:

H5P__iterate_plist() first creates the "seen" skip list that is used to track the names of properties that have already been seen in the scan of the properties.
It then initializes udata_int, which is an instance of H5P_iter_plist_ud_t (definition below)

```
/* Typedef for property list iterator callback */
typedef struct {
   H5P_iterate_int_t     cb_func;      /* Iterator callback */
   void *                udata;        /* Iterator callback pointer */
   const H5P_genplist_t *plist;        /* Property list pointer */
   H5SL_t *              seen;         /* Skip list to hold names of
                                          properties already seen */
   int *                 curr_idx_ptr; /* Pointer to current iteration
                                          index */
   int                   prev_idx;     /* Previous iteration index */
} H5P_iter_plist_ud_t;
```

as follows:

```
/* Set up iterator callback info */
udata_int.plist        = plist;
udata_int.cb_func      = cb_func; // H5P__iterate_cb() in this case
udata_int.udata        = udata;   // the udata initialized by H5Piterate()
udata_int.seen         = seen;
udata_int.curr_idx_ptr = &curr_idx; // curr_idx is a local integer
                                    // that is initialized to zero
udata_int.prev_idx     = *idx;
```

This done, H5P__iterate_plist() calls

```
H5SL_iterate(plist->props, H5P__iterate_plist_cb, &udata_int)
```

After this call returns, H5P__iterate_plist() tests to see if the iter_all_prop parameter is TRUE (which it is in this case). If it is, the function scans the property list(s) of the parent property list class(es) staring with plist→parent->props via the calls

```
H5SL_iterate(tclass->props, H5P__iterate_plist_pclass_cb, &udata_int)
```

where tclass is the parent property list class currently under scan. Note that it breaks out of this scan of the parent property list class(es) if an error is return by H5SL_iterate().

Before returning, H5P__iterate_plist() sets *idx equal to *(udata_int.curr_idx) and frees the "seen" skip list.

H5SL_iterate() simply walks the skip list, calling the supplied call back function on the contents of each node until it either reaches the end of the list, or the supplied callback returns a non-zero value. In this case, it calls either

```
H5P__iterate_plist_cb(prop, name, udata_int)
```

or

```
H5P__iterate_plist_pclass_cb(prop, name, udata_int)
```

depending on which invocation in H5P__iterate_plist() we are looking at. Here, prop is a pointer to the instance of H5P_getprop_t, and name is the name of the property pointed to by prop.

After some sanity checks, **H5P__iterate_plist_cb()** tests to see if

```
*(udata_int→curr_idx_ptr) >= udata_int→prev_idx
```

if it is, H5P__iterate_plist_cb() calls:

```
ret_value = (*udata_int->cb_func)(prop, udata_int→udata)
```

or, in this case

```
ret_value =  H5P__iterate_cb(prop, udata_int→udata)
```

If the above test is false, or if ret_value is non-zero, H5P__iterate_plist_cb() increments *(udata_int→cur_idx_ptr) and adds name to the "seen" skip list (udata_int→seen) prior to returning.

In reading the above, recall that udata→prev_idx is either the starting index passed into H5Piterate() in *idx, or 0 if idx was NULL, and that udata_int→cur_idx_ptr points to a local variable in H5Piterate that was initialized to zero. Thus the initial test has the effect of skipping over the first udata_int→prev_idx items.

Further, note that udata_int→udata is (in this case) the instance of H5P_iter_ud_t allocated on the stack of H5Piterate() and initilized by that function.

H5P__iterate_cb() simply calls the user supplied function

```
ret_value = (*udata->iter_func)(udata->id, prop->name, udata->iter_data)
```

where udata→iter_func, and udata→iter_data are the iter_func and iter_data parameters passed to H5Piterate, and udata→id is both the id of the property list within which *prop resides, and also the id parameter passed to H5Piterate().

H5P__iterate_cb() returns ret_value unconditionally.

Backing up a bit, **H5P__iterate_plist_pclass_cb()** first checks to see if the supplied name is in either the "seen" skip list (udata_int→seen) or the properties deleted skip list (prop→del). If it isn't, H5P__iterate_plist_pclass_cb() calls

```
ret_value = H5P__iterate_plist_cb(prop, name, udata_int)
```

and returns ret_value. See above for a discussion of H5P__iterate_plist_cb.

THE PROPERTY LIST CLASS CASE:

H5P__iterate_pclass() is similar to H5P_iterate_plist(), but simpler.

It does not create the "seen" skip list, as only the properties of the target property list class are scaned, and thus the "seen" list is not necessary. It does initialize udata_int (an instance of H5P_iter_plist_ud_t) as follows:

```
/* Set up iterator callback info */
udata_int.cb_func      = cb_func;   // H5P__iterate_cb() in this case
udata_int.udata        = udata;     // the udata initialized by H5Piterate()
udata_int.curr_idx_ptr = &curr_idx; // curr_idx is a local integer
udata_int.prev_idx     = *idx;
```

It then calls

```
ret_value = H5SL_iterate(pclass->props, H5P__iterate_pclass_cb, &udata_int);
```

H5P__iterate_pclass() returns ret_value after setting *idx = *(udata_int.cur_idx_ptr).

As discussed above, **H5SL_iterate()** simply walks the skip list, and calls the supplied call back function on the contents of each node until it either reaches the end of the list, or the supplied function returns a non-zero value. In this case, it calls:

```
H5P__iterate_pclass_cb(prop, name, udata_int)
```

Here, prop is a pointer to the instance of H5P_getprop_t, and name is the name of the property pointed to by prop.

Likewise, **H5P__iterate_pclass_cb()** is similar to H5P__iterate_plist_cb(), only simpler.

As per H5P__iterate_plist_cb(), H5p__iterate_pclass_cb() tests to see if

```
*(udata_int→curr_idx_ptr) >= udata_int→prev_idx
```

and calls:

```
ret_value = (*udata_int->cb_func)(prop, udata_int→udata)
```

if it is. Since udata_int→cb_func == H5P__iterate_cb() this case, the call is really:

```
ret_value =  H5P__iterate_cb(prop, udata_int→udata)
```

If ret_value isn't zero, H5P__iterate_plist() returns ret_value immediately.

Otherwise, H5P__iterate_plist_cb() increments *(udata_int→curr_idx_ptr) before returning ret_value.

As discussed at the end of the PROPERTY LIST CASE above, **H5P__iterate_cb()** simply calls the user supplied function

```
ret_value = (*udata->iter_func)(udata->id, prop->name, udata->iter_data)
```

where udata→iter_func, and udata→iter_data are the iter_func and iter_data parameters passed to H5Piterate, and udata→id is both the id of the property list within which *prop resides, and also the id parameter passed to H5Piterate().

Multi-thread safety concerns:

```
/**
* \ingroup GPLOA
*
* \brief Registers a permanent property with a property list class
*
* \plistcls_id{cls_id}
* \param[in] name       Name of property to register
* \param[in] size       Size of property in bytes
* \param[in] def_value  Default value for property in newly created
*                       property lists
* \param[in] create     Callback routine called when a property list is
*                       being created and the property value will be
*                       initialized
* \param[in] set        Callback routine called before a new value is
*                       copied into the property's value
* \param[in] get        Callback routine called when a property value is
*                       retrieved from the property
* \param[in] prp_del    Callback routine called when a property is deleted
*                       from a property list
* \param[in] copy       Callback routine called when a property is copied
*                       from a property list
* \param[in] compare    Callback routine called when a property is compared
*                       with another property list
* \param[in] close      Callback routine called when a property list is
*                       being closed and the property value will be
*                       disposed of
*
* \return  \herr_t
*
* \details H5Pregister2() registers a new property with a property list
*          class. The \p cls_id identifier can be obtained by calling
*          H5Pcreate_class(). The property will exist in all property
*          list objects of \p cl_id created after this routine finishes. The
*          name of the property must not already exist, or this routine
*          will fail. The default property value must be provided and all
*          new property lists created with this property will have the
*          property value set to the default value. Any of the callback
*          routines may be set to NULL if they are not needed.
*
*          Zero-sized properties are allowed and do not store any data in
*          the property list. These may be used as flags to indicate the
*          presence or absence of a particular piece of information. The
*          default pointer for a zero-sized property may be set to NULL.
*          The property \p create and \p close callbacks are called for
*          zero-sized properties, but the \p set and \p get callbacks are
*          never called.
*
*          The \p create routine is called when a new property list with
*          this property is being created. The #H5P_prp_create_func_t
*          callback function is defined as follows:
*
*          \snippet this H5P_prp_cb1_t_snip
*
*          The parameters to this callback function are defined as follows:
*
*          <table>
*           <tr>
*            <td>\Code{const char * name}</td>
*            <td>IN: The name of the property being modified</td>
*           </tr>
*           <tr>
*            <td>\Code{size_t size}</td>
```

```
*             <td>IN: The size of the property in bytes</td>
*            </tr>
*            <tr>
*             <td>\Code{void * value}</td>
*             <td>IN/OUT: The default value for the property being created,
*                which will be passed to H5Pregister2()</td>
*            </tr>
*           </table>
*
*           The \p create routine may modify the value to be set and those
*           changes will be stored as the initial value of the property.
*           If the \p create routine returns a negative value, the new
*           property value is not copied into the property and the
*           \p create routine returns an error value.
*
*           The \p set routine is called before a new value is copied into
*           the property. The #H5P_prp_set_func_t callback function is defined
*           as follows:
*
*           \snippet this H5P_prp_cb2_t_snip
*
*           The parameters to this callback function are defined as follows:
*
*           <table>
*            <tr>
*             <td>\ref hid_t \c prop_id</td>
*             <td>IN: The identifier of the property list being modified</td>
*            </tr>
*            <tr>
*             <td>\Code{const char * name}</td>
*             <td>IN: The name of the property being modified</td>
*            </tr>
*            <tr>
*             <td>\Code{size_t size}</td>
*             <td>IN: The size of the property in bytes</td>
*            </tr>
*            <tr>
*             <td>\Code{void *value}</td>
*             <td>IN/OUT: Pointer to new value pointer for the property
*                 being modified</td>
*            </tr>
*           </table>
*
*           The \p set routine may modify the value pointer to be set and
*           those changes will be used when setting the property's value.
*           If the \p set routine returns a negative value, the new property
*           value is not copied into the property and the \p set routine
*           returns an error value. The \p set routine will not be called
*           for the initial value; only the \p create routine will be called.
*
*           \b Note: The \p set callback function may be useful to range
*           check the value being set for the property or may perform some
*           transformation or translation of the value set. The \p get
*           callback would then reverse the transformation or translation.
*           A single \p get or \p set callback could handle multiple
*           properties by performing different actions based on the property
*           name or other properties in the property list.
*
*           The \p get routine is called when a value is retrieved from a
*           property value. The #H5P_prp_get_func_t callback function is
*           defined as follows:
*
*           \snippet this H5P_prp_cb2_t_snip
```

```
 *
 *          The parameters to the callback function are defined as follows:
 *
 *          <table>
 *           <tr>
 *            <td>\ref hid_t \c prop_id</td>
 *            <td>IN: The identifier of the property list being
 *                queried</td>
 *           </tr>
 *           <tr>
 *            <td>\Code{const char * name}</td>
 *            <td>IN: The name of the property being queried</td>
 *           </tr>
 *           <tr>
 *            <td>\Code{size_t size}</td>
 *            <td>IN: The size of the property in bytes</td>
 *           </tr>
 *           <tr>
 *            <td>\Code{void * value}</td>
 *            <td>IN/OUT: The value of the property being returned</td>
 *           </tr>
 *          </table>
 *
 *          The \p get routine may modify the value to be returned from the
 *          query and those changes will be returned to the calling routine.
 *          If the \p set routine returns a negative value, the query
 *          routine returns an error value.
 *
 *          The \p prp_del routine is called when a property is being
 *          deleted from a property list. The #H5P_prp_delete_func_t
 *          callback function is defined as follows:
 *
 *          \snippet this H5P_prp_cb2_t_snip
 *
 *          The parameters to the callback function are defined as follows:
 *
 *          <table>
 *           <tr>
 *            <td>\ref hid_t \c prop_id</td>
 *            <td>IN: The identifier of the property list the property is
 *                being deleted from</td>
 *           </tr>
 *           <tr>
 *            <td>\Code{const char * name}</td>
 *            <td>IN: The name of the property in the list</td>
 *           </tr>
 *           <tr>
 *            <td>\Code{size_t size}</td>
 *            <td>IN: The size of the property in bytes</td>
 *           </tr>
 *           <tr>
 *            <td>\Code{void * value}</td>
 *            <td>IN: The value for the property being deleted</td>
 *           </tr>
 *          </table>
 *
 *          The \p prp_del routine may modify the value passed in, but the
 *          value is not used by the library when the \p prp_del routine
 *          returns. If the \p prp_del routine returns a negative value,
 *          the property list  delete routine returns an error value but
 *          the property is still deleted.
 *
 *          The \p copy routine is called when a new property list with
```

```
*          this property is being created through a \p copy operation.
*          The #H5P_prp_copy_func_t callback function is defined as follows:
*
*          \snippet this H5P_prp_cb1_t_snip
*
*          The parameters to the callback function are defined as follows:
*
*          <table>
*           <tr>
*            <td>\Code{const char * name}</td>
*            <td>IN: The name of the property being copied</td>
*           </tr>
*           <tr>
*            <td>\Code{size_t size}</td>
*            <td>IN: The size of the property in bytes</td>
*           </tr>
*           <tr>
*            <td>\Code{void * value}</td>
*            <td>IN/OUT: The value for the property being copied</td>
*           </tr>
*          </table>
*
*          The \p copy routine may modify the value to be set and those
*          changes will be stored as the new value of the property. If
*          the \p copy routine returns a negative value, the new
*          property value is not copied into the property and the \p copy
*          routine returns an error value.
*
*          The \p compare routine is called when a property list with this
*          property is compared to another property list with the same
*          property. The #H5P_prp_compare_func_t callback function is
*          defined as follows:
*
*          \snippet this H5P_prp_compare_func_t_snip
*
*          The parameters to the callback function are defined as follows:
*
*          <table>
*           <tr>
*            <td>\Code{const void * value1}</td>
*            <td>IN: The value of the first property to compare</td>
*           </tr>
*           <tr>
*            <td>\Code{const void * value2}</td>
*            <td>IN: The value of the second property to compare</td>
*           </tr>
*           <tr>
*            <td>\Code{size_t size}</td>
*            <td>IN: The size of the property in bytes</td>
*           </tr>
*          </table>
*
*          The \p compare routine may not modify the values. The \p compare
*          routine should return a positive value if \p value1 is greater
*          than \p value2, a negative value if \p value2 is greater than
*          \p value1 and zero if \p value1 and \p value2 are equal.
*
*          The \p close routine is called when a property list with this
*          property is being closed. The #H5P_prp_close_func_t callback
*          function is defined as follows:
*
*          \snippet this H5P_prp_cb1_t_snip
*
```

```
*          The parameters to the callback function are defined as follows:
*
*          <table>
*           <tr>
*            <td>\Code{const char * name}</td>
*            <td>IN: The name of the property in the list</td>
*           </tr>
*           <tr>
*            <td>\Code{size_t size}</td>
*            <td>IN: The size of the property in bytes</td>
*           </tr>
*           <tr>
*            <td>\Code{void * value}</td>
*            <td>IN: The value for the property being closed</td>
*           </tr>
*          </table>
*
*          The \p close routine may modify the value passed in, but the
*          value is not used by the library when the \p close routine returns.
*          If the \p close routine returns a negative value, the property
*          list close routine returns an error value but the property list is
*          still closed.
*
* \since 1.8.0
*
*/
H5_DLL herr_t H5Pregister2(hid_t cls_id, const char *name, size_t size,
                           void *def_value, H5P_prp_create_func_t create,
                           H5P_prp_set_func_t set, H5P_prp_get_func_t get,
                           H5P_prp_delete_func_t prp_del,
                           H5P_prp_copy_func_t copy,
                           H5P_prp_compare_func_t compare,
                           H5P_prp_close_func_t close);
```

```
H5Pregister2(cls_id, name, size, def_value, create, set, get, prp_del,
 |           copy, compare, close)
 +-H5I_object_verify()
 |  +- …
 +-H5P__register()
 |  | // duplicate class if required
 |  +-H5P__create_class()
 |  |  +-H5FL_CALLOC()
 |  |  +-H5MM_xstrdup()
 |  |  +-H5SL_create()
 |  |  |  +- …
 |  |  +-H5P__access_class(par_class, H5P_MOD_INC_CLS)
 |  |  |  +-H5MM_xfree()
 |  |  |  +-H5SL_destroy()
 |  |  |  |  +- …
 |  |  |  +-H5P__access_class(par_class, H5P_MOD_DEC_CLS) // can’t happen in this case
 |  |  |     + …
 |  |  +-H5MM_xfree() // error cleanup
 |  |  +-H5SL_destroy()
 |  |     +- … // eventually
 |  |          H5P__free_prop_cb()
 |  |           +-(tprop->close)(tprop->name, tprop->size, tprop->value);
 |  |           |  +- … // property specific – may not exist
 |  |           +-H5P__free_prop(tprop);
 |  |              +-H5MM_xfree()
 |  |              +-H5FL_FREE()
```

```
 |  +-H5SL_first()
 |  |  +- …
 |  +-H5P__dup_prop(prop, H5P_PROP_WITHIN_LIST)
 |  |  +-H5FL_MALLOC()
 |  |  +-H5MM_memcpy()
 |  |  +-H5MM_xstrdup()
 |  |  +-H5MM_xfree() // error cleanup
 |  |  +-H5FL_FREE()  // error cleanup
 |  +-H5P__add_prop()
 |  |  +-H5SL_insert()
 |  |     +- …
 |  +-H5SL_next()
 |  |  +- …
 |  |
 |  +-H5P__register_real()
 |  |  +-H5SL_search()
 |  |  |  +- …
 |  |  +-H5P__create_prop()
 |  |  |  +-H5FL_MALLOC()
 |  |  |  +-H5MM_xstrdup()
 |  |  |  +-H5MM_malloc()
 |  |  |  +-H5MM_memcpy()
 |  |  |  +-H5MM_xfree()  // error cleanup
 |  |  |  +-H5FL_FREE()   // error cleanup
 |  |  +-H5P__add_prop()
 |  |  |  +- // see above
 |  |  +-H5P__free_prop()
 |  |     +- // see above
 |  +-H5P__close_class() // error recovery
 |     +-H5P__access_class(pclass, H5P_MOD_DEC_REF)
 |        +-H5MM_xfree()
 |        +-H5SL_destroy()
 |        |  +- …
 |        +-H5P__access_class(par_class, H5P_MOD_DEC_CLS)
 |           +- … // see abpve
 +-H5I_subst()
 |  +- …
 +-H5P__close_class()
    +- // see above
```

In a nutshell:

Insert a new property in a property list class.

If the target class has any directly derived property lists or property list classes, this will result in duplication of the target property list, with the duplicate (with the new property added) replacing the old version in the index.

In greater detail:

H5Pregister2(cls_id, name, size, def_value, prp_create, prp_set, prp_get, prp_delete, prp_copy, prp_cmp, prp_close) first calls H5I_object_verify(cls_id) to obtain a pointer to the target property list class (pclass).

After some sanity checks, it saves a copy of pclass in orig_pclass, and then calls:

```
H5P__register(&pclass, name, size, def_value, prp_create,
              prp_set, prp_get, NULL, NULL, prp_delete,
              prp_copy, prp_cmp, prp_close)
```

H5P__register() is discussed at great length in the “PROPERTY LIST CLASS CASE” of the section on H5Pcopy_prop(), and the reader is referred there for details.

For purposes of this discussion, perhaps the following summary will suffice.

The objective of H5P__register() is to insert the new property into the target property list class. However, if there are any extant property lists, or property list classes directly derived from dst_pclass, it must duplicate the supplied property list class, insert the new property into the duplicate, and set *pclass to point to the modified duplicate. The duplicate later replaces the earlier version of *pclass in the index. The original version of *pclass is then is only accessible via the parent pointers in its derived property lists and property list classes. It is retained until both its plists (number of derived property lists) and classes (number of derived property list classes) fields drop to zero – at which point it is discarded.

After H5P__register() returns, H5Pregister() compares pclass with orig_pclass – the copy it made just before calling H5P__register(). If the two don’t match, it must replace orig_pclass with pclass in the index. It does this with the call

```
H5I_subst(cls_id, pclass)
```

and then calls

```
H5P__close_class(orig_pclass)
```

which decrements old_dst_pclass→ref_count, and may delete it. See discussion in H5Pclose_class() above for further details.

Multi-thread safety concerns:

```
/**
* \ingroup GPLOA
*
* \brief Removes a property from a property list
*
* \plist_id
* \param[in] name Name of property to remove
*
* \return \herr_t
*
* \details H5Premove() removes a property from a property list. Both
*          properties which were in existence when the property list was
*          created (i.e. properties registered with H5Pregister()) and
*          properties added to the list after it was created (i.e. added
*          with H5Pinsert1() may be removed from a property list.
*          Properties do not need to be removed from a property list
*          before the list itself is closed; they will be released
*          automatically when H5Pclose() is called.
*
*          If a \p close callback exists for the removed property, it
*          will be called before the property is released.
*
* \since 1.4.0
*
*/
H5_DLL herr_t H5Premove(hid_t plist_id, const char *name);

H5Premove()
 +-H5I_object_verify()
 |  +- …
 +-H5P_remove()
    +-H5P__do_prop(plist, name, H5P__del_plist_cb, H5P__del_pclass_cb, NULL)
       +-H5SL_search(plist->del, name)
       |  +- …
       +-(*plist_op)(plist, name, prop, udata)
       |  ||
       | H5P__del_plist_cb() // in this case
       |  +-(*(prop->del))(plist->plist_id, name, prop->size, prop→value)
       |  |  // property specific call - different for each property
       |  +-H5MM_xstrdup(name)
       |  +-H5SL_insert()
       |  |  + …
       |  +-H5P__free_prop()
       |     +-H5MM_xfree()
       |     +-H5FL_FREE()
       +-(*pclass_op)(plist, name, prop, udata)
          ||
         H5P__del_pclass_cb() // in this case
          +-H5MM_malloc()
          +-H5MM_memcpy()
          +-(*(prop->del))(plist->plist_id, name, prop->size, tmp_value)
          |  // property specific call - different for each property
          +-H5MM_xstrdup()
          +-H5SL_insert()
             +- …
```

In a nutshell:

Remove a property from a property list.

In greater detail:

H5Premove() looks up the supplied property list id to obtain a pointer (plist) to it. It then calls H5P_remove(plist, name) to remove the named property from plist, and returns whatever that function returns.

H5P_remove() simply calls

```
H5P__do_prop(plist, name, H5P__del_plist_cb, H5P__del_pclass_cb, NULL)
```

and returns whatever that call returns.

H5P__do_prop() first searches plist→del to see if the target property has already been deleted, and fails if it has.

It then searches plist→props for the target property. If it finds it, it stores its address in prop, calls

```
H5P__del_plist_cb(plist, name, prop, NULL)
```

and returns success or failure depending on whether this call succeeds or fails.

If the search of plist→props fails, H5P__do_prop() searches the property lists of the parent property list class(es) for the target property, starting with plist→pclass→props, and working its way up until it either finds the target property, or it runs out of parent property list classes. If this search is successful, it stores a pointer to the target property in prop, calls

```
H5P__del_pclass_cb(plist, name, prop, NULL)
```

and returns success or failure depending on whether this call succeeds or fails. If the search of the parent property list class(es) fails, H5P__do_prop() returns failure.

H5P__del_plist_cb() calls the properties delete callback

```
(*(prop->del))(plist->plist_id, name, prop->size, prop→value)
```

if it exists, duplicates the property name string and inserts it into the plist→del skip list, deletes the property from the plist→props skip list, frees it, and decrements plist→nprops.

H5P__del_pclass_cb() is similar to H5P__del_plist_cb() but subtly different.

Like H5P__del_plist_cb() it calls the properties delete callback if it exists. However, before it does so, it duplicates *(prop→value), and passes a pointer to this duplicate as the final parameter to the properties delete callback. After that, it duplicates the property name string and inserts it into the plist→del skip list, and decrements plist→nprops. Note, however, that since the target property is not in the plist→props skip list, it doesn't attempt to remove it.

Multi-thread safety concerns:

```
/**
* \ingroup GPLOA
*
* \brief Sets a property list value
*
* \plist_id
* \param[in] name  Name of property to modify
* \param[in] value Pointer to value to set the property to
*
* \return \herr_t
*
* \details H5Pset() sets a new value for a property in a property list.
*          If there is a \p set callback routine registered for this
*          property, the \p value will be passed to that routine and any
*          changes to the \p value will be used when setting the property
*          value. The information pointed to by the \p value pointer
*          (possibly modified by the \p set callback) is copied into the
*          property list value and may be changed by the application
*          making the H5Pset() call without affecting the property value.
*
*          The property name must exist or this routine will fail.
*
*          If the \p set callback routine returns an error, the property
*          value will not be modified.
*
*          This routine may not be called for zero-sized properties and
*          will return an error in that case.
*
* \since 1.4.0
*
*/
H5_DLL herr_t H5Pset(hid_t plist_id, const char *name, const void *value);

H5Pset(plist_id, name, value)
 +-H5I_object_verify(plist_id)
 |  +- …
 +-H5P_set(plist, name, value)
    +-H5P__do_prop(plist, name, H5P__set_plist_cb, H5P_set_pclass_cb, &udata)
       +-H5SL_search()
       |  +- …
       +-(*plist_op)(plist, name, prop, udata)
       |  || // in this case
       | H5P__set_plist_cb(plist, name, prop, udata)
       |  +-H5MM_malloc()
       |  +-H5MM_memcopy()
       |  +-(*(prop→set))(plist→plist_id, name, prop→size, tmp_value)
       |  |  +- // property specific
       |  |
       |  | // if prop→del != NULL
       |  +-(*(prop→del)(plist→plist-id, name, prop→size, prop→value)
       |  |  +- // property specific
       |  |
       |  +-H5MM_xfree()
       |
       +-(*pclass_op)(plist, name, prop, udata)
          || // in this case
         H5P__set_pclass_cb(plist, name, prop, udata)
          +-H5MM_malloc()
          +-H5MM_memcpy()
          +-(*(prop→set))(plist→plist_id, name, prop→size, tmp_value)
          |  +- // property specific
```

```
+-H5P__dup_prop(prop, H5P_PROP_WITHIN_LIST)
|  +-H5FL_MALLOC()
|  +-H5MM_memcpy()
|  +-H5MM_xstrdup()
|  +-H5MM_xfree() // error cleanup
|  +-H5FL_FREE()  // error cleanup
+-H5P__add_prop()
|  +-H5SL_insert()
|     +- …
+-H5MM_xfree()
+-H5P__free_prop() // error cleanup
   +-H5MM_xfree()
   +-H5FL_FREE()
```

In a nutshell:

Set the value of an existing property in a property list.

In greater detail:

After some sanity checking, **H5Pset()** calls H5I_object_verify() to obtain a pointer to the target property list, calls:

```
H5P_set(plist, name, value)
```

and returns.

H5P_set() allocates an instance of H5P_prop_set_ud_t (definition below) on its stack

```
/* Typedef for property list set/poke callbacks */
typedef struct {
    const void *value; /* Pointer to value to set */
} H5P_prop_set_ud_t;
```

and initializes it as follows:

```
udata.value = value
```

It then calls

```
H5P__do_prop(plist, name, H5P__set_plist_cb, H5P_set_pclass_cb, &udata)
```

and returns.

H5P__do_prop() first searches plist→del to see if the target property has already been deleted, and fails if it has.

It then searches plist→props for the target property. If it finds it, it stores its address in prop, calls (in this case)

```
H5P__set_plist_cb(plist, name, prop, udata)
```

and returns success or failure depending on whether this call succeeds or fails.

If the search of plist→props fails, H5P__do_prop() searches the property lists of the parent property list class(es) for the target property, starting with plist→pclass→props, and working its way up until it either finds the target property, or it runs out of parent property list classes. If this search is successful, it stores a pointer to the target property in prop, calls (in this case)

```
H5P__set_pclass_cb(plist, name, prop, NULL)
```

and returns success or failure depending on whether this call succeeds or fails. If the search of the parent property list class(es) fails, H5P__do_prop() returns failure.

If prop→set is not NULL, **H5P__set_plist_cb()** allocates a buffer of size prop→size, stores its address in tmp_value, memcpy's udata→value into *tmp_value, calls

```
(*(prop->set))(plist->plist_id, name, prop->size, tmp_value)
```

and then set prp_value = tmp_value.

If prop→set is NULL, it just sets prp_value = udata→value.

This done, H5P__set_plist_cb() test to see if prop→del is not NULL, and if so, calls

```
(*(prop->del))(plist->plist_id, name, prop->size, prop->value)
```

Finally, it memcpy's prp_value into prop→value, and frees the buffer pointed to by tmp_value (if it exists) before returning.

Note the implication that the del callback doesn't free prop→value.

H5P__set_pclass_cb() starts by setting up prp_value as per H5P__set_plist_cb() above.

It then calls

```
pcopy = H5P__dup_prop(prop, H5P_PROP_WITHIN_LIST)
```

Here, **H5P__dup_prop()** allocates a new instance of H5P_genprop_t (pcopy), copies the image of *prop into it, and (if prop→shared_name is FALSE), duplicates the string pointed to by prop→name and stored its address in pcopy→name. It then duplicates the buffer pointed to by prop→value (if it exists) storing the address of the duplicate in pcopy->value, and returns the address of the new instance of H5P_genprop_t.

This done, H5P__set_pclass_cb() memccpy's prp_value into pcopy→value, and calls

```
H5P__add_prop(plist->props, pcopy)
```

which inserts pcopy into the skip list pointed to by plist→props via a call to H5SL_insert().

Finally, H5P__set_pclass_db() deletes *tmp_value if it has been allocated, and returns. On error, it discards *pcopy via a call to H5P__free_prop() if it was created prior to the failure.

Multi-Thread safety concerns:

```
/**
* \ingroup GPLOA
*
* \brief Removes a property from a property list class
*
* \plistcls_id{pclass_id}
* \param[in] name Name of property to remove
*
* \return \herr_t
*
* \details H5Punregister() removes a property from a property list class.
*          Future property lists created of that class will not contain
*          this property; existing property lists containing this property
*          are not affected.
*
* \since 1.4.0
*
*/
H5_DLL herr_t H5Punregister(hid_t pclass_id, const char *name);

H5Punregister()
 +-H5I_object_verify()
 |  +- …
 +-H5P__unregister()
    +-H5SL_search()
    |  +- …
    +-H5SL_remove()
    |  +- …
    +-H5P__free_prop()
       +-H5MM_xfree()
       +-H5FL_FREE()
```

In a nutshell:

Delete a property from the specified property list class.

According to the user documentation, this should have no effect on existing property lists – however examination of the code suggests otherwise. See detailed discussion below.

In greater detail:

H5Punregister() looks up the supplied property list class id to obtain a pointer (pclass) to it. It then calls H5P_unregister(pclass, name) to remove the named property from pclass, and returns whatever that function returns.

H5P__unregister() calls

```
prop = H5SL_search(pclass->props, name)
```

to obtain a pointer to the target property, calls

```
H5SL_remove(pclass->props, prop→name)
```

to remove it from pclass→props, calls

```
H5P__free_prop(prop)
```

to free it, decrements pclass→nprops, sets pclass->revision equal to the global variable H5P_next_rev, increments H5P_next_rev, and returns.

Note: While H5Punregister removes a property from a property list class, decrements its nprops field, and assigns a new unique revision number, it does not duplicate the property list class, apply these changes to the duplicate, and then replace the original version with the new modified version in the index as per H5P__register() (see discussion of H5Pcopy_prop() above).

This has two problematic effects:

First, the version number seen by property lists previously derived from the starting version of the property list class changes. This may or may not be an issue, but it should be noted.

Second, if the target property doesn't have a create callback, and a derived property list class never changes its value, the call to H5Punregister has the effect of removing the target property from all such property lists without decrementing the associated nprops field – recall that such properties are not copied into the property list but are instead read from the parent property list class(es) if accessed (see H5Pcreate() above).

This second issue is much more serious, as not only is it at odds with the documented behavior of the function, it also corrupts the property list data structure. I presume that this is a bug.

Multi-thread safety concerns:

Appendix 2 – H5P internal API calls

In addition to its public API, H5P also has a private API providing property list services to the HDF5 library. For the most part, these calls are similar to their cognates in the public API – but there are some differences, and also some calls which offer additional capabilities.

Since the objective of this exercise is make H5P multi-thread safe so it can be safely called by multiple threads in multi-thread safe VOL connectors, at first glance, the internal H5P API is not relevant to this effort. However, the internal H5P API is used by other packages – including some that are necessary to support multi-thread VOL connectors.

H5Iprivate.h is reproduced below, with annotations to some of the internal API calls and the some of the package initialization calls. Most entries are annotated with a reference to the relevant public API call. Those with no public API cognate have more extensive annotations or no annotation at all for calls specific to particular property lists. Annotations to the initialization calls are a work in progress.

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Copyright by The HDF Group.                                               *
 * Copyright by the Board of Trustees of the University of Illinois.         *
 * All rights reserved.                                                      *
 *                                                                           *
 * This file is part of HDF5.  The full HDF5 copyright notice, including     *
 * terms governing use, modification, and redistribution, is contained in    *
 * the COPYING file, which can be found at the root of the source code       *
 * distribution tree, or in https://www.hdfgroup.org/licenses.               *
 * If you do not have access to either file, you may request a copy from     *
 * help@hdfgroup.org.                                                        *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/*
 * This file contains private information about the H5P module
 */
#ifndef H5Pprivate_H
#define H5Pprivate_H

/* Early typedefs to avoid circular dependencies */
typedef struct H5P_genplist_t H5P_genplist_t;

/* Include package's public header */
#include "H5Ppublic.h"

/* Private headers needed by this file */
#include "H5private.h" /* Generic Functions                 */

/**************************/
/* Library Private Macros */
/**************************/

/* ========  String creation property names ======== */
#define H5P_STRCRT_CHAR_ENCODING_NAME "character_encoding" /* Character set encoding 
for string */

/* If the module using this macro is allowed access to the private variables, access 
them directly */
```

```
#ifdef H5P_MODULE
#define H5P_PLIST_ID(P) ((P)->plist_id)
#define H5P_CLASS(P)    ((P)->pclass)
#else /* H5P_MODULE */
#define H5P_PLIST_ID(P) (H5P_get_plist_id(P))
#define H5P_CLASS(P)    (H5P_get_class(P))
#endif /* H5P_MODULE */

#define H5_COLL_MD_READ_FLAG_NAME "collective_metadata_read"

/****************************/
/* Library Private Typedefs */
/****************************/

typedef enum H5P_coll_md_read_flag_t {
    H5P_FORCE_FALSE = -1,
    H5P_USER_FALSE  = 0,
    H5P_USER_TRUE   = 1
} H5P_coll_md_read_flag_t;

/* Forward declarations for anonymous H5P objects */
typedef struct H5P_genclass_t H5P_genclass_t;

typedef enum H5P_plist_type_t {
    H5P_TYPE_USER             = 0,
    H5P_TYPE_ROOT             = 1,
    H5P_TYPE_OBJECT_CREATE    = 2,
    H5P_TYPE_FILE_CREATE      = 3,
    H5P_TYPE_FILE_ACCESS      = 4,
    H5P_TYPE_DATASET_CREATE   = 5,
    H5P_TYPE_DATASET_ACCESS   = 6,
    H5P_TYPE_DATASET_XFER     = 7,
    H5P_TYPE_FILE_MOUNT       = 8,
    H5P_TYPE_GROUP_CREATE     = 9,
    H5P_TYPE_GROUP_ACCESS     = 10,
    H5P_TYPE_DATATYPE_CREATE  = 11,
    H5P_TYPE_DATATYPE_ACCESS  = 12,
    H5P_TYPE_STRING_CREATE    = 13,
    H5P_TYPE_ATTRIBUTE_CREATE = 14,
    H5P_TYPE_OBJECT_COPY      = 15,
    H5P_TYPE_LINK_CREATE      = 16,
    H5P_TYPE_LINK_ACCESS      = 17,
    H5P_TYPE_ATTRIBUTE_ACCESS = 18,
    H5P_TYPE_VOL_INITIALIZE   = 19,
    H5P_TYPE_MAP_CREATE       = 20,
    H5P_TYPE_MAP_ACCESS       = 21,
    H5P_TYPE_REFERENCE_ACCESS = 22,
    H5P_TYPE_MAX_TYPE
} H5P_plist_type_t;

/* Function pointer for library classes with properties to register */
typedef herr_t (*H5P_reg_prop_func_t)(H5P_genclass_t *pclass);

/*
 * Each library property list class has a variable of this type that contains
 * class variables and methods used to initialize the class.
 */
typedef struct H5P_libclass_t {
    const char *     name; /* Class name */
    H5P_plist_type_t type; /* Class type */

    H5P_genclass_t **   par_pclass;    /* Pointer to global parent class
                                          property list class */
```

```
    H5P_genclass_t **   pclass;        /* Pointer to global property list class */
    hid_t *const        class_id;      /* Pointer to global property list class
                                          ID */
    hid_t *const        def_plist_id;  /* Pointer to global default property
                                          list ID */
    H5P_reg_prop_func_t reg_prop_func; /* Register class's properties */

    /* Class callback function pointers & info */
    H5P_cls_create_func_t create_func; /* Function to call when a property
                                          list is created */
    void *                create_data; /* Pointer to user data to pass along
                                          to create callback */
    H5P_cls_copy_func_t   copy_func;   /* Function to call when a property list
                                          is copied */
    void *                copy_data;   /* Pointer to user data to pass along
                                          to copy callback */
    H5P_cls_close_func_t  close_func;  /* Function to call when a property list
                                          is closed */
    void *                close_data;  /* Pointer to user data to pass along
                                          to close callback */
} H5P_libclass_t;

/*****************************/
/* Library Private Variables */
/*****************************/

/* Predefined property list classes. */
H5_DLLVAR H5P_genclass_t *H5P_CLS_ROOT_g;
H5_DLLVAR H5P_genclass_t *H5P_CLS_OBJECT_CREATE_g;
H5_DLLVAR H5P_genclass_t *H5P_CLS_FILE_CREATE_g;
H5_DLLVAR H5P_genclass_t *H5P_CLS_FILE_ACCESS_g;
H5_DLLVAR H5P_genclass_t *H5P_CLS_DATASET_CREATE_g;
H5_DLLVAR H5P_genclass_t *H5P_CLS_DATASET_ACCESS_g;
H5_DLLVAR H5P_genclass_t *H5P_CLS_DATASET_XFER_g;
H5_DLLVAR H5P_genclass_t *H5P_CLS_FILE_MOUNT_g;
H5_DLLVAR H5P_genclass_t *H5P_CLS_GROUP_CREATE_g;
H5_DLLVAR H5P_genclass_t *H5P_CLS_GROUP_ACCESS_g;
H5_DLLVAR H5P_genclass_t *H5P_CLS_DATATYPE_CREATE_g;
H5_DLLVAR H5P_genclass_t *H5P_CLS_DATATYPE_ACCESS_g;
H5_DLLVAR H5P_genclass_t *H5P_CLS_MAP_CREATE_g;
H5_DLLVAR H5P_genclass_t *H5P_CLS_MAP_ACCESS_g;
H5_DLLVAR H5P_genclass_t *H5P_CLS_ATTRIBUTE_CREATE_g;
H5_DLLVAR H5P_genclass_t *H5P_CLS_ATTRIBUTE_ACCESS_g;
H5_DLLVAR H5P_genclass_t *H5P_CLS_OBJECT_COPY_g;
H5_DLLVAR H5P_genclass_t *H5P_CLS_LINK_CREATE_g;
H5_DLLVAR H5P_genclass_t *H5P_CLS_LINK_ACCESS_g;
H5_DLLVAR H5P_genclass_t *H5P_CLS_STRING_CREATE_g;

/* Internal property list classes */
H5_DLLVAR const struct H5P_libclass_t H5P_CLS_LCRT[1]; /* Link creation */
H5_DLLVAR const struct H5P_libclass_t H5P_CLS_LACC[1]; /* Link access */
H5_DLLVAR const struct H5P_libclass_t H5P_CLS_AACC[1]; /* Attribute access */
H5_DLLVAR const struct H5P_libclass_t H5P_CLS_DACC[1]; /* Dataset access */
H5_DLLVAR const struct H5P_libclass_t H5P_CLS_GACC[1]; /* Group access */
H5_DLLVAR const struct H5P_libclass_t H5P_CLS_TACC[1]; /* Named datatype access */
H5_DLLVAR const struct H5P_libclass_t H5P_CLS_MACC[1]; /* Map access */
H5_DLLVAR const struct H5P_libclass_t H5P_CLS_FACC[1]; /* File access */
H5_DLLVAR const struct H5P_libclass_t H5P_CLS_OCPY[1]; /* Object copy */

/******************************/
/* Library Private Prototypes */
/******************************/
```

```
/* Forward declaration of structs used below */
struct H5O_fill_t;
struct H5T_t;
struct H5VL_connector_prop_t;

/* Package initialization routines */
H5_DLL herr_t H5P_init_phase1(void);
H5_DLL herr_t H5P_init_phase2(void);


H5P_init_phase1()
 +-H5I_register_type()
 |  +- …
 +-H5P__create_class()
 |  +-H5FL_CALLOC()
 |  +-H5MM_xstrdup()
 |  +-H5SL_create()
 |  |  +- …
 |  +-H5P__access_class(par_class, H5P_MOD_INC_CLS)
 |  |  +-H5MM_xfree()
 |  |  +-H5SL_destroy()
 |  |  |  +- …
 |  |  +-H5P__access_class(par_class, H5P_MOD_DEC_CLS) // not in this case
 |  |     + …
 |  +-H5MM_xfree() // error cleanup
 |  +-H5SL_destroy()
 |     +- … // eventually
 |          H5P__free_prop_cb()
 |           +-(tprop->close)(tprop->name, tprop->size, tprop->value);
 |           |  +- … // property specific - may not exist
 |           +-H5P__free_prop(tprop);
 |              +-H5MM_xfree()
 |              +-H5FL_FREE()
 +-(*lib_class→reg_prop_func)(*lib_class→pclass)
 |  +- … // varies
 +-H5I_register(H5I_GENPROP_CLS, *lib_class->pclass, FALSE)
 |  +- …
 +-H5P_create_id(*lib_class->pclass, FALSE)
 |  +-H5P__create()
 |  |  +-H5FL_CALLOC()
 |  |  +-H5SL_create()
 |  |  |  +- …
 |  |  +-H5SL_first()
 |  |  |  +- …
 |  |  +-H5SL_item()
 |  |  |  +- …
 |  |  +-H5SL_search()
 |  |  |  +- …
 |  |  +-H5P__do_prop_cb1(plist->props, tmp, tmp->create)
 |  |  |  +-H5MM_malloc()
 |  |  |  +-cb(prop->name, prop->size, tmp_value) // cb == tmp→create
 |  |  |  +-H5P__dup_prop(prop, H5P_PROP_WITHIN_LIST)
 |  |  |  |  +-H5FL_MALLOC()
 |  |  |  |  +-H5MM_memcpy()
 |  |  |  |  +-H5MM_xstrdup()
 |  |  |  |  +-H5MM_xfree() // error cleanup
 |  |  |  |  +-H5FL_FREE()  // error cleanup
 |  |  |  +-H5MM_memcpy()
 |  |  |  +-H5P__add_prop()
 |  |  |  |  +-H5SL_insert()
 |  |  |  |     +- …
 |  |  |  +-H5MM_xfree () // error cleanup
```

```
 |  |  |  +-H5P__free_prop()
 |  |  |  +-H5P__free_prop()
 |  |  |     +-H5MM_xfree()
 |  |  |     +-H5FL_FREE()
 |  |  +-H5SL_insert()
 |  |  |  +- …
 |  |  +-H5SL_next()
 |  |  |  +- …
 |  |  +-H5P__access_class(plist->pclass, H5P_MOD_INC_LST)
 |  |  |   // in this case, increment the plist count in the pclass
 |  |  |
 |  |  +-H5S
0
 |  |  +-H5SL_destroy() // error cleanup
 |  |  |  +- …
 |  |  +-H5SL_close()  // error cleanup
 |  |  |  +- …
 |  |  +-H5FL_FREE() // error cleanup
 |  |
 |  +-H5I_register()
 |  |  +- …
 |  +-(tclass->create_func)(plist_id, tclass→create_data) // class specific,
 |  |                                                    // may not exist
 |  +-H5I_remove() // error cleanup only
 |  |  +- …
 |  +-H5P_close() // error cleanup only
 |     +- // see H5Pclose() above
 |
 +-H5I_clear_type() // error cleanup only
 |  +- …
 +-H5I_dec_ref() // error cleanup only
 |  +- …
 +-H5P__close_class() // error cleanup only
    +- // see H5Pclose_class()


H5P_init_phase2()
 +-H5P__facc_set_def_driver()
    +-HDgetenv(HDF5_DRIVER)
    |  +- …
    +-H5FD_is_driver_registered_by_name()
    |  +- …
    +-H5I_inc_ref()
    |  +- …
    +-H5P__facc_set_def_driver_check_predefined(driver_env_var, &driver_id)
    |  +-
    +-H5FD_register_driver_by_name(driver_env_var, TRUE)
    |  +- …
    +-H5I_object()
    |  +- …
    +-H5P__class_set(def_fapclass, H5F_ACS_FILE_DRV_NAME, &driver_prop)
    |  +-
    +-H5P_set_driver(def_fapl, driver_prop.driver_id, driver_prop.driver_info,
    |  |             driver_prop.driver_config_str)
    |  +-
    +-H5I_dec_app_ref(driver_id) // error cleanup
       +- …
```

H5P__init_phase1() starts by creating the indexes for property list classes and property lists:

```
H5I_register_type(H5I_GENPROPCLS_CLS)
H5I_register_type(H5I_GENPROPLST_CLS)
```

The next step is to create the various property list classes used by the HDF5 library, populate their property lists setting default values in passing, and then create the default property lists. The data required to create the property list classes is stored in init_class[] – an array of pointers to constant instances of H5P_libclass_t, whose definition is reproduced below for convenience:

```
typedef struct H5P_libclass_t {
    const char *     name; /* Class name */
    H5P_plist_type_t type; /* Class type */

    H5P_genclass_t **   par_pclass;    /* Pointer to global parent
                                          class property list class */
    H5P_genclass_t **   pclass;        /* Pointer to global property
                                          list class */
    hid_t *const        class_id;      /* Pointer to global property
                                          list class  ID */
    hid_t *const        def_plist_id;  /* Pointer to global default
                                          property list ID */
    H5P_reg_prop_func_t reg_prop_func; /* Register class's
                                          properties */

    /* Class callback function pointers & info */
    H5P_cls_create_func_t create_func; /* Function to call when a
                                          property list is created */
    void *                create_data; /* Pointer to user data to pass
                                          along to create callback */
    H5P_cls_copy_func_t   copy_func;   /* Function to call when a
                                          property list is copied */
    void *                copy_data;   /* Pointer to user data to pass
                                          along to copy callback */
    H5P_cls_close_func_t  close_func;  /* Function to call when a
                                          property list is closed */
    void *                close_data;  /* Pointer to user data to pass
                                          along to close callback */
} H5P_libclass_t;
```

init_class[] is initialized as follows:

```
/* List of all property list classes in the library */
/* (order here is not important, they will be initialized in the proper
*      order according to their parent class dependencies)
*/
static H5P_libclass_t const *const init_class[] = {
   H5P_CLS_ROOT,   /* Root */
   H5P_CLS_OCRT,   /* Object create */
   H5P_CLS_STRCRT, /* String create */
   H5P_CLS_LACC,   /* Link access */
   H5P_CLS_GCRT,   /* Group create */
   H5P_CLS_OCPY,   /* Object copy */
   H5P_CLS_GACC,   /* Group access */
   H5P_CLS_FCRT,   /* File creation */
   H5P_CLS_FACC,   /* File access */
   H5P_CLS_DCRT,   /* Dataset creation */
   H5P_CLS_DACC,   /* Dataset access */
   H5P_CLS_DXFR,   /* Data transfer */
   H5P_CLS_FMNT,   /* File mount */
   H5P_CLS_TCRT,   /* Datatype creation */
```

```
    H5P_CLS_TACC,    /* Datatype access */
    H5P_CLS_MCRT,    /* Map creation */
    H5P_CLS_MACC,    /* Map access */
    H5P_CLS_ACRT,    /* Attribute creation */
    H5P_CLS_AACC,    /* Attribute access */
    H5P_CLS_LCRT,    /* Link creation */
    H5P_CLS_VINI,    /* VOL initialization */
    H5P_CLS_RACC     /* Reference access */
};
```

The individual entries in the array are mostly initialized in H5Pint.c. Note that the instances of H5P_libclass_t contain references to values that are not known until run time. This is handled by initializing fields to point to global variables that are initialized at run time. The following initialization of H5P_CLS_FACC gives an example of this.

```
/* File access property list class library initialization object */
const H5P_libclass_t H5P_CLS_FACC[1] = {{
  "file access",       /* Class name for debugging     */
  H5P_TYPE_FILE_ACCESS, /* Class type                  */

  &H5P_CLS_ROOT_g,          /* Parent class               */
  &H5P_CLS_FILE_ACCESS_g,    /* Pointer to class           */
  &H5P_CLS_FILE_ACCESS_ID_g, /* Pointer to class ID        */
  &H5P_LST_FILE_ACCESS_ID_g, /* Pointer to default property list ID */
  H5P__facc_reg_prop,       /* Default property registration routine */

  NULL, /* Class creation callback      */
  NULL, /* Class creation callback info */
  NULL, /* Class copy callback          */
  NULL, /* Class copy callback info     */
  NULL, /* Class close callback         */
  NULL  /* Class close callback info    */
}};
```

With the background, we can now discuss the actual initialization.

At the conceptual level, H5P_init_phase1() scans each entry in the init_class[] array. If the property list class and default property list described by this entry is uninitialized, it checks to see if all prerequisites for initialization are met, and if so, creates the indicated property list class and default property list. It repeats this scan until there is no activity for a full scan of of init_class[], verifies that the required number of initializations have occurred, and returns.

Let lib_class be a pointer to the element of init_class[] currently under examination in the above scan. Proceed as follows for each such entry before going on to the next:

If *lib_class→class_id is not equal -1, the indicated property list class has been initialized. Go on to the next entry in the scan.

Similarly, if lib_class→par_pclass is not NULL, and *lib_class→par_pclass is, the parent property list class hasn't been created yet. Go on to the next entry in the scan.

If lib_class→par_pclass is NULL (i.e. this is the root property list class) or *lib_class->par_pclass is not NULL (i.e. the parent class has been initiaized), create the indicated property list class as follows:

Call:

```
*lib_class->pclass = H5P__create_class(par_pclass,
                                       lib_class->name,
                                       lib_class->type,
                                       lib_class->create_func,
                                       lib_class->create_data,
                                       lib_class->copy_func,
                                       lib_class->copy_data,
                                       lib_class->close_func,
                                       lib_class->close_data)
```

where par_class = NULL if lib_class→par_pclass is NULL, and *lib_class→par_pclass otherwise, This call creates the indicated property list class, but does not populate it beyond the properties inherited from its parent class. See H5Pcopy() above for further details on H5P__create_class().

Next, if lib_class→reg_prop_func is not NULL, call:

```
(*lib_class->reg_prop_func)(*lib_class->pclass)
```

To create the properties specific to the new property list class, and insert them.

Once the new property list class is created, it is registered via the call:

```
*lib_class->class_id = H5I_register(H5I_GENPROP_CLS, *lib_class->pclass, FALSE)
```

Finally, check to see if the default property list of the new class already exists, and create it if not via the call:

```
*lib_class->def_plist_id = H5P_create_id(*lib_class->pclass, FALSE)
```

All this done, go on to the next entry in the scan.

```
/* Internal versions of API routines */
H5_DLL herr_t H5P_close(H5P_genplist_t *plist);

**** See H5Pclose() ****


H5_DLL hid_t  H5P_create_id(H5P_genclass_t *pclass, hbool_t app_ref);

**** See H5Pcreate() *****


H5_DLL hid_t  H5P_copy_plist(const H5P_genplist_t *old_plist, hbool_t app_ref);

**** See H5Pcopy() ****


H5_DLL herr_t H5P_get(H5P_genplist_t *plist, const char *name, void *value);

**** See H5Pget() ****


H5_DLL herr_t H5P_set(H5P_genplist_t *plist, const char *name, const void *value);

**** See H5Pset() ****


H5_DLL herr_t H5P_peek(H5P_genplist_t *plist, const char *name, void *value);

H5P_peek(plist, name, value)
 |
 | // udata.value = value
 |
 +-H5P__do_prop(plist, name, H5P__peek_cb, H5P__peek_cb, &udata)
    +-H5SL_search()
    |  +- …
    |  // In this case, the same callback is provided for both the
    |  // plist_op and pclass_op parameters – hence simplifying the
    |  // call tree in this case.
    +-H5P__peek_cb(plist, name, prop, udata)
       +-H5MM_memcpy()
```

Similar to H5P_get() – the main difference is that H5P__peek_cb simply memcpy()s prop→value into the supplied value buffer instead of calling prop→copy() for this purpose.

```
H5_DLL herr_t H5P_poke(H5P_genplist_t *plist, const char *name,
                       const void *value);

**** See H5Pdecode() ****


H5_DLL herr_t H5P_insert(H5P_genplist_t *plist,
                         const char *name,
                         size_t size,
                         void *value,
                         H5P_prp_set_func_t prp_set,
                         H5P_prp_get_func_t prp_get,
                         H5P_prp_encode_func_t prp_encode,
                         H5P_prp_decode_func_t prp_decode,
                         H5P_prp_delete_func_t prp_delete,
                         H5P_prp_copy_func_t prp_copy,
```

```
                         H5P_prp_compare_func_t prp_cmp,
                         H5P_prp_close_func_t prp_close);

**** See H5Pinsert2() ****


H5_DLL herr_t H5P_remove(H5P_genplist_t *plist, const char *name);

***** See H5Premove() ****


H5_DLL htri_t H5P_exist_plist(const H5P_genplist_t *plist, const char *name);

**** See H5Pexist() ****


H5_DLL htri_t H5P_class_isa(const H5P_genclass_t *pclass1,
                            const H5P_genclass_t *pclass2);

**** See H5Pisa_class() ****


H5_DLL char * H5P_get_class_name(H5P_genclass_t *pclass);

**** See H5Pget_class_name() ****


/* Internal helper routines */
H5_DLL herr_t      H5P_get_nprops_pclass(const H5P_genclass_t *pclass,
                                         size_t *nprops, hbool_t recurse);

**** See H5Pget_nprops() ****

// only used in H5P.c and H5Pint.c – make it a package function?


H5_DLL hid_t       H5P_peek_driver(H5P_genplist_t *plist);

// used in H5FD, and H5F


H5_DLL const void *H5P_peek_driver_info(H5P_genplist_t *plist);

// used in H5FD


H5_DLL const char *H5P_peek_driver_config_str(H5P_genplist_t *plist);

// used in H5FD, and H5F


H5_DLL herr_t      H5P_set_driver(H5P_genplist_t *plist, hid_t new_driver_id,
                                  const void *new_driver_info,
                                  const char *new_driver_config_str);

// used in H5FD


H5_DLL herr_t      H5P_set_driver_by_name(H5P_genplist_t *plist,
                                          const char *driver_name,
                                          const char *driver_config,
                                          hbool_t app_ref);
```

```
// Only in 5Pfapl.c

H5_DLL herr_t      H5P_set_driver_by_value(H5P_genplist_t *plist,
                                           H5FD_class_value_t driver_value,
                                           const char *driver_config,
                                           hbool_t app_ref);

// Used in H5FD

H5_DLL herr_t      H5P_set_vol(H5P_genplist_t *plist, hid_t vol_id,
                               const void *vol_info);

// used in H5VL

H5_DLL herr_t H5P_reset_vol_class(const H5P_genclass_t *pclass,
                                  const struct H5VL_connector_prop_t *vol_prop);

// Used in H5VL

H5_DLL herr_t H5P_set_vlen_mem_manager(H5P_genplist_t *plist,
                                       H5MM_allocate_t alloc_func,
                                       void *alloc_info,
                                       H5MM_free_t free_func, void *free_info);

// Used in H5D

H5_DLL herr_t H5P_is_fill_value_defined(const struct H5O_fill_t *fill,
                                        H5D_fill_value_t *status);

// Used in H5D and H5O

H5_DLL int    H5P_fill_value_cmp(const void *value1, const void *value2,
                                 size_t size);

// Used in H5D

H5_DLL herr_t H5P_modify_filter(H5P_genplist_t *plist, H5Z_filter_t filter,
                                unsigned flags, size_t cd_nelmts,
                                const unsigned cd_values[]);

// Used in H5Z

H5_DLL herr_t H5P_get_filter_by_id(H5P_genplist_t *plist, H5Z_filter_t id,
                                   unsigned int *flags, size_t *cd_nelmts,
                                   unsigned cd_values[], size_t namelen,
                                   char name[], unsigned *filter_config);

// Used in H5Z

H5_DLL htri_t H5P_filter_in_pline(H5P_genplist_t *plist, H5Z_filter_t id);

// Used in H5Z
```

```
/* Query internal fields of the property list struct */
H5_DLL hid_t H5P_get_plist_id(const H5P_genplist_t *plist);

// Doesn't appear to be used in the library


H5_DLL H5P_genclass_t *H5P_get_class(const H5P_genplist_t *plist);

**** See H5Pget_class()

// Used in H5trace.c



/* *SPECIAL* Don't make more of these! -QAK */
H5_DLL htri_t H5P_isa_class(hid_t plist_id, hid_t pclass_id);

**** See H5Pisa_class() ****

// Used in H5CX, H5D, H5F, H5FD, H5G, H5L, H5M, H5O, H5R, H5T, and H5VL


H5_DLL H5P_genplist_t *H5P_object_verify(hid_t plist_id, hid_t pclass_id);


// Used in H5FD, H5F, H5L, H5VL, and H5Z


/* Private DCPL routines */
H5_DLL herr_t H5P_fill_value_defined(H5P_genplist_t *plist,
                                     H5D_fill_value_t *status);

// Used in H5Z


H5_DLL herr_t H5P_get_fill_value(H5P_genplist_t *plist,
                                 const struct H5T_t *type,
                                 void *value);

// used in H5Z


H5_DLL int    H5P_ignore_cmp(const void H5_ATTR_UNUSED *val1,
                             const void H5_ATTR_UNUSED *val2,
                             size_t H5_ATTR_UNUSED size);

// Doesn't appear to be used in the library

#endif /* H5Pprivate_H */
```

Appendix 3 – Original Proposal For Generic Properties?

Thank you to Elena Pourmal for finding the following document, and bringing it to my attention. https://support.hdfgroup.org/HDF5/doc_resource/H5Generic_Props.html#H5Pcopy_prop.

My understanding is that the original property list facility was implemented as a flat structure, supporting only a pre-defined set of properties for each type of property list.

The following document appears to be an early RFC written by Quincey Koziol for the re-write of H5P to support generic properties. Until only five or ten years ago, proposals for extensions / changes in the HDF5 library (typically referred to as RFCs) usually consisted of user level documentation for the new feature / revision along with a brief discussion of motivation. Implementation details were seldom if ever discussed. As it is dated 9/10/01, the document is consistent with this practice.

Leaving aside implementation concerns, the motivation still applies – specifically the need to allow user defined properties to configure user supplied VFDs. Indeed, the VOL layer has only accentuated this requirement.

While this is not a design document as I would use the term, it does shed light on the objectives of the current implementation of H5P – and thus makes the current implementation easier to understand.

Generic Properties Overview and Justification

It is useful to allow "drivers" (VFL, VDL, datatype conversion, etc.) to create properties for controlling features they wish to add to the library. Allowing a driver to create properties when installed at run-time will enable new features to be easily created and controlled while localizing the changes to just the section of code being modified or added. This should allow easier maintenance and evolution of the library's properties in future versions of the library code.

It would also be useful to give users the ability to create and set properties which are temporary in nature and do not need to be stored longer than the application is active. These would allow users to set application specific properties which can be set and queried during the application's execution.

Generic Properties Implementation

The existing property list classes would be modified so that the existing properties are generic properties which are registered when the library starts up. The existing property list API functions would become wrappers around the new "generic" get/set functions. This would allow the library to become more modular, with each driver or API registering its own properties without hard- wiring new fields in the property list class.

The library will provide a default or "empty" property list class with no property values defined for property lists of that class. A mechanism for deriving a new property list class will be defined by inheriting from an existing property list class. The existing property list classes defined by the library (file access, dataset creation, etc) will be created during library initialization and will have global constants available for applications to use. (This is similar to the way the library-defined datatypes are created at run-time)

Users may derive new property list classes from any existing property list class, including the completely new classes derived from the default "empty" property list class, or other user-derived property list classes. User-derived property list classes which are derived from the library-defined classes may be passed to API functions which expect library-defined property lists and the API functions will traverse the inherited classes to find the correct class to retrieve information.

The new generic property list API functions allow properties to be registered for each property list class (library or user defined) to create a set of initial properties for newly created property lists of that class. These registered properties can have default values for each new property list created for that class.

Temporary generic properties can also be attached to any existing property list without affecting new property lists of that class.

Property names beginning with "H5" are reserved for library use and should not be used by third-party applications or libraries.

The names and sizes of property values for each property are local to each property list and changing them in a property list class do not affect existing property lists.

API Changes for Implementing Generic Properties

New Functions:

H5Pcreate_class	-	Create a new property list class.
H5Pcreate_list	-	Create a new property list of a given class.
H5Pregister	-	Register a permanent property with a class.
H5Pinsert	-	Create a temporary property for a property list.
H5Pset	-	Set an existing property (permanent or temporary) to a value.
H5Pexist	-	Query whether a property exists in a property list or class.
H5Pget_size	-	Query size of property value in bytes.

H5Pget_nprops	-	Query number of properties in list or class
H5Pget_class_name	-	Retrieve the name of a class object
H5Pget_class_parent	-	Retrieve a property class's parent class
H5Pisa_class	-	Checks if a property list is a member of a property class
H5Pget	-	Retrieve property value.
H5Pequal	-	Compares two property lists or classes for equality
H5Piterate	-	Iterates over properties in a property class or list
H5Pcopy_prop	-	Copies a property from one list to another
H5Premove	-	Removes a property from a property list.
H5Punregister	-	Un-register a permanent property from a class.
H5Pclose_list	-	Close a property list.
H5Pclose_class	-	Remove a property list class.

Removed Functions:

H5Pcreate and H5Pclose are replaced with H5Pcreate_list and H5Pclose_list.

Changes to Existing Functions:

All the existing H5Pget/set routines would need to be changed to use the new generic register/unregister and get/set routines for the properties they manage, but that shouldn't be a user-visible change. Also, H5Pget_class will change from returning a H5P_class_t to the ID of a generic property class.

New API Function Definitions

NAME

H5Pcreate_class

PURPOSE

Create a new property list class

USAGE

hid_t H5Pcreate_class(*class, name, create, copy, close*)

hid_t *class*;	IN: Property list class to inherit from.
const char **name*;	IN: Name of property list class to register.
H5P_cls_create_func_t *create*;	IN: Callback routine called when a property list is

	created.
H5P_cls_copy_func_t *copy*;	IN: Callback routine called when a property list is copied.
H5P_cls_close_func_t *close*;	IN: Callback routine called when a property list is being closed.

RETURNS

Success: Valid property list class ID
Failure: negative value

DESCRIPTION

Registers a new property list class with the library. The new property list class can inherit from an existing property list class or may be derived from the default "empty" class. New classes with inherited properties from existing classes may not remove those existing properties, only add or remove their own class properties.

The *create* routine is called when a new property list of this class is being created. The H5P_cls_create_func_t is defined as:

typedef herr_t (*H5P_cls_create_func_t)(hid_t *prop_id*, void * *create_data*);

where the parameters to the callback function are:

hid_t *prop_id*;	IN: The ID of the property list being created.
void * *create_data*;	IN/OUT: User pointer to any class creation information needed

The *create* routine is called after any registered *create* function is called for each property value. If the *create* routine returns a negative value, the new list is not returned to the user and the property list creation routine returns an error value.

The *copy* routine is called when an existing property list of this class is copied. The H5P_cls_copy_func_t is defined as:

typedef herr_t (*H5P_cls_copy_func_t)(hid_t *prop_id*, void * *copy_data*);

where the parameters to the callback function are:

hid_t *prop_id*;	IN: The ID of the property list created by copying.
void * *copy_data*;	IN/OUT: User pointer to any class copy information needed

The *copy* routine is called after any registered *copy* function is called for each property value. If the *copy* routine returns a negative value, the new list is not returned to the user and the property list copy routine returns an error value.

The *close* routine is called when a property list of this class is being closed. The H5P_cls_close_func_t is defined as:

typedef herr_t (*H5P_cls_close_func_t)(hid_t *prop_id*, void * *close_data*);

where the parameters to the callback function are:

hid_t *prop_id*; IN: The ID of the property list being closed.

void * *close_data*; IN/OUT: User pointer to any class close information needed

The *close* routine is called before any registered *close* function is called for each property value. If the *close* routine returns a negative value, the property list close routine returns an error value but the property list is still closed.

COMMENTS, BUGS, ASSUMPTIONS

I would like to say "the property list is not closed" when a _close_ routine fails, but I don't think that's possible due to other properties in the list being successfully closed & removed from the property list. I suppose that it would be possible to just remove the properties which have successful _close_ callbacks, but I'm not happy with the ramifications of a mangled, un-closable property list hanging around... Any comments?

NAME

H5Pcreate_list

PURPOSE

Create a new property list class of a given class

USAGE

hid_t H5Pcreate_list(*class*)

hid_t *class*; IN: Class of property list to create.

RETURNS

Success: Valid property list ID
Failure: negative value

DESCRIPTION

Creates a property list of a given class. If a *create* callback exists for the property list class, it is called before the property list is passed back to the user. If *create* callbacks exist for any individual properties in the property list, they are called before the class *create* callback.

COMMENTS, BUGS, ASSUMPTIONS

[?]

NAME

H5Pregister

PURPOSE

Register a permanent property with a property list class

USAGE

herr_t H5Pregister(*class, name, size, default, create, set, get, close*)

hid_t *class*;	IN: Property list class to register permanent property within.
const char * *name*;	IN: Name of property to register.
size_t *size*;	IN: Size of property in bytes.
void * *default*;	IN: Default value for property in newly created property lists.
H5P_prp_create_func_t *create*;	IN: Callback routine called when a property list is being created and the property value will be initialized.
H5P_prp_set_func_t *set*;	IN: Callback routine called before a new value is copied into the property's value.
H5P_prp_get_func_t *get*;	IN: Callback routine called when a property value is retrieved from the property.
H5P_prp_delete_func_t *delete*;	IN: Callback routine called when a property is deleted from a property list.
H5P_prp_copy_func_t *copy*;	IN: Callback routine called when a property is copied from in a property list.
H5P_prp_close_func_t *close*;	IN: Callback routine called when a property list is being closed and the property value will be disposed of.

RETURNS

Success: non-negative value
Failure: negative value

DESCRIPTION

Registers a new property with a property list class. The property will exist in all property list objects of *class* created after this routine finishes. The name of the property must not already exist, or this routine will fail. The default property value must be provided and all new property lists created with this property will have the property value set to the default value. Any of the callback routines may be set to NULL if they are not needed.

Zero-sized properties are allowed and do not store any data in the property list. These may be used as flags to indicate the presence or absence of a particular piece of information. The 'default' pointer for a zero-sized property may be set to NULL. The property 'create' & 'close' callbacks are called for zero-sized properties, but the 'set' and 'get' callbacks are never called.

The *create* routine is called when a new property list with this property is being created. H5P_prp_create_func_t is defined as:

> typedef herr_t (*H5P_prp_create_func_t)(const char **name*, size_t *size*, void **initial_value*);

where the parameters to the callback function are:

const char * *name*;	IN: The name of the property being modified.
size_t *size*;	IN: The size of the property in bytes.
void * *initial_value*;	IN/OUT: The default value for the property being created. (The *default* value passed to H5Pregister)

The *create* routine may modify the value to be set and those changes will be stored as the initial value of the property. If the *create* routine returns a negative value, the new property value is not copied into the property and the create routine returns an error value.

The *set* routine is called before a new value is copied into the property. H5P_prp_set_func_t is defined as:

> typedef herr_t (*H5P_prp_set_func_t)(hid_t *prop_id*, const char **name*, size_t *size*, void **new_value*);

where the parameters to the callback function are:

hid_t *prop_id*;	IN: The ID of the property list being modified.
const char * *name*;	IN: The name of the property being modified.
size_t *size*;	IN: The size of the property in bytes.
void ** *new_value*;	IN/OUT: Pointer to new value pointer for the property being modified.

The *set* routine may modify the value pointer to be set and those changes will be used when setting the property's value. If the *set* routine returns a negative value, the new property value is not copied into the property and the set routine returns an error value. The *set* routine will not be called for the initial value, only the *create* routine will be called.

The *get* routine is called when a value is retrieved from a property value. H5P_prp_get_func_t is defined as:

> typedef herr_t (*H5P_prp_get_func_t)(hid_t *prop_id*, const char **name*, size_t *size*, void **value*);

where the parameters to the callback function are:

hid_t *prop_id*;	IN: The ID of the property list being queried.
const char * *name*;	IN: The name of the property being queried.
size_t *size*;	IN: The size of the property in bytes.
void * *value*;	IN/OUT: The value of the property being returned.

The *get* routine may modify the value to be returned from the query and those changes will be returned to the calling routine. If the *set* routine returns a negative value, the query routine returns an error value.

The *delete* routine is called when a property is being deleted from a property list. H5P_prp_delete_func_t is defined as:

> typedef herr_t (*H5P_prp_delete_func_t)(hid_t *prop_id*, const char **name*, size_t *size*, void **value*);

where the parameters to the callback function are:

hid_t *prop_id*;	IN: The ID of the property list the property is being deleted from.
const char * *name*;	IN: The name of the property in the list.
size_t *size*;	IN: The size of the property in bytes.
void * *value*;	IN: The value for the property being deleted.

The *delete* routine may modify the value passed in, but the value is not used by the library when the *delete* routine returns. If the *delete* routine returns a negative value, the property list delete routine returns an error value but the property is still deleted.

The *copy* routine is called when a new property list with this property is being created through a copy operation. H5P_prp_copy_func_t is defined as:

> typedef herr_t (*H5P_prp_copy_func_t)(const char **name*, size_t *size*, void **value*);

where the parameters to the callback function are:

const char * *name*;	IN: The name of the property being copied.
size_t *size*;	IN: The size of the property in bytes.
void * *value*;	IN/OUT: The value for the property being copied.

The *copy* routine may modify the value to be set and those changes will be stored as the new value of the property. If the *copy* routine returns a negative value, the new property value is not copied into the property and the copy routine returns an error value.

The *close* routine is called when a property list with this property is being closed. H5P_prp_close_func_t is defined as:

> typedef herr_t (*H5P_prp_close_func_t)(hid_t *prop_id*, const char **name*, size_t *size*, void **value*);

where the parameters to the callback function are:

hid_t *prop_id*;	IN: The ID of the property list being closed.
const char * *name*;	IN: The name of the property in the list.
size_t *size*;	IN: The size of the property in bytes.
void * *value*;	IN: The value for the property being closed.

The *close* routine may modify the value passed in, but the value is not used by the library when the *close* routine returns. If the *close* routine returns a negative value, the property list close routine returns an error value but the property list is still closed.

COMMENTS, BUGS, ASSUMPTIONS

The *set* callback function may be useful to range check the value being set for the property or may perform some tranformation/translation of the value set. The *get* callback would then [probably] reverse the transformation, etc. A single *get* or *set* callback could handle multiple properties by performing different actions based on the property name or other properties in the property list.

I would like to say "the property list is not closed" when a *close* routine fails, but I don't think that's possible due to other properties in the list being successfully closed & removed from the property list. I suppose that it would be possible to just remove the properties which have successful *close* callbacks, but I'm not happy with the ramifications of a mangled, un-closable property list hanging around... Any comments?

NAME

H5Pinsert

PURPOSE

Register a temporary property with a property list

USAGE

herr_t H5Pinsert(*plid, name, size, value, set, get, close*)

hid_t *plid*;	IN: Property list id to create temporary property within.
const char **name*;	IN: Name of property to create.
size_t *size*;	IN: Size of property in bytes.
void **value*;	IN: Initial value for the property.
H5P_prp_set_func_t *set*;	IN: Callback routine called before a new value is copied into the property's value.
H5P_prp_get_func_t *get*;	IN: Callback routine called when a property value is

	retrieved from the property.
H5P_prp_delete_func_t *delete*;	IN: Callback routine called when a property is deleted from a property list.
H5P_prp_copy_func_t *copy*;	IN: Callback routine called when a property is copied from an existing property list.
H5P_prp_close_func_t *close*;	IN: Callback routine called when a property list is being closed and the property value will be disposed of.

RETURNS

Success: non-negative value
Failure: negative value

DESCRIPTION

Create a new property in a property list. The property will exist only in this property list and copies made from it. The name of the property must not already exist in this list, or this routine will fail. The initial property value must be provided and the property value will be set to it. The *set* and *get* callback routines may be set to NULL if they are not needed.

Zero-sized properties are allowed and do not store any data in the property list. The default value of a zero-size property may be set to NULL. They may be used to indicate the presence or absence of a particular piece of information.

The *set* routine is called before a new value is copied into the property. The H5P_prp_set_func_t is defined as:

typedef herr_t (*H5P_prp_set_func_t)(hid_t *prop_id*, const char **name*, size_t *size*, void **new_value*);

where the parameters to the callback function are:

hid_t *prop_id*;	IN: The ID of the property list being modified.
const char **name*;	IN: The name of the property being modified.
size_t *size*;	IN: The size of the property in bytes.
void ***new_value*;	IN: Pointer to new value pointer for the property being modified.

The *set* routine may modify the value pointer to be set and those changes will be used when setting the property's value. If the *set* routine returns a negative value, the new property value is not copied into the property and the set routine returns an error value. The *set* routine will be called for the initial value.

The *get* routine is called when a value is retrieved from a property value. The H5P_prp_get_func_t is defined as:

typedef herr_t (*H5P_prp_get_func_t)(hid_t *prop_id*, const char **name*, size_t *size*, void **value*);

where the parameters to the callback function are:

hid_t *prop_id*;	IN: The ID of the property list being queried.
const char **name*;	IN: The name of the property being queried.
size_t *size*;	IN: The size of the property in bytes.
void **value*;	IN: The value of the property being returned.

The *get* routine may modify the value to be returned from the query and those changes will be preserved. If the *get* routine returns a negative value, the query routine returns an error value.

The *delete* routine is called when a property is being deleted from a property list. H5P_prp_delete_func_t is defined as:

typedef herr_t (*H5P_prp_delete_func_t)(hid_t *prop_id*, const char **name*, size_t *size*, void **value*);

where the parameters to the callback function are:

hid_t *prop_id*;	IN: The ID of the property list the property is being deleted from.
const char * *name*;	IN: The name of the property in the list.
size_t *size*;	IN: The size of the property in bytes.
void * *value*;	IN: The value for the property being deleted.

The *delete* routine may modify the value passed in, but the value is not used by the library when the *delete* routine returns. If the *delete* routine returns a negative value, the property list delete routine returns an error value but the property is still deleted.

The *copy* routine is called when a new property list with this property is being created through a copy operation. H5P_prp_copy_func_t is defined as:

typedef herr_t (*H5P_prp_copy_func_t)(const char **name*, size_t *size*, void **value*);

where the parameters to the callback function are:

const char * *name*;	IN: The name of the property being copied.
size_t *size*;	IN: The size of the property in bytes.
void * *value*;	IN/OUT: The value for the property being copied.

The *copy* routine may modify the value to be set and those changes will be stored as the new value of the property. If the *copy* routine returns a negative value, the new property value is not copied into the property and the copy routine returns an error value.

The *close* routine is called when a property list with this property is being closed. The H5P_prp_close_func_t is defined as:

typedef herr_t (*H5P_prp_close_func_t)(hid_t *prop_id*, const char **name*, size_t *size*, void **value*);

where the parameters to the callback function are:

hid_t *prop_id*;	IN: The ID of the property list being closed.
const char **name*;	IN: The name of the property in the list.
size_t *size*;	IN: The size of the property in bytes.
void **value*;	IN: The value for the property being closed.

The *close* routine may modify the value passed in, the value is not used by the library when the *close* routine returns. If the *close* routine returns a negative value, the property list close routine returns an error value but the property list is still closed.

COMMENTS, BUGS, ASSUMPTIONS

The *set* callback function may be useful to range check the value being set for the property or may perform some tranformation/translation of the value set. The *get* callback would then [probably] reverse the transformation, etc. A single *get* or *set* callback could handle multiple properties by performing different actions based on the property name or other properties in the property list.

There is no *create* callback routine for temporary property list objects, the initial value is assumed to have any necessary setup already performed on it.

I would like to say "the property list is not closed" when a *close* routine fails, but I don't think that's possible due to other properties in the list being successfully closed & removed from the property list. I suppose that it would be possible to just remove the properties which have successful *close* callbacks, but I'm not happy with the ramifications of a mangled, un-closable property list hanging around... Any comments?

NAME

H5Pset

PURPOSE

Set a property list value

USAGE

herr_t H5Pset(*plid, name, value*)

hid_t *plid*;	IN: Property list id to modify
const char **name*;	IN: Name of property to modify.

void **value*; IN: Pointer to value to set the property to.

RETURNS

Success: non-negative value
Failure: negative value

DESCRIPTION

Sets a new value for a property in a property list. The property name must exist or this routine will fail. If there is a *set* callback routine registered for this property, the *value* will be passed to that routine and any changes to the *value* will be used when setting the property value. The information pointed at by the *value* pointer (possibly modified by the *set* callback) is copied into the property list value and may be changed by the application making the H5Pset call without affecting the property value.

If the *set* callback routine returns an error, the property value will not be modified. This routine may not be called for zero-sized properties and will return an error in that case.

COMMENTS, BUGS, ASSUMPTIONS

[?]

NAME

H5Pexist

PURPOSE

Query if a property name exists in a property list or class

USAGE

htri_t H5Pexist(*id, name*)
hid_t *id*; IN: Property ID to query
const char **name*; IN: Name of property to check for.

RETURNS

Success: Positive if the property exists in the property object, zero if the property does not exist.
Failure: negative value

DESCRIPTION

This routine checks if a property exists within a property list or class.

COMMENTS, BUGS, ASSUMPTIONS

[?]

NAME

H5Pget_size

PURPOSE

Query size of property value in bytes

USAGE

int H5Pget_size(*id, name, size*)

hid_t *id*;	IN: ID of property object to query
const char **name*;	IN: Name of property to query
size_t **size*;	OUT: Size of property in bytes

RETURNS

Success: non-negative value
Failure: negative value

DESCRIPTION

This routine retrieves the size of a property's value in bytes. Zero- sized properties are allowed and return 0. This function operates on both poperty lists and property classes

COMMENTS, BUGS, ASSUMPTIONS

[?]

NAME

H5Pget_nprops

PURPOSE

Query number of properties in property list or class

USAGE

int H5Pget_nprops(*id, nprops*)

hid_t *id*; IN: ID of property object to query

size_t **nprops*; OUT: Number of properties in object

RETURNS

Success: non-negative value
Failure: negative value

DESCRIPTION

This routine retrieves the number of properties in a property list or class. If a property class ID is given, the number of registered properties in the class is returned in nprops. If a property list ID is given, the current number of properties in the list is returned in nprops.

COMMENTS, BUGS, ASSUMPTIONS

[?]

NAME

H5Pget_class_name

PURPOSE

Retrieve the name of a class

USAGE

char * H5Pget_class_name(*pcid*)

hid_t *pcid*; IN: Property class id to query

RETURNS

Success: Pointer to a malloc'ed string containing the class name
Failure: NULL

DESCRIPTION

This routine retrieves the name of a generic property list class. The pointer to the name must be free'd by the user for successful calls.

COMMENTS, BUGS, ASSUMPTIONS

[?]

NAME

H5Pget_class_parent

PURPOSE

Retrieve the parent class of a property class

USAGE

hid_t H5Pget_class_parent(*pcid*)
    hid_t *pcid*;   IN: Property class ID to query

RETURNS

Success: ID of parent class object
Failure: negative

DESCRIPTION

This routine retrieves an ID for the parent class of a property class.

COMMENTS, BUGS, ASSUMPTIONS

[?]

NAME

H5Pisa_class

PURPOSE

Check if a property list is a member of a class

USAGE

htri_t H5Pisa_class(*plist, pclass*)
    hid_t *plist*;   IN: ID of property list to compare

hid_t *pclass*; IN: ID of property class to compare against

RETURNS

Success: TRUE (positive) if equal, FALSE (zero) if unequal
Failure: negative value

DESCRIPTION

This routine checks if a property list is a member of a class.

COMMENTS, BUGS, ASSUMPTIONS

[?]

NAME

H5Pget

PURPOSE

Query value of property

USAGE

herr_t H5Pget(*plid, name, value*)

hid_t *plid*;	IN: Property list id to query
const char **name*;	IN: Name of property to query
void **value*;	OUT: Pointer to location to copy value of property retrieved into.

RETURNS

Success: non-negative value.
Failure: negative value

DESCRIPTION

Retrieves a copy of the value for a property in a property list. The property name must exist or this routine will fail. If there is a *get* callback routine registered for this property, the copy of the value of the property will first be passed to that routine and any changes to the copy of the value will be used when returning the property value from this routine. If the *get* callback routine returns an error, *value* will not be modified. This routine may be called for zero-sized properties with the *value* set to NULL and the *get* routine will be called with a NULL value if the callback exists.

COMMENTS, BUGS, ASSUMPTIONS

[?]

NAME

H5Pequal

PURPOSE

Compare two property lists or classes for equality

USAGE

htri_t H5Pequal(*id1, id2*)

hid_t *id1*;	IN: First property object to compare
hid_t *id2*;	IN: Second property object to compare

RETURNS

Success: TRUE (positive) if equal, FALSE (zero) if unequal
Failure: negative value

DESCRIPTION

This routine determines whether two property lists or classes are equal to one another. Both id1 and id2 must be either property lists or classes, comparing a list to a class is an error.

COMMENTS, BUGS, ASSUMPTIONS

[?]

NAME

H5Piterate

PURPOSE

Iterates over properties in a property class or list

USAGE

int H5Piterate(*idass_id, idx, iter_func, iter_data*)

hid_t *id*;	IN: ID of property object to iterate over
int * *idx*;	IN/OUT: Index of the property to begin with
H5P_iterate_t *iter_func*;	IN: Function pointer to function to be called with each property iterated over.
void * *iter_data*;	IN/OUT: Pointer to iteration data from user

RETURNS

Success:	Returns the return value of the last call to iter_func if it was non-zero, or zero if all properties have been processed.
Failure:	negative value

DESCRIPTION

This routine iterates over the properties in the property object specified with ID. The properties in both property lists and classes may be iterated over with this function. For each property in the object, the iter_func and some additional information, specified below, are passed to the iter_func function. The iteration begins with the idx property in the object and the next element to be processed by the operator is returned in idx. If idx is NULL, then the iterator starts at the first property; since no stopping point is returned in this case, the iterator cannot be restarted if one of the calls to its operator returns non-zero.

The prototype for H5P_iterate_t is:

```
typedef herr_t (*H5P_iterate_t)(hid_t id, const char *name, void *iter_data)
```

The operation receives the property list or class identifier for the object being iterated over, ID, the name of the current property within the object, name, and the pointer to the operator data passed in to H5Piterate, iter_data.

The return values from an operator are:

Zero:	Causes the iterator to continue, returning zero when all properties have been processed.
Positive:	Causes the iterator to immediately return that positive value, indicating short-circuit success. The iterator can be restarted at the index of the next property.
Negative:	Causes the iterator to immediately return that value, indicating failure. The iterator can be restarted at the index of the next property.

H5Piterate assumes that the properties in the object identified by ID remains unchanged through the iteration. If the membership changes during the iteration, the function's behavior is undefined.

COMMENTS, BUGS, ASSUMPTIONS

[?]

NAME

H5Pcopy_prop

PURPOSE

Copies a property from one list or class to another

USAGE

herr_t H5Pcopy_prop(*dst_id, src_id, name*)

hid_t *dst_id*;	IN: ID of destination property list or class
hid_t *src_id*;	IN: ID of source property list or class
const char **name*;	IN: Name of property to copy

RETURNS

Success: non-negative value.
Failure: negative value

DESCRIPTION

Copies a property from one property list or class to another.

If a property is copied from one class to another, all the property information will be first deleted from the destination class and then the property information will be copied from the source class into the destination class.

If a property is copied from one list to another, the property will be first deleted from the destination list (generating a call to the *close* callback for the property, if one exists) and then the property is copied from the source list to the destination list (generating a call to the *copy* callback for the property, if one exists).

If the property does not exist in the class or list, this call is equivalent to calling H5Pregister or H5Pinsert (for a class or list, as appropriate) and the *create* callback will be called in the case of the property being copied into a list (if such a callback exists for the property).

COMMENTS, BUGS, ASSUMPTIONS

NAME

H5Premove

PURPOSE

Removes a property from a property list

USAGE

herr_t H5Premove(*plid, name*)

hid_t *plid*;	IN: Property list id to modify
const char **name*;	IN: Name of property to remove

RETURNS

Success: non-negative value.
Failure: negative value

DESCRIPTION

Removes a property from a property list. Both properties which were in existance when the property list was created (i.e. properties registered with H5Pregister) and properties added to the list after it was created (i.e. added with H5Pinsert) may be removed from a property list. Properties do not need to be removed a property list before the list itself is closed, they will be released automatically when H5Pclose is called. The *close* callback for this property is called before the property is release, if the callback exists.

COMMENTS, BUGS, ASSUMPTIONS

NAME

H5Punregister

PURPOSE

Removes a property from a property list class

USAGE

herr_t H5Punregister(*class, name*)
H5P_class_t *class*; IN: Property list class to remove permanent property from.
const char **name*; IN: Name of property to remove

RETURNS

Success: non-negative value.
Failure: negative value

DESCRIPTION

Removes a property from a property list class. Future property lists created of that class will not contain this property. Existing property lists containing this property are not affected.

COMMENTS, BUGS, ASSUMPTIONS

[?]

NAME

H5Pclose_list

PURPOSE

Close a property list

USAGE

herr_t H5Pclose_list(*plist*)
hid_t *plist*; IN: Property list to close.

RETURNS

Success: non-negative value
Failure: negative value

DESCRIPTION

Closes a property list. If a *close* callback exists for the property list class, it is called before the property list is destroyed. If *close* callbacks exist for any individual properties in the property list, they are called after the class *close* callback.

COMMENTS, BUGS, ASSUMPTIONS

[?]

NAME

H5Pclose_class

PURPOSE

Closes an existing property list class

USAGE

herr_t H5Pclose_class(*class*)
hid_t *class*; IN: Property list class to close.

RETURNS

Success: non-negative value
Failure: negative value

DESCRIPTION

Removes a property list class from the library. Existing property lists of this class will continue to exist, but new ones are not able to be created.

COMMENTS, BUGS, ASSUMPTIONS

[?]

Examples:

Example #1: Register a new property for future dataset creation property lists. This property uses a "set" callback to range check the values for the property. This set of features would likely be used with Virtual File or Dataset drivers. This example also shows how get/set API functions for the property registered might work.

```
/* Property "set" callback */
herr_t driver_set_check(hid_t prop, const char *name, void *_value)
{
   int *value=(int *)_value;
```

```
    /* Check that this routine is called for proper property name */
    if(strcmp(name,"property1"))
        return(-1);

    /* Range check property value */
    if(*value<0 || *value>SOME_LIMIT)
        return(-1);

    /* Name and value are OK */
    return(0);
}

/* An API routine that sets the property for this driver */
herr_t H5Pset_driver_property1(hid_t prop, int value)
{
    /*
     *  Call the generic H5Pset routine and let the "set" callback do the
     *  range checking, etc.
     */
    return(H5Pset(prop, "property1", &value));
}

/* An API routine that gets the property for this driver */
herr_t H5Pget_driver_property1(hid_t prop, int *value)
{
    /* Call the generic H5Pget routine to retrieve the value */
    return(H5Pget(prop, "property1", value));
}

int setup_driver()
{
    int prop1_default=12;       /* Default value for "property1" */

    [Set up other driver information]

    /* Register new property with Dataset Creation property list class */
    /* Provide a "set" callback, but no "get" callback for this property */
    H5Pregister(H5P_DATASET_CREATE, "property1", sizeof(int), &prop1_default,
        driver_set_check, NULL);

}

int shutdown_driver()
{
    [Shut down other driver information]

    /* Unregister property from Dataset Creation property list class */
    H5Punregister(H5P_DATASET_CREATE, "property1");
}
```

QAK:9/10/01

Appendix 4: Property and Property List Class Callbacks

The callbacks supported by the properties and property list classes are an obvious source of multi-thread safety issues. My objective in this appendix is to identify all the property and property list class callbacks defined by the HDF5 library, and examine them for potential multi-thread issues.

In this appendix, I list the callbacks associated with each of the library defined property list classes, and their associated properties, review the callbacks, and note any potential difficulties.

As it is assumed that it will be necessary to maintain appropriate mutual exclusion on properties when reading or writing them, the primary focus is identifying any other places where mutual exclusion is required when executing property callbacks. While these are identified in the discussions of the individual properties, the following summary may be useful:

- Encode / decode buffer during property list encode / decode operations.
- Calls to H5Z (DXPL / Data transform property)
- Calls to H5S (DXPL / Dataset selection property)

In the case of the property list classes this is easy – none of them appear to have any callbacks. Unfortunately, properties far out number property list classes.

Per the following typedef

```
typedef enum H5P_plist_type_t {
  H5P_TYPE_USER           = 0,
  H5P_TYPE_ROOT           = 1,
  H5P_TYPE_OBJECT_CREATE    = 2,
  H5P_TYPE_FILE_CREATE      = 3,
  H5P_TYPE_FILE_ACCESS      = 4,
  H5P_TYPE_DATASET_CREATE   = 5,
  H5P_TYPE_DATASET_ACCESS   = 6,
  H5P_TYPE_DATASET_XFER     = 7,
  H5P_TYPE_FILE_MOUNT       = 8,
  H5P_TYPE_GROUP_CREATE     = 9,
  H5P_TYPE_GROUP_ACCESS     = 10,
  H5P_TYPE_DATATYPE_CREATE  = 11,
  H5P_TYPE_DATATYPE_ACCESS  = 12,
  H5P_TYPE_STRING_CREATE    = 13,
  H5P_TYPE_ATTRIBUTE_CREATE = 14,
  H5P_TYPE_OBJECT_COPY      = 15,
  H5P_TYPE_LINK_CREATE      = 16,
  H5P_TYPE_LINK_ACCESS      = 17,
  H5P_TYPE_ATTRIBUTE_ACCESS = 18,
  H5P_TYPE_VOL_INITIALIZE   = 19,
  H5P_TYPE_MAP_CREATE       = 20,
```

```
  H5P_TYPE_MAP_ACCESS      = 21,
  H5P_TYPE_REFERENCE_ACCESS = 22,
  H5P_TYPE_MAX_TYPE
} H5P_plist_type_t;
```

the HDF5 library defines 22 property list classes, which form the following inheritance tree:

```
H5P_TYPE_ROOT
 +-Data Transfer Property List (DXPL)
 +-File Access Property List (FAPL)
 |  +-Reference Access Property List (RACCPL) Class
 +-File Mount Property List (FMPL)
 +-VOL Initialization Property List (VIPL)
 +-Link Access Property List (LAPL)
 |  +-Datatype Access Property List (TAPL)
 |  +-Attribute Access Property List (AAPL)
 |  +-Group Access Property List (GAPL)
 |  +-Dataset Access Property List (DAPL)
 |  +-Map Access Property List (MAPL) Class
 +-Object Creation Property List (OCRTPL)
 |  +-Datatype Creation Property List (TCPL)
 |  +-Dataset Creation Property List (DCRTPL)
 |  +-Group Creation Property List (GCRTPL)
 |  |  +-File Creation Property list (FCRTPL) Class
 |  +-Map Create Property List (MCRTPL)
 +-Object Copy Property List (OCPYPL)
 +-String Creation Property List (STRCRTPL)
   +-Attribute Creation Property List (ACRTPL)
   +-Link Creation Property List (LCRTPL)
```

In the following sections, I review each of these, in the order listed in the above inheritance tree.

Root Property List Class

The instance of H5P_libclass_t defining the initialization of the Root property list class is shown below.

```
/* Root property list class library initialization object */
const H5P_libclass_t H5P_CLS_ROOT[1] = {{
  "root",       /* Class name for debugging     */
  H5P_TYPE_ROOT, /* Class type                  */

  NULL,             /* Parent class              */
  &H5P_CLS_ROOT_g,    /* Pointer to class           */
  &H5P_CLS_ROOT_ID_g, /* Pointer to class ID        */
  NULL,             /* Pointer to default property list ID */
  NULL,             /* Default property registration routine */
```

```
  NULL, /* Class creation callback      */
  NULL, /* Class creation callback info */
  NULL, /* Class copy callback          */
  NULL, /* Class copy callback info     */
  NULL, /* Class close callback         */
  NULL  /* Class close callback info    */
}};
```

As there are no callbacks associated with the Root property list class, and no default properties, it follows that the Root property list class does not introduce any multi-thread issues.

Data Transfer Property List (DXPL) Class

Inheritance: ROOT→ DXPL

The instance of H5P_libclass_t defining the initialization of the data transfer property list class is shown below.

```
/* Data transfer property list class library initialization object */
const H5P_libclass_t H5P_CLS_DXFR[1] = {{
  "data transfer",      /* Class name for debugging     */
  H5P_TYPE_DATASET_XFER, /* Class type                  */

  &H5P_CLS_ROOT_g,           /* Parent class              */
  &H5P_CLS_DATASET_XFER_g,    /* Pointer to class            */
  &H5P_CLS_DATASET_XFER_ID_g, /* Pointer to class
ID
A*/
  &H5P_LST_DATASET_XFER_ID_g, /* Pointer to default property list ID */
  H5P__dxfr_reg_prop,        /* Default property registration routine */

  NULL, /* Class creation callback      */
  NULL, /* Class creation callback info */
  NULL, /* Class copy callback          */
  NULL, /* Class copy callback info     */
  NULL, /* Class close callback         */
  NULL  /* Class close callback info    */
}};
```

While there are no callbacks associated with the property list class, the following properties are registered by H5P__dxfr_reg_prop(), the default property registration routine, registers the following properties:

1. Max. temp buffer size property
2. Type conversion buffer property
3. Background buffer property

4. Background buffer type property
5. B-Tree note splitting ratios property
6. Vlen allocation function property
7. Vlen allocation information property
8. Vlen free function property
9. Vlen free information property
10. Vector size property
11. I/O transfer mode property (H5D_XFER_IO_XFER_MODE)
12. I/O transfer mode property (H5D_XFER_MPIO_COLLECTIVE_OPT)
13. I/O transfer mode property (H5D_XFER_MPIO_CHUNK_OPT_HARD)
14. I/O transfer mode property(H5D_XFER_MPIO_CHUNK_OPT_NUM)
15. I/O transfer mode property (H5D_XFER_MPIO_CHUNK_OPT_RATIO)
16. Chunk Optimization mode property
17. Actual I/O mode property
18. Local cause of broken collective I/O property
19. Global cause of broken collective I/O property
20. EDC property
21. Filter callback property
22. Type conversion callback property
23. Data transform property
24. Dataset selection property

Each of these properties is discussed below

Maximum Temp Buffer Size Property

The call used to register the Max. temp buffer size property is reproduced below with the formal parameters added as comments

```
H5P__register_real(pclass                      // H5P_genclass_t *pclass,
                   H5D_XFER_MAX_TEMP_BUF_NAME, // const char *name,
                   H5D_XFER_MAX_TEMP_BUF_SIZE, // size_t size,
                   &H5D_def_max_temp_buf_g,    // const void *def_value,
```

```
                     NULL,                       // H5P_prp_create_func_t prp_create,
                     NULL,                       // H5P_prp_set_func_t prp_set,
                     NULL,                       // H5P_prp_get_func_t prp_get,
                     H5D_XFER_MAX_TEMP_BUF_ENC,  // H5P_prp_encode_func_t prp_encode,
                     H5D_XFER_MAX_TEMP_BUF_DEC,  // H5P_prp_decode_func_t prp_decode,
                     NULL,                       // H5P_prp_delete_func_t prp_delete,
                     NULL,                       // H5P_prp_copy_func_t prp_copy,
                     NULL,                       // H5P_prp_compare_func_t prp_cmp,
                     NULL);                      // H5P_prp_close_func_t prp_close
```

Observe that only the encode and decode callbacks are defined.

H5D_XFER_MAX_TEMP_BUF_ENC and H5D_XFER_MAX_TEMP_BUF_DEC resolve to H5P__encode_size_t() and H5P__decode_size_t() respectively. Their signatures appear in H5Ppkg.h, and are reproduced below:

```
herr_t H5P__encode_size_t(const void *value, void **_pp, size_t *size);

herr_t H5P__decode_size_t(const void **_pp, void *_value);
```

These two routines appear to have no potential for multi-thread issues outside of variables pointed to by their parameters. As it is assumed that appropriate mutual exclusion must be maintained on the property, the only remaining potential area of concern is the encode / decode buffer.

Type conversion buffer property

The call used to register the type conversion buffer property is reproduced below with the formal parameters added as comments

```
H5P__register_real(pclass                        // H5P_genclass_t *pclass,
                     H5D_XFER_TCONV_BUF_NAME,    // const char *name,
                     H5D_XFER_TCONV_BUF_SIZE,    // size_t size,
                     &H5D_def_tconv_buf_g,       // const void *def_value,
                     NULL,                       // H5P_prp_create_func_t prp_create,
                     NULL,                       // H5P_prp_set_func_t prp_set,
                     NULL,                       // H5P_prp_get_func_t prp_get,
                     NULL,                       // H5P_prp_encode_func_t prp_encode,
                     NULL,                       // H5P_prp_decode_func_t prp_decode,
                     NULL,                       // H5P_prp_delete_func_t prp_delete,
                     NULL,                       // H5P_prp_copy_func_t prp_copy,
                     NULL,                       // H5P_prp_compare_func_t prp_cmp,
                     NULL);                      // H5P_prp_close_func_t prp_close
```

Since no callbacks are defined., there are no callback related multi-thread safety issues with this property.

Background buffer property

The call used to register the background buffer property is reproduced below with the formal parameters added as comments

```
H5P__register_real(pclass                          // H5P_genclass_t *pclass,
                   H5D_XFER_BKGR_BUF_NAME,        // const char *name,
                   H5D_XFER_BKGR_BUF_SIZE,        // size_t size,
                   &H5D_def_bkgr_buf_g,,          // const void *def_value,
                   NULL,                          // H5P_prp_create_func_t prp_create,
                   NULL,                          // H5P_prp_set_func_t prp_set,
                   NULL,                          // H5P_prp_get_func_t prp_get,
                   NULL,                          // H5P_prp_encode_func_t prp_encode,
                   NULL,                          // H5P_prp_decode_func_t prp_decode,
                   NULL,                          // H5P_prp_delete_func_t prp_delete,
                   NULL,                          // H5P_prp_copy_func_t prp_copy,
                   NULL,                          // H5P_prp_compare_func_t prp_cmp,
                   NULL);                         // H5P_prp_close_func_t prp_close
```

Since no callbacks are defined., there are no callback related multi-thread safety issues with this property.

Background buffer type property

The call used to register the background buffer property is reproduced below with the formal parameters added as comments

```
H5P__register_real(pclass                          // H5P_genclass_t *pclass,
                   H5D_XFER_BKGR_BUF_TYPE_NAME,   // const char *name,
                   H5D_XFER_BKGR_BUF_TYPE_SIZE,   // size_t size,
                   &H5D_def_bkgr_buf_type_g,      // const void *def_value,
                   NULL,                          // H5P_prp_create_func_t prp_create,
                   NULL,                          // H5P_prp_set_func_t prp_set,
                   NULL,                          // H5P_prp_get_func_t prp_get,
                   H5D_XFER_BKGR_BUF_TYPE_ENC,    // H5P_prp_encode_func_t prp_encode,
                   H5D_XFER_BKGR_BUF_TYPE_DEC,    // H5P_prp_decode_func_t prp_decode,
                   NULL,                          // H5P_prp_delete_func_t prp_delete,
                   NULL,                          // H5P_prp_copy_func_t prp_copy,
                   NULL,                          // H5P_prp_compare_func_t prp_cmp,
                   NULL);                         // H5P_prp_close_func_t prp_close
```

Observe that only the encode and decode callbacks are defined.

H5D_XFER_BKGR_BUF_TYPE_ENC and H5D_XFER_BKGR_BUF_TYPE_DEC resolve to H5P__dxfr_bkgr_buf_type_enc() and H5P__dxfr_bkgr_buf_type_dec() respectively. Their signatures appear in H5Pdxpl.c, and are reproduced below:

```
herr_t H5P__dxfr_bkgr_buf_type_enc(const void *value, void **pp, size_t *size);

herr_t H5P__dxfr_bkgr_buf_type_dec(const void **pp, void *value);
```

These two routines appear to have no potential for multi-thread issues outside of variables pointed to by their parameters. As it is assumed that appropriate mutual exclusion must be maintained on the property, the only remaining potential area of concern is the encode / decode buffer.

B-Tree node splitting ratios property

The call used to register the B-Tree node splitting property is reproduced below with the formal parameters added as comments

```
H5P__register_real(pclass                          // H5P_genclass_t *pclass,
                H5D_XFER_BTREE_SPLIT_RATIO_NAME, // const char *name,
                H5D_XFER_BTREE_SPLIT_RATIO_SIZE,// size_t size,
                H5D_def_btree_split_ratio_g,    // const void *def_value,
                NULL,                           // H5P_prp_create_func_t prp_create,
                NULL,                           // H5P_prp_set_func_t prp_set,
                NULL,                           // H5P_prp_get_func_t prp_get,
                H5D_XFER_BTREE_SPLIT_RATIO_ENC, // H5P_prp_encode_func_t prp_encode,
                H5D_XFER_BTREE_SPLIT_RATIO_DEC, // H5P_prp_decode_func_t prp_decode,
                NULL,                           // H5P_prp_delete_func_t prp_delete,
                NULL,                           // H5P_prp_copy_func_t prp_copy,
                NULL,                           // H5P_prp_compare_func_t prp_cmp,
                NULL);                          // H5P_prp_close_func_t prp_close
```

Observe that only the encode and decode callbacks are defined.

H5D_XFER_BTREE_SPLIT_RATIO_ENC and H5D_XFER_BTREE_SPLIT_RATIO_DEC resolve to H5P__dxfr_btree_split_ratio_enc() and H5P__dxfr_btree_split_ratio_dec() respectively. Their signatures appear in H5Pdxpl.c, and are reproduced below:

```
herr_t H5P__dxfr_btree_split_ratio_enc(const void *value, void **pp, size_t *size);

herr_t H5P__dxfr_btree_split_ratio_dec(const void **pp, void *value);
```

These two routines appear to have no potential for multi-thread issues outside of variables pointed to by their parameters. As it is assumed that appropriate mutual exclusion must be maintained on the property, the only remaining potential area of concern is the encode / decode buffer.

Vlen allocation function property

The call used to register the vlen allocation function property is reproduced below with the formal parameters added as comments

```
H5P__register_real(pclass                       // H5P_genclass_t *pclass,
                   H5D_XFER_VLEN_ALLOC_NAME,    // const char *name,
                   H5D_XFER_VLEN_ALLOC_SIZE,    // size_t size,
                   &H5D_def_vlen_alloc_g,       // const void *def_value,
                   NULL,                        // H5P_prp_create_func_t prp_create,
                   NULL,                        // H5P_prp_set_func_t prp_set,
                   NULL,                        // H5P_prp_get_func_t prp_get,
                   NULL,                        // H5P_prp_encode_func_t prp_encode,
                   NULL,                        // H5P_prp_decode_func_t prp_decode,
                   NULL,                        // H5P_prp_delete_func_t prp_delete,
                   NULL,                        // H5P_prp_copy_func_t prp_copy,
                   NULL,                        // H5P_prp_compare_func_t prp_cmp,
                   NULL);                       // H5P_prp_close_func_t prp_close
```

Since no callbacks are defined., there are no callback related multi-thread safety issues with this property.

Vlen allocation information property

The call used to register the vlen allocation information property is reproduced below with the formal parameters added as comments

```
H5P__register_real(pclass                            // H5P_genclass_t *pclass,
                   H5D_XFER_VLEN_ALLOC_INFO_NAME, // const char *name,
                   H5D_XFER_VLEN_ALLOC_INFO_SIZE, // size_t size,
                   &H5D_def_vlen_alloc_info_g,    // const void *def_value,
                   NULL,                          // H5P_prp_create_func_t prp_create,
                   NULL,                          // H5P_prp_set_func_t prp_set,
                   NULL,                          // H5P_prp_get_func_t prp_get,
                   NULL,                          // H5P_prp_encode_func_t prp_encode,
                   NULL,                          // H5P_prp_decode_func_t prp_decode,
                   NULL,                          // H5P_prp_delete_func_t prp_delete,
                   NULL,                          // H5P_prp_copy_func_t prp_copy,
                   NULL,                          // H5P_prp_compare_func_t prp_cmp,
                   NULL);                         // H5P_prp_close_func_t prp_close
```

Since no callbacks are defined., there are no callback related multi-thread safety issues with this property.

Vlen free function property

The call used to register the vlen free function property is reproduced below with the formal parameters added as comments

```
H5P__register_real(pclass                            // H5P_genclass_t *pclass,
                   H5D_XFER_VLEN_FREE_NAME, // const char *name,
                   H5D_XFER_VLEN_FREE_SIZE, // size_t size,
                   &H5D_def_vlen_free_g,    // const void *def_value,
                   NULL,                          // H5P_prp_create_func_t prp_create,
                   NULL,                          // H5P_prp_set_func_t prp_set,
                   NULL,                          // H5P_prp_get_func_t prp_get,
                   NULL,                          // H5P_prp_encode_func_t prp_encode,
                   NULL,                          // H5P_prp_decode_func_t prp_decode,
                   NULL,                          // H5P_prp_delete_func_t prp_delete,
                   NULL,                          // H5P_prp_copy_func_t prp_copy,
                   NULL,                          // H5P_prp_compare_func_t prp_cmp,
                   NULL);                         // H5P_prp_close_func_t prp_close
```

Since no callbacks are defined., there are no callback related multi-thread safety issues with this property.

Vlen free information property

The call used to register the vlen free information property is reproduced below with the formal parameters added as comments

```
H5P__register_real(pclass                            // H5P_genclass_t *pclass,
                   H5D_XFER_VLEN_FREE_INFO_NAME, // const char *name,
                   H5D_XFER_VLEN_FREE_INFO_SIZE, // size_t size,
                   &H5D_def_vlen_free_info_g,    // const void *def_value,
```

```
                NULL,                          // H5P_prp_create_func_t prp_create,
                NULL,                          // H5P_prp_set_func_t prp_set,
                NULL,                          // H5P_prp_get_func_t prp_get,
                NULL,                          // H5P_prp_encode_func_t prp_encode,
                NULL,                          // H5P_prp_decode_func_t prp_decode,
                NULL,                          // H5P_prp_delete_func_t prp_delete,
                NULL,                          // H5P_prp_copy_func_t prp_copy,
                NULL,                          // H5P_prp_compare_func_t prp_cmp,
                NULL);                         // H5P_prp_close_func_t prp_close
```

Since no callbacks are defined., there are no callback related multi-thread safety issues with this property.

Vector size property

The call used to register the vector size property is reproduced below with the formal parameters added as comments

```
H5P__register_real(pclass                       // H5P_genclass_t *pclass,
                H5D_XFER_HYPER_VECTOR_SIZE_NAME,// const char *name,
                H5D_XFER_HYPER_VECTOR_SIZE_SIZE,// size_t size,
                &H5D_def_hyp_vec_size_g,        // const void *def_value,
                NULL,                           // H5P_prp_create_func_t prp_create,
                NULL,                           // H5P_prp_set_func_t prp_set,
                NULL,                           // H5P_prp_get_func_t prp_get,
                H5D_XFER_HYPER_VECTOR_SIZE_ENC, // H5P_prp_encode_func_t prp_encode,
                H5D_XFER_HYPER_VECTOR_SIZE_DEC, // H5P_prp_decode_func_t prp_decode,
                NULL,                           // H5P_prp_delete_func_t prp_delete,
                NULL,                           // H5P_prp_copy_func_t prp_copy,
                NULL,                           // H5P_prp_compare_func_t prp_cmp,
                NULL);                          // H5P_prp_close_func_t prp_close
```

Observe that only the encode and decode callbacks are defined.

H5D_XFER_HYPER_VECTOR_SIZE_ENC and H5D_XFER_HYPER_VECTOR_SIZE_DEC resolve to H5P__encode_size_t() and H5P__decode_size_t() respectively. Their signatures appear in H5Ppkg.h, and are reproduced below:

```
herr_t H5P__encode_size_t(const void *value, void **_pp, size_t *size);

herr_t H5P__decode_size_t(const void **_pp, void *_value);
```

These two routines appear to have no potential for multi-thread issues outside of variables pointed to by their parameters. As it is assumed that appropriate mutual exclusion must be maintained on the property, the only remaining potential area of concern is the encode / decode buffer.

I/O transfer mode property (H5D_XFER_IO_XFER_MODE)

The call used to register the I/O transfer mode property (H5D_XFER_IO_XFER_MODE) is reproduced below with the formal parameters added as comments

```
H5P__register_real(pclass                          // H5P_genclass_t *pclass,
                H5D_XFER_IO_XFER_MODE_NAME,        // const char *name,
                H5D_XFER_IO_XFER_MODE_SIZE,        // size_t size,
                &H5D_def_io_xfer_mode_g,           // const void *def_value,
                NULL,                              // H5P_prp_create_func_t prp_create,
                NULL,                              // H5P_prp_set_func_t prp_set,
                NULL,                              // H5P_prp_get_func_t prp_get,
                H5D_XFER_IO_XFER_MODE_ENC,         // H5P_prp_encode_func_t prp_encode,
                H5D_XFER_IO_XFER_MODE_DEC,         // H5P_prp_decode_func_t prp_decode,
                NULL,                              // H5P_prp_delete_func_t prp_delete,
                NULL,                              // H5P_prp_copy_func_t prp_copy,
                NULL,                              // H5P_prp_compare_func_t prp_cmp,
                NULL);                             // H5P_prp_close_func_t prp_close
```

Observe that only the encode and decode callbacks are defined.

H5D_XFER_IO_XFER_MODE_ENC and H5D_XFER_IO_XFER_MODE_DEC resolve to H5P__dxfr_io_xfer_mode_enc() and H5P__dxfr_io_xfer_mode_dec() respectively. Their signatures appear in H5Pdxpl.c, and are reproduced below:

```
herr_t H5P__dxfr_io_xfer_mode_enc(const void *value, void **pp, size_t *size);

herr_t H5P__dxfr_io_xfer_mode_dec(const void **pp, void *value);
```

These two routines appear to have no potential for multi-thread issues outside of variables pointed to by their parameters. As it is assumed that appropriate mutual exclusion must be maintained on the property, the only remaining potential area of concern is the encode / decode buffer.

I/O transfer mode property (H5D_XFER_MPIO_COLLECTIVE_OPT)

The call used to register the I/O transfer mode property (MPI Collective Optimization) is reproduced below with the formal parameters added as comments

```
H5P__register_real(pclass                            // H5P_genclass_t *pclass,
              H5D_XFER_MPIO_COLLECTIVE_OPT_NAME,     // const char *name,
              H5D_XFER_MPIO_COLLECTIVE_OPT_SIZE,     // size_t size,
              &H5D_def_mpio_collective_opt_mode_g,// const void *def_value,
              NULL,                                  // H5P_prp_create_func_t prp_create,
              NULL,                                  // H5P_prp_set_func_t prp_set,
              NULL,                                  // H5P_prp_get_func_t prp_get,
              H5D_XFER_MPIO_COLLECTIVE_OPT_ENC,      // H5P_prp_encode_func_t prp_encode,
              H5D_XFER_MPIO_COLLECTIVE_OPT_DEC,      // H5P_prp_decode_func_t prp_decode,
              NULL,                                  // H5P_prp_delete_func_t prp_delete,
              NULL,                                  // H5P_prp_copy_func_t prp_copy,
              NULL,                                  // H5P_prp_compare_func_t prp_cmp,
              NULL);                                 // H5P_prp_close_func_t prp_close
```

Observe that only the encode and decode callbacks are defined.

H5D_XFER_MPIO_COLLECTIVE_OPT_ENC and H5D_XFER_MPIO_COLLECTIVE_OPT_DEC resolve to H5P__dxfr_mpio_collective_opt_enc() and H5P__dxfr_mpio_collective_opt_dec() respectively. Their signatures appear in H5Pdxpl.c, and are reproduced below:

```
herr_t H5P__dxfr_mpio_collective_opt_enc(const void *value, void **pp, size_t *size);
herr_t H5P__dxfr_mpio_collective_opt_dec(const void **pp, void *value);
```

These two routines appear to have no potential for multi-thread issues outside of the variables pointed to by their parameters. As it is assumed that appropriate mutual exclusion must be maintained on the property, the only remaining potential area of concern is the encode / decode buffer.

I/O transfer mode property (H5D_XFER_MPIO_CHUNK_OPT_HARD)

The call used to register the I/O transfer mode property (H5D_XFER_MPIO_CHUNK_OPT_HARD) is reproduced below with the formal parameters added as comments

```
H5P__register_real(pclass                          // H5P_genclass_t *pclass,
              H5D_XFER_MPIO_CHUNK_OPT_HARD_NAME,  // const char *name,
              H5D_XFER_MPIO_CHUNK_OPT_HARD_SIZE,  // size_t size,
              &H5D_def_mpio_chunk_opt_mode_g,     // const void *def_value,
              NULL,                               // H5P_prp_create_func_t prp_create,
              NULL,                               // H5P_prp_set_func_t prp_set,
              NULL,                               // H5P_prp_get_func_t prp_get,
              H5D_XFER_MPIO_CHUNK_OPT_HARD_ENC,   // H5P_prp_encode_func_t prp_encode,
              H5D_XFER_MPIO_CHUNK_OPT_HARD_DEC,   // H5P_prp_decode_func_t prp_decode,
              NULL,                               // H5P_prp_delete_func_t prp_delete,
              NULL,                               // H5P_prp_copy_func_t prp_copy,
              NULL,                               // H5P_prp_compare_func_t prp_cmp,
              NULL);                              // H5P_prp_close_func_t prp_close
```

Observe that only the encode and decode callbacks are defined.

H5D_XFER_MPIO_CHUNK_OPT_HARD_ENC and H5D_XFER_MPIO_CHUNK_OPT_HARD_DEC resolve to H5P__dxfr_mpio_chunk_opt_hard_enc() and H5P__dxfr_mpio_chunk_opt_hard_dec() respectively. Their signatures appear in H5Pdxpl.c, and are reproduced below:

herr_t H5P__dxfr_mpio_chunk_opt_hard_enc(const void *value, void **pp, size_t *size);

herr_t H5P__dxfr_mpio_chunk_opt_hard_dec(const void **pp, void *value);

These two routines appear to have no potential for multi-thread issues outside of the variables pointed to by their parameters. As it is assumed that appropriate mutual exclusion must be maintained on the property, the only remaining potential area of concern is the encode / decode buffer.

I/O transfer mode property(H5D_XFER_MPIO_CHUNK_OPT_NUM)

The call used to register the I/O transfer mode property (H5D_XFER_MPIO_CHUNK_OPT_NUM) is reproduced below with the formal parameters added as comments

```
H5P__register_real(pclass                          // H5P_genclass_t *pclass,
              H5D_XFER_MPIO_CHUNK_OPT_NUM_NAME,   // const char *name,
              H5D_XFER_MPIO_CHUNK_OPT_NUM_SIZE,   // size_t size,
              &H5D_def_mpio_chunk_opt_num_g,      // const void *def_value,
```

```
            NULL,                               // H5P_prp_create_func_t prp_create,
            NULL,                               // H5P_prp_set_func_t prp_set,
            NULL,                               // H5P_prp_get_func_t prp_get,
            H5D_XFER_MPIO_CHUNK_OPT_NUM_ENC,    // H5P_prp_encode_func_t prp_encode,
            H5D_XFER_MPIO_CHUNK_OPT_NUM_DEC,    // H5P_prp_decode_func_t prp_decode,
            NULL,                               // H5P_prp_delete_func_t prp_delete,
            NULL,                               // H5P_prp_copy_func_t prp_copy,
            NULL,                               // H5P_prp_compare_func_t prp_cmp,
            NULL);                              // H5P_prp_close_func_t prp_close
```

Observe that only the encode and decode callbacks are defined.

H5D_XFER_MPIO_CHUNK_OPT_NUM_ENC and H5D_XFER_MPIO_CHUNK_OPT_NUM_DEC resolve to H5P__encode_unsigned() and H5P__decode_unsigned() respectively. Their signatures appear in H5Ppkg.h, and are reproduced below:

```
herr_t H5P__encode_unsigned(const void *value, void **_pp, size_t *size);

herr_t H5P__decode_unsigned(const void **_pp, void *value);
```

These two routines appear to have no potential for multi-thread issues outside of the variables pointed to by their parameters. As it is assumed that appropriate mutual exclusion must be maintained on the property, the only remaining potential area of concern is the encode / decode buffer.

I/O transfer mode property (H5D_XFER_MPIO_CHUNK_OPT_RATIO)

he call used to register the I/O transfer mode property (H5D_XFER_MPIO_CHUNK_OPT_RATIO) is reproduced below with the formal parameters added as comments

```
H5P__register_real(pclass                          // H5P_genclass_t *pclass,
            H5D_XFER_MPIO_CHUNK_OPT_RATIO_NAME, // const char *name,
            H5D_XFER_MPIO_CHUNK_OPT_RATIO_SIZE, // size_t size,
            &H5D_def_mpio_chunk_opt_ratio_g,    // const void *def_value,
            NULL,                               // H5P_prp_create_func_t prp_create,
            NULL,                               // H5P_prp_set_func_t prp_set,
            NULL,                               // H5P_prp_get_func_t prp_get,
            H5D_XFER_MPIO_CHUNK_OPT_RATIO_ENC,  // H5P_prp_encode_func_t prp_encode,
            H5D_XFER_MPIO_CHUNK_OPT_RATIO_DEC,  // H5P_prp_decode_func_t prp_decode,
            NULL,                               // H5P_prp_delete_func_t prp_delete,
            NULL,                               // H5P_prp_copy_func_t prp_copy,
            NULL,                               // H5P_prp_compare_func_t prp_cmp,
            NULL);                              // H5P_prp_close_func_t prp_close
```

Observe that only the encode and decode callbacks are defined.

H5D_XFER_MPIO_CHUNK_OPT_RATIO_ENC and H5D_XFER_MPIO_CHUNK_OPT_RATIO_DEC resolve to H5P__encode_unsigned() and H5P__decode_unsigned() respectively. Their signatures appear in H5Pdxpl.c, and are reproduced below:

```
herr_t H5P__encode_unsigned(const void *value, void **_pp, size_t *size);

herr_t H5P__decode_unsigned(const void **_pp, void *value);
```

These two routines appear to have no potential for multi-thread issues outside of the variables pointed to by their parameters. As it is assumed that appropriate mutual exclusion must be maintained on the property, the only remaining potential area of concern is the encode / decode buffer.

Chunk Optimization mode property

The call used to register the chunk optimization mode property is reproduced below with the formal parameters added as comments

```
H5P__register_real(pclass                              // H5P_genclass_t *pclass,
          H5D_MPIO_ACTUAL_CHUNK_OPT_MODE_NAME,         // const char *name,
          H5D_MPIO_ACTUAL_CHUNK_OPT_MODE_SIZE,         // size_t size,
          &H5D_def_mpio_actual_chunk_opt_mode_g,       // const void *def_value,
          NULL,                                        // H5P_prp_create_func_t prp_create,
          NULL,                                        // H5P_prp_set_func_t prp_set,
          NULL,                                        // H5P_prp_get_func_t prp_get,
          NULL,                                        // H5P_prp_encode_func_t prp_encode,
          NULL,                                        // H5P_prp_decode_func_t prp_decode,
          NULL,                                        // H5P_prp_delete_func_t prp_delete,
          NULL,                                        // H5P_prp_copy_func_t prp_copy,
          NULL,                                        // H5P_prp_compare_func_t prp_cmp,
          NULL);                                       // H5P_prp_close_func_t prp_close
```

Since no callbacks are defined., there are no callback related multi-thread safety issues with this property.

Actual I/O mode property

The call used to register the actual I/O mode property is reproduced below with the formal parameters added as comments

```
H5P__register_real(pclass                              // H5P_genclass_t *pclass,
          H5D_MPIO_ACTUAL_IO_MODE_NAME,                // const char *name,
          H5D_MPIO_ACTUAL_IO_MODE_SIZE,                // size_t size,
          &H5D_def_mpio_actual_io_mode_g,              // const void *def_value,
          NULL,                                        // H5P_prp_create_func_t prp_create,
          NULL,                                        // H5P_prp_set_func_t prp_set,
          NULL,                                        // H5P_prp_get_func_t prp_get,
          NULL,                                        // H5P_prp_encode_func_t prp_encode,
          NULL,                                        // H5P_prp_decode_func_t prp_decode,
          NULL,                                        // H5P_prp_delete_func_t prp_delete,
          NULL,                                        // H5P_prp_copy_func_t prp_copy,
          NULL,                                        // H5P_prp_compare_func_t prp_cmp,
          NULL);                                       // H5P_prp_close_func_t prp_close
```

Since no callbacks are defined., there are no callback related multi-thread safety issues with this property.

Local cause of broken collective I/O property

The call used to register the local cause of broken collective I/O property is reproduced below with the formal parameters added as comments

```
H5P__register_real(pclass                              // H5P_genclass_t *pclass,
```

```
        H5D_MPIO_LOCAL_NO_COLLECTIVE_CAUSE_NAME, // const char *name,
        H5D_MPIO_NO_COLLECTIVE_CAUSE_SIZE,       // size_t size,
        &H5D_def_mpio_no_collective_cause_g,     // const void *def_value,
        NULL,                                    // H5P_prp_create_func_t prp_create,
        NULL,                                    // H5P_prp_set_func_t prp_set,
        NULL,                                    // H5P_prp_get_func_t prp_get,
        NULL,                                    // H5P_prp_encode_func_t prp_encode,
        NULL,                                    // H5P_prp_decode_func_t prp_decode,
        NULL,                                    // H5P_prp_delete_func_t prp_delete,
        NULL,                                    // H5P_prp_copy_func_t prp_copy,
        NULL,                                    // H5P_prp_compare_func_t prp_cmp,
        NULL);                                   // H5P_prp_close_func_t prp_close
```

Since no callbacks are defined., there are no callback related multi-thread safety issues with this property.

Global cause of broken collective I/O property

The call used to register the global cause of broken collective I/O property is reproduced below with the formal parameters added as comments

```
H5P__register_real(pclass                        // H5P_genclass_t *pclass,
        H5D_MPIO_GLOBAL_NO_COLLECTIVE_CAUSE_NAME, // const char *name,
        H5D_MPIO_NO_COLLECTIVE_CAUSE_SIZE,       // size_t size,
        &H5D_def_mpio_no_collective_cause_g,     // const void *def_value,
        NULL,                                    // H5P_prp_create_func_t prp_create,
        NULL,                                    // H5P_prp_set_func_t prp_set,
        NULL,                                    // H5P_prp_get_func_t prp_get,
        NULL,                                    // H5P_prp_encode_func_t prp_encode,
        NULL,                                    // H5P_prp_decode_func_t prp_decode,
        NULL,                                    // H5P_prp_delete_func_t prp_delete,
        NULL,                                    // H5P_prp_copy_func_t prp_copy,
        NULL,                                    // H5P_prp_compare_func_t prp_cmp,
        NULL);                                   // H5P_prp_close_func_t prp_close
```

Since no callbacks are defined., there are no callback related multi-thread safety issues with this property.

EDC property

The call used to register the EDC property is reproduced below with the formal parameters added as comments

```
H5P__register_real(pclass                        // H5P_genclass_t *pclass,
        H5D_XFER_EDC_NAME,                       // const char *name,
        H5D_XFER_EDC_SIZE,                       // size_t size,
        &H5D_def_enable_edc_g,                   // const void *def_value,
        NULL,                                    // H5P_prp_create_func_t prp_create,
        NULL,                                    // H5P_prp_set_func_t prp_set,
        NULL,                                    // H5P_prp_get_func_t prp_get,
        H5D_XFER_EDC_ENC,                        // H5P_prp_encode_func_t prp_encode,
        H5D_XFER_EDC_DEC,                        // H5P_prp_decode_func_t prp_decode,
        NULL,                                    // H5P_prp_delete_func_t prp_delete,
        NULL,                                    // H5P_prp_copy_func_t prp_copy,
        NULL,                                    // H5P_prp_compare_func_t prp_cmp,
        NULL);                                   // H5P_prp_close_func_t prp_close
```

Observe that only the encode and decode callbacks are defined.

H5D_XFER_EDC_ENC and H5D_XFER_EDC_DEC resolve to H5P__dxfr_edc_enc() and H5P__dxfr_edc_dec() respectively. Their signatures appear in H5Pdxpl.c, and are reproduced below:

```
herr_t H5P__dxfr_edc_enc(const void *value, void **pp, size_t *size)

herr_t H5P__dxfr_edc_dec(const void **pp, void *value);
```

These two routines appear to have no potential for multi-thread issues outside of the variables pointed to by their parameters. As it is assumed that appropriate mutual exclusion must be maintained on the property, the only remaining potential area of concern is the encode / decode buffer.

Filter callback property

The call used to register the filter callback property is reproduced below with the formal parameters added as comments

```
H5P__register_real(pclass                          // H5P_genclass_t *pclass,
        H5D_XFER_FILTER_CB_NAME,                   // const char *name,
        H5D_XFER_FILTER_CB_SIZE,                   // size_t size,
        &H5D_def_filter_cb_g,                       // const void *def_value,
        NULL,                                      // H5P_prp_create_func_t prp_create,
        NULL,                                      // H5P_prp_set_func_t prp_set,
        NULL,                                      // H5P_prp_get_func_t prp_get,
        NULL,                                      // H5P_prp_encode_func_t prp_encode,
        NULL,                                      // H5P_prp_decode_func_t prp_decode,
        NULL,                                      // H5P_prp_delete_func_t prp_delete,
        NULL,                                      // H5P_prp_copy_func_t prp_copy,
        NULL,                                      // H5P_prp_compare_func_t prp_cmp,
        NULL);                                     // H5P_prp_close_func_t prp_close
```

Since no callbacks are defined., there are no callback related multi-thread safety issues with this property.

Type conversion callback property

The call used to register the type conversion callback property is reproduced below with the formal parameters added as comments

```
H5P__register_real(pclass                          // H5P_genclass_t *pclass,
        H5D_XFER_CONV_CB_NAME,                     // const char *name,
        H5D_XFER_CONV_CB_SIZE,                     // size_t size,
        &H5D_def_conv_cb_g,                        // const void *def_value,
        NULL,                                      // H5P_prp_create_func_t prp_create,
        NULL,                                      // H5P_prp_set_func_t prp_set,
        NULL,                                      // H5P_prp_get_func_t prp_get,
        NULL,                                      // H5P_prp_encode_func_t prp_encode,
        NULL,                                      // H5P_prp_decode_func_t prp_decode,
        NULL,                                      // H5P_prp_delete_func_t prp_delete,
        NULL,                                      // H5P_prp_copy_func_t prp_copy,
        NULL,                                      // H5P_prp_compare_func_t prp_cmp,
        NULL);                                     // H5P_prp_close_func_t prp_close
```

Since no callbacks are defined., there are no callback related multi-thread safety issues with this property.

Data transform property

The call used to register the data transform property is reproduced below with the formal parameters added as comments

```
H5P__register_real(pclass                          // H5P_genclass_t *pclass,
        H5D_XFER_XFORM_NAME,                       // const char *name,
        H5D_XFER_XFORM_SIZE,                       // size_t size,
        &H5D_def_xfer_xform_g,                     // const void *def_value,
        NULL,                                      // H5P_prp_create_func_t prp_create,
        H5D_XFER_XFORM_SET,                        // H5P_prp_set_func_t prp_set,
        H5D_XFER_XFORM_GET,                        // H5P_prp_get_func_t prp_get,
        H5D_XFER_XFORM_ENC,                        // H5P_prp_encode_func_t prp_encode,
        H5D_XFER_XFORM_DEC,                        // H5P_prp_decode_func_t prp_decode,
        H5D_XFER_XFORM_DEL,                        // H5P_prp_delete_func_t prp_delete,
        H5D_XFER_XFORM_COPY,                       // H5P_prp_copy_func_t prp_copy,
        H5D_XFER_XFORM_CMP,                        // H5P_prp_compare_func_t prp_cmp,
        H5D_XFER_XFORM_CLOSE);                     // H5P_prp_close_func_t prp_close
```

With the exception of create, all the property callbacks are set. The following table shows the mapping of the macros to actual function names:

H5D_XFER_XFORM_SET	H5P__dxfr_xform_set()
H5D_XFER_XFORM_GET	H5P__dxfr_xform_get()
H5D_XFER_XFORM_ENC	H5P__dxfr_xform_enc()
H5D_XFER_XFORM_DEC	H5P__dxfr_xform_dec()
H5D_XFER_XFORM_DEL	H5P__dxfr_xform_del()
H5D_XFER_XFORM_COPY	H5P__dxfr_xform_copy()
H5D_XFER_XFORM_CMP	H5P__dxfr_xform_cmp()
H5D_XFER_XFORM_CLOSE	H5P__dxfr_xform_close()

All of the above functions are defined in H5Pdxpl.c. The function signatures are reproduced below:

```
herr_t H5P__dxfr_xform_set(hid_t prop_id, const char *name, size_t size, void *value);
herr_t H5P__dxfr_xform_get(hid_t prop_id, const char *name, size_t size, void *value);
herr_t H5P__dxfr_xform_enc(const void *value, void **pp, size_t *size);
herr_t H5P__dxfr_xform_dec(const void **pp, void *value);
herr_t H5P__dxfr_xform_del(hid_t prop_id, const char *name, size_t size, void *value);
herr_t H5P__dxfr_xform_copy(const char *name, size_t size, void *value);
int    H5P__dxfr_xform_cmp(const void *value1, const void *value2, size_t size);
herr_t H5P__dxfr_xform_close(const char *name, size_t size, void *value);
```

All of these functions make calls into H5Z – must evaluate for thread safety.

Dataset selection property

The call used to register the dataset selection property is reproduced below with the formal parameters added as comments

```
H5P__register_real(pclass                         // H5P_genclass_t *pclass,
        H5D_XFER_DSET_IO_SEL_NAME,                // const char *name,
        H5D_XFER_DSET_IO_SEL_SIZE,                // size_t size,
        &H5D_def_dset_io_sel_g,                   // const void *def_value,
        NULL,                                     // H5P_prp_create_func_t prp_create,
        NULL,                                     // H5P_prp_set_func_t prp_set,
        NULL,                                     // H5P_prp_get_func_t prp_get,
        NULL,                                     // H5P_prp_encode_func_t prp_encode,
        NULL,                                     // H5P_prp_decode_func_t prp_decode,
        NULL,                                     // H5P_prp_delete_func_t prp_delete,
        H5D_XFER_DSET_IO_SEL_COPY,                // H5P_prp_copy_func_t prp_copy,
        H5D_XFER_DSET_IO_SEL_CMP,                 // H5P_prp_compare_func_t prp_cmp,
        H5D_XFER_DSET_IO_SEL_CLOSE);              // H5P_prp_close_func_t prp_close
```

Note that the copy, compare, and close callbacks are set. The following table shows the mapping of the macros to actual function names:

H5D_XFER_DSET_IO_SEL_COPY	H5P__dxfr_dset_io_hyp_sel_copy()
H5D_XFER_DSET_IO_SEL_CMP	H5P__dxfr_dset_io_hyp_sel_cmp()
H5D_XFER_DSET_IO_SEL_CLOSE	H5P__dxfr_dset_io_hyp_sel_close()

All of the above functions are defined in H5Pdxpl.c. The function signatures are reproduced below:

```
herr_t H5P__dxfr_dset_io_hyp_sel_copy(const char *name, size_t size, void *value);
int    H5P__dxfr_dset_io_hyp_sel_cmp(const void *value1, const void *value2, size_t 
size);
herr_t H5P__dxfr_dset_io_hyp_sel_close(const char *name, size_t size, void *value);
```

All of these functions make calls into H5S. While we must make H5S thread safe eventually, that may be a while. We will need some interim solution.

File Access Property List (FAPL) Class (ROOT → FAPL)

As seen from the initialization below:

```
/* File access property list class library initialization object */
const H5P_libclass_t H5P_CLS_FACC[1] = {{
  "file access",        /* Class name for debugging     */
  H5P_TYPE_FILE_ACCESS, /* Class type                   */

  &H5P_CLS_ROOT_g,          /* Parent class                 */
  &H5P_CLS_FILE_ACCESS_g,   /* Pointer to class             */
  &H5P_CLS_FILE_ACCESS_ID_g, /* Pointer to class ID         */
  &H5P_LST_FILE_ACCESS_ID_g, /* Pointer to default property list ID */
  H5P__facc_reg_prop,       /* Default property registration routine */
```

```
  NULL, /* Class creation callback      */
  NULL, /* Class creation callback info */
  NULL, /* Class copy callback          */
  NULL, /* Class copy callback info     */
  NULL, /* Class close callback         */
  NULL  /* Class close callback info    */
}};
```

the File Access Property List Class has no callbacks – and hence no multi-thread safety issues.

File Mount Property List (FMPL) Class (ROOT → FMPL)

As seen from the initialization below:

```
/* File mount property list class library initialization object */
const H5P_libclass_t H5P_CLS_FMNT[1] = {{
  "file mount",        /* Class name for debugging    */
  H5P_TYPE_FILE_MOUNT, /* Class type                  */

  &H5P_CLS_ROOT_g,          /* Parent class               */
  &H5P_CLS_FILE_MOUNT_g,    /* Pointer to class           */
  &H5P_CLS_FILE_MOUNT_ID_g, /* Pointer to class ID        */
  &H5P_LST_FILE_MOUNT_ID_g, /* Pointer to default property list ID */
  H5P__fmnt_reg_prop,       /* Default property registration routine */

  NULL, /* Class creation callback      */
  NULL, /* Class creation callback info */
  NULL, /* Class copy callback          */
  NULL, /* Class copy callback info     */
  NULL, /* Class close callback         */
  NULL  /* Class close callback info    */
}};
```

the File Mount Property List Class has no callbacks – and hence no multi-thread safety issues.

VOL Initialization Property List (VIPL) Class (ROOT ->VIPL)

As seen from the initialization below:

```
/* VOL initialization property list class library initialization object */
/* (move to proper source code file when used for real) */
const H5P_libclass_t H5P_CLS_VINI[1] = {{
  "VOL initialization",    /* Class name for debugging     */
  H5P_TYPE_VOL_INITIALIZE, /* Class type                   */

  &H5P_CLS_ROOT_g,              /* Parent class               */
  &H5P_CLS_VOL_INITIALIZE_g,    /* Pointer to class           */
  &H5P_CLS_VOL_INITIALIZE_ID_g, /* Pointer to class ID        */
  &H5P_LST_VOL_INITIALIZE_ID_g, /* Pointer to default property list ID */
```

```
  NULL,                     /* Default property registration routine */

  NULL, /* Class creation callback      */
  NULL, /* Class creation callback info */
  NULL, /* Class copy callback          */
  NULL, /* Class copy callback info     */
  NULL, /* Class close callback         */
  NULL  /* Class close callback info    */
}};
```

the VOL Initialization Property List Class has no callbacks – and hence no multi-thread safety issues.

Link Access Property List (LAPL) Class (ROOT → LAPL)

As seen from the initialization below:

```
/* Link access property list class library initialization object */
const H5P_libclass_t H5P_CLS_LACC[1] = {{
  "link access",        /* Class name for debugging     */
  H5P_TYPE_LINK_ACCESS, /* Class type                   */

  &H5P_CLS_ROOT_g,          /* Parent class                */
  &H5P_CLS_LINK_ACCESS_g,    /* Pointer to class            */
  &H5P_CLS_LINK_ACCESS_ID_g, /* Pointer to class ID         */
  &H5P_LST_LINK_ACCESS_ID_g, /* Pointer to default property list ID */
  H5P__lacc_reg_prop,       /* Default property registration routine */

  NULL, /* Class creation callback      */
  NULL, /* Class creation callback info */
  NULL, /* Class copy callback          */
  NULL, /* Class copy callback info     */
  NULL, /* Class close callback         */
  NULL  /* Class close callback info    */
}};
```

the Link Access Property List Class has no callbacks – and hence no multi-thread safety issues.

Object Creation Property List (OCRTPL) Class (ROOT → OCRTPL)

As seen from the initialization below:

```
/* Object creation property list class library initialization object */
const H5P_libclass_t H5P_CLS_OCRT[1] = {{
  "object create",        /* Class name for debugging     */
  H5P_TYPE_OBJECT_CREATE, /* Class type                   */
```

```
  &H5P_CLS_ROOT_g,           /* Parent class               */
  &H5P_CLS_OBJECT_CREATE_g,    /* Pointer to class            */
  &H5P_CLS_OBJECT_CREATE_ID_g, /* Pointer to class ID         */
  NULL,                    /* Pointer to default property list ID   */
  H5P__ocrt_reg_prop,        /* Default property registration routine */

  NULL, /* Class creation callback      */
  NULL, /* Class creation callback info */
  NULL, /* Class copy callback          */
  NULL, /* Class copy callback info     */
  NULL, /* Class close callback         */
  NULL  /* Class close callback info    */
}};
```

the Object Creation Property List Class has no callbacks – and hence no multi-thread safety issues.

Object Copy Property List (OCPYPL) Class (ROOT → OCPYPL)

As seen from the initialization below:

```
/* Object copy property list class library initialization object */
const H5P_libclass_t H5P_CLS_OCPY[1] = {{
  "object copy",       /* Class name for debugging      */
  H5P_TYPE_OBJECT_COPY, /* Class type                  */

  &H5P_CLS_ROOT_g,          /* Parent class              */
  &H5P_CLS_OBJECT_COPY_g,    /* Pointer to class           */
  &H5P_CLS_OBJECT_COPY_ID_g, /* Pointer to class ID        */
  &H5P_LST_OBJECT_COPY_ID_g, /* Pointer to default property list ID */
  H5P__ocpy_reg_prop,       /* Default property registration routine */

  NULL, /* Class creation callback      */
  NULL, /* Class creation callback info */
  NULL, /* Class copy callback          */
  NULL, /* Class copy callback info     */
  NULL, /* Class close callback         */
  NULL  /* Class close callback info    */
}};
```

the Object Copy Property List Class has no callbacks – and hence no multi-thread safety issues.

String Creation Property List (STRCRTPL) Class (ROOT→ STRCRTPL)

As seen from the initialization below:

```
/* String creation property list class library initialization object */
const H5P_libclass_t H5P_CLS_STRCRT[1] = {{
  "string create",      /* Class name for debugging     */
  H5P_TYPE_STRING_CREATE, /* Class type                 */

  &H5P_CLS_ROOT_g,          /* Parent class               */
  &H5P_CLS_STRING_CREATE_g,   /* Pointer to class           */
  &H5P_CLS_STRING_CREATE_ID_g, /* Pointer to class ID        */
  NULL,                  /* Pointer to default property list ID */
  H5P__strcrt_reg_prop,      /* Default property registration routine */

  NULL, /* Class creation callback      */
  NULL, /* Class creation callback info */
  NULL, /* Class copy callback          */
  NULL, /* Class copy callback info     */
  NULL, /* Class close callback         */
  NULL  /* Class close callback info    */
}};
```

the String Creation Property List Class has no callbacks – and hence no multi-thread safety issues.

Datatype Access Property List (TAPL) Class (ROOT → LAPL → TAPL)

As seen from the initialization below:

```
/* Datatype access property list class library initialization object */
/* (move to proper source code file when used for real) */
const H5P_libclass_t H5P_CLS_TACC[1] = {{
  "datatype access",      /* Class name for debugging     */
  H5P_TYPE_DATATYPE_ACCESS, /* Class type                 */

  &H5P_CLS_LINK_ACCESS_g,      /* Parent class               */
  &H5P_CLS_DATATYPE_ACCESS_g,   /* Pointer to class           */
  &H5P_CLS_DATATYPE_ACCESS_ID_g, /* Pointer to class ID        */
  &H5P_LST_DATATYPE_ACCESS_ID_g, /* Pointer to default property list ID */
  NULL,                    /* Default property registration routine */

  NULL, /* Class creation callback      */
  NULL, /* Class creation callback info */
  NULL, /* Class copy callback          */
  NULL, /* Class copy callback info     */
  NULL, /* Class close callback         */
  NULL  /* Class close callback info    */
}};
```

the Datatype Access Property List Class has no callbacks – and hence no multi-thread safety issues.

Attribute Access Property List (AAPL) Class (ROOT → LAPL → AAPL)

As seen from the initialization below:

```
/* Attribute access property list class library initialization object */
/* (move to proper source code file when used for real) */
const H5P_libclass_t H5P_CLS_AACC[1] = {{
  "attribute access",       /* Class name for debugging     */
  H5P_TYPE_ATTRIBUTE_ACCESS, /* Class type                  */

  &H5P_CLS_LINK_ACCESS_g,       /* Parent class              */
  &H5P_CLS_ATTRIBUTE_ACCESS_g,    /* Pointer to class           */
  &H5P_CLS_ATTRIBUTE_ACCESS_ID_g, /* Pointer to class ID         */
  &H5P_LST_ATTRIBUTE_ACCESS_ID_g, /* Pointer to default property list ID */
  NULL,                       /* Default property registration
                             routine */

  NULL, /* Class creation callback      */
  NULL, /* Class creation callback info */
  NULL, /* Class copy callback          */
  NULL, /* Class copy callback info     */
  NULL, /* Class close callback         */
  NULL  /* Class close callback info    */
}};
```

the Attribute Access Property List Class has no callbacks – and hence no multi-thread safety issues.

Group Access Property List (GAPL) Class (ROOT → LAPL → GAPL)

As seen from the initialization below:

```
/* Group access property list class library initialization object */
/* (move to proper source code file when used for real) */
const H5P_libclass_t H5P_CLS_GACC[1] = {{
  "group access",       /* Class name for debugging     */
  H5P_TYPE_GROUP_ACCESS, /* Class type                  */

  &H5P_CLS_LINK_ACCESS_g,     /* Parent class              */
  &H5P_CLS_GROUP_ACCESS_g,    /* Pointer to class           */
  &H5P_CLS_GROUP_ACCESS_ID_g, /* Pointer to class ID         */
  &H5P_LST_GROUP_ACCESS_ID_g, /* Pointer to default property list ID */
  NULL,                   /* Default property registration routine */
```

```
  NULL, /* Class creation callback      */
  NULL, /* Class creation callback info */
  NULL, /* Class copy callback          */
  NULL, /* Class copy callback info     */
  NULL, /* Class close callback         */
  NULL  /* Class close callback info    */
}};
```

the Group Access Property List Class has no callbacks – and hence no multi-thread safety issues.

Datatype Creation Property List (TCPL) Class (ROOT → OCPL → TCPL)

As seen from the initialization below:

```
/* Datatype creation property list class library initialization object */
/* (move to proper source code file when used for real) */
const H5P_libclass_t H5P_CLS_TCRT[1] = {{
  "datatype create",        /* Class name for debugging     */
  H5P_TYPE_DATATYPE_CREATE, /* Class type                   */

  &H5P_CLS_OBJECT_CREATE_g,      /* Parent class              */
  &H5P_CLS_DATATYPE_CREATE_g,    /* Pointer to class          */
  &H5P_CLS_DATATYPE_CREATE_ID_g, /* Pointer to class ID       */
  &H5P_LST_DATATYPE_CREATE_ID_g, /* Pointer to default property list ID */
  NULL,                          /* Default property registration routine */

  NULL, /* Class creation callback      */
  NULL, /* Class creation callback info */
  NULL, /* Class copy callback          */
  NULL, /* Class copy callback info     */
  NULL, /* Class close callback         */
  NULL  /* Class close callback info    */
}};
```

the Datatype Creation Property List Class has no callbacks – and hence no multi-thread safety issues.

Dataset Creation Property List (DCRTPL) Class (ROOT → OCPL → DCRTPL)

As seen from the initialization below:

```
/* Dataset creation property list class library initialization object */
const H5P_libclass_t H5P_CLS_DCRT[1] = {{
  "dataset create",       /* Class name for debugging     */
  H5P_TYPE_DATASET_CREATE, /* Class type                  */

  &H5P_CLS_OBJECT_CREATE_g,     /* Parent class              */
  &H5P_CLS_DATASET_CREATE_g,    /* Pointer to class          */
  &H5P_CLS_DATASET_CREATE_ID_g, /* Pointer to class ID       */
  &H5P_LST_DATASET_CREATE_ID_g, /* Pointer to default property list ID */
  H5P__dcrt_reg_prop,         /* Default property registration routine */

  NULL, /* Class creation callback      */
  NULL, /* Class creation callback info */
  NULL, /* Class copy callback          */
  NULL, /* Class copy callback info     */
  NULL, /* Class close callback         */
  NULL  /* Class close callback info    */
}};
```

the Dataset Creation Property List Class has no callbacks – and hence no multi-thread safety issues.

Dataset Access Property List (DAPL) Class (ROOT → LAPL → DAPL)

As seen from the initialization below:

```
/* Dataset access property list class library initialization object */
const H5P_libclass_t H5P_CLS_DACC[1] = {{
  "dataset access",       /* Class name for debugging     */
  H5P_TYPE_DATASET_ACCESS, /* Class type                  */

  &H5P_CLS_LINK_ACCESS_g,      /* Parent class              */
  &H5P_CLS_DATASET_ACCESS_g,    /* Pointer to class          */
  &H5P_CLS_DATASET_ACCESS_ID_g, /* Pointer to class ID       */
  &H5P_LST_DATASET_ACCESS_ID_g, /* Pointer to default property list ID */
  H5P__dacc_reg_prop,         /* Default property registration routine */

  NULL, /* Class creation callback      */
  NULL, /* Class creation callback info */
  NULL, /* Class copy callback          */
  NULL, /* Class copy callback info     */
  NULL, /* Class close callback         */
  NULL  /* Class close callback info    */
}};
```

the Dataset Access Property List Class has no callbacks – and hence no multi-thread safety issues.

Group Creation Property List (GCRTPL) Class (ROOT → OCRTPL → GCRTPL)

As seen from the initialization below:

```
/* Group creation property list class library initialization object */
const H5P_libclass_t H5P_CLS_GCRT[1] = {{
  "group create",       /* Class name for debugging     */
  H5P_TYPE_GROUP_CREATE, /* Class type                  */

  &H5P_CLS_OBJECT_CREATE_g,   /* Parent class              */
  &H5P_CLS_GROUP_CREATE_g,    /* Pointer to class           */
  &H5P_CLS_GROUP_CREATE_ID_g, /* Pointer to class ID        */
  &H5P_LST_GROUP_CREATE_ID_g, /* Pointer to default property list ID */
  H5P__gcrt_reg_prop,        /* Default property registration routine */

  NULL, /* Class creation callback      */
  NULL, /* Class creation callback info */
  NULL, /* Class copy callback          */
  NULL, /* Class copy callback info     */
  NULL, /* Class close callback         */
  NULL  /* Class close callback info    */
}};
```

the Group Creation Property List Class has no callbacks – and hence no multi-thread safety issues.

Attribute Creation Property List (ACRTPL) Class (ROOT → STRCRTPL → ACRTPL)

As seen from the initialization below:

```
/* Attribute creation property list class library initialization object */
const H5P_libclass_t H5P_CLS_ACRT[1] = {{
  "attribute create",       /* Class name for debugging              */
  H5P_TYPE_ATTRIBUTE_CREATE, /* Class type                           */

  &H5P_CLS_STRING_CREATE_g,      /* Parent class                    */
  &H5P_CLS_ATTRIBUTE_CREATE_g,    /* Pointer to class                */
  &H5P_CLS_ATTRIBUTE_CREATE_ID_g, /* Pointer to class ID             */
  &H5P_LST_ATTRIBUTE_CREATE_ID_g, /* Pointer to default property list ID */
  NULL,                        /* Default property registration
```

```
                         routine */

  NULL, /* Class creation callback              */
  NULL, /* Class creation callback info          */
  NULL, /* Class copy callback                  */
  NULL, /* Class copy callback info              */
  NULL, /* Class close callback                 */
  NULL  /* Class close callback info             */
}};
```

the Attribute Creation Property List Class has no callbacks – and hence no multi-thread safety issues.

Link Creation Property List (LCRTPL) Class (ROOT → STRCRTPL → LCRTPL)

As seen from the initialization below:

```
/* Link creation property list class library initialization object */
const H5P_libclass_t H5P_CLS_LCRT[1] = {{
  "link create",       /* Class name for debugging    */
  H5P_TYPE_LINK_CREATE, /* Class type                 */

  &H5P_CLS_STRING_CREATE_g,  /* Parent class              */
  &H5P_CLS_LINK_CREATE_g,    /* Pointer to class          */
  &H5P_CLS_LINK_CREATE_ID_g, /* Pointer to class ID        */
  &H5P_LST_LINK_CREATE_ID_g, /* Pointer to default property list ID */
  H5P__lcrt_reg_prop,       /* Default property registration routine */

  NULL, /* Class creation callback     */
  NULL, /* Class creation callback info */
  NULL, /* Class copy callback         */
  NULL, /* Class copy callback info    */
  NULL, /* Class close callback        */
  NULL  /* Class close callback info    */
}};
```

the Link Creation Property List Class has no callbacks – and hence no multi-thread safety issues.

Map Create Property List (MCRTPL) Class (ROOT → OCRTPL → MCRTPL)

As seen from the initialization below:

```
/* Map create property list class library initialization object */
const H5P_libclass_t H5P_CLS_MCRT[1] = {{
  "map create",       /* Class name for debugging     */
  H5P_TYPE_MAP_CREATE, /* Class type                  */

  &H5P_CLS_OBJECT_CREATE_g, /* Parent class              */
  &H5P_CLS_MAP_CREATE_g,    /* Pointer to class          */
  &H5P_CLS_MAP_CREATE_ID_g, /* Pointer to class ID        */
  &H5P_LST_MAP_CREATE_ID_g, /* Pointer to default property list ID */
  H5P__mcrt_reg_prop,       /* Default property registration routine */

  NULL, /* Class creation callback      */
  NULL, /* Class creation callback info */
  NULL, /* Class copy callback          */
  NULL, /* Class copy callback info     */
  NULL, /* Class close callback         */
  NULL  /* Class close callback info    */
}};
```

the Map Creation Property List Class has no callbacks – and hence no multi-thread safety issues.

Map Access Property List (MAPL) Class (ROOT → LAPL → MAPL)

As seen from the initialization below:

```
/* Map access property list class library initialization object */
const H5P_libclass_t H5P_CLS_MACC[1] = {{
  "map access",       /* Class name for debugging     */
  H5P_TYPE_MAP_ACCESS, /* Class type                  */

  &H5P_CLS_LINK_ACCESS_g,   /* Parent class              */
  &H5P_CLS_MAP_ACCESS_g,    /* Pointer to class          */
  &H5P_CLS_MAP_ACCESS_ID_g, /* Pointer to class ID        */
  &H5P_LST_MAP_ACCESS_ID_g, /* Pointer to default property list ID */
  H5P__macc_reg_prop,       /* Default property registration routine */

  NULL, /* Class creation callback      */
  NULL, /* Class creation callback info */
  NULL, /* Class copy callback          */
  NULL, /* Class copy callback info     */
  NULL, /* Class close callback         */
  NULL  /* Class close callback info    */
}};
```

the Map Access Property List Class has no callbacks – and hence no multi-thread safety issues.

Reference Access Property List (RACCPL) Class (Root → FAPL →RACCPL)

As seen from the initialization below:

```
/* Reference access property list class library initialization object */
/* (move to proper source code file when used for real) */
const H5P_libclass_t H5P_CLS_RACC[1] = {{
  "reference access",       /* Class name for debugging     */
  H5P_TYPE_REFERENCE_ACCESS, /* Class type                  */

  &H5P_CLS_FILE_ACCESS_g,        /* Parent class                    */
  &H5P_CLS_REFERENCE_ACCESS_g,    /* Pointer to class                  */
  &H5P_CLS_REFERENCE_ACCESS_ID_g, /* Pointer to class ID               */
  &H5P_LST_REFERENCE_ACCESS_ID_g, /* Pointer to default property list ID  */
  NULL,                        /* Default property registration routine*/

  NULL, /* Class creation callback            */
  NULL, /* Class creation callback info        */
  NULL, /* Class copy callback                */
  NULL, /* Class copy callback info           */
  NULL, /* Class close callback               */
  NULL  /* Class close callback info          */
}};
```

the Reference Access Property List Class has no callbacks – and hence no multi-thread safety issues.

File Creation Property List (FCRTPL) Class (ROOT → OCRTPL → GCRTPL → FCRTPL)

As seen from the initialization below:

```
/* File creation property list class library initialization object */
const H5P_libclass_t H5P_CLS_FCRT[1] = {{
  "file create",       /* Class name for debugging     */
  H5P_TYPE_FILE_CREATE, /* Class type                  */

  &H5P_CLS_GROUP_CREATE_g,   /* Parent class               */
  &H5P_CLS_FILE_CREATE_g,    /* Pointer to class           */
  &H5P_CLS_FILE_CREATE_ID_g, /* Pointer to class ID        */
  &H5P_LST_FILE_CREATE_ID_g, /* Pointer to default property list ID */
  H5P__fcrt_reg_prop,       /* Default property registration routine */

  NULL, /* Class creation callback      */
```

```
  NULL, /* Class creation callback info */
  NULL, /* Class copy callback         */
  NULL, /* Class copy callback info     */
  NULL, /* Class close callback        */
  NULL  /* Class close callback info    */
}};
```

the File Creation Property List Class has no callbacks – and hence no multi-thread safety issues.

Appendix 5 – Census of H5P Callbacks

The following is a census of H5P callbacks prepared by Mathew Larson – in particular his June 12, 2024 version of this document.

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 aroundH5P__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.

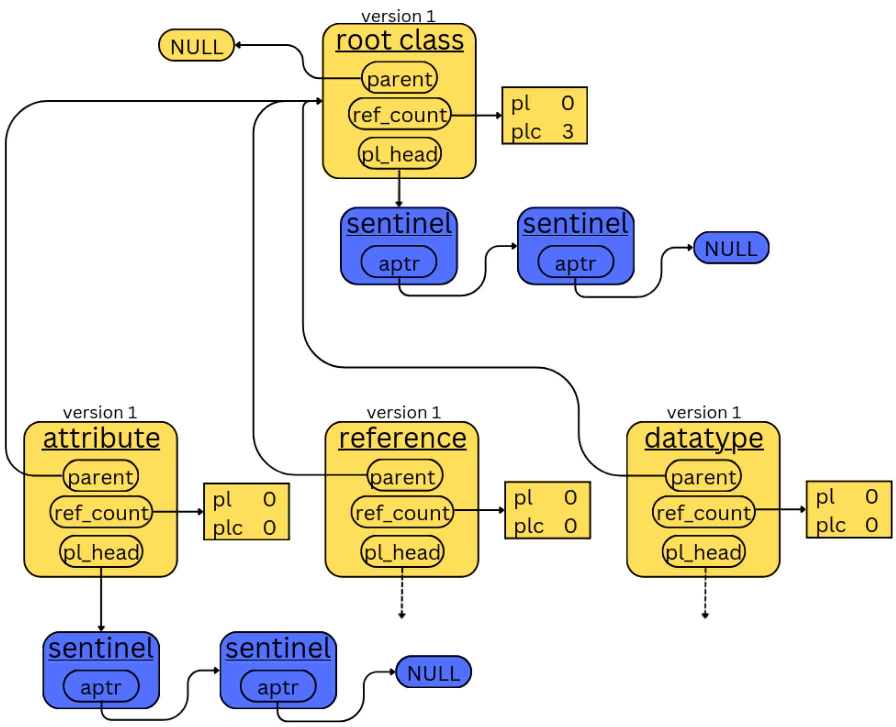
Test Property Class Callbacks

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.

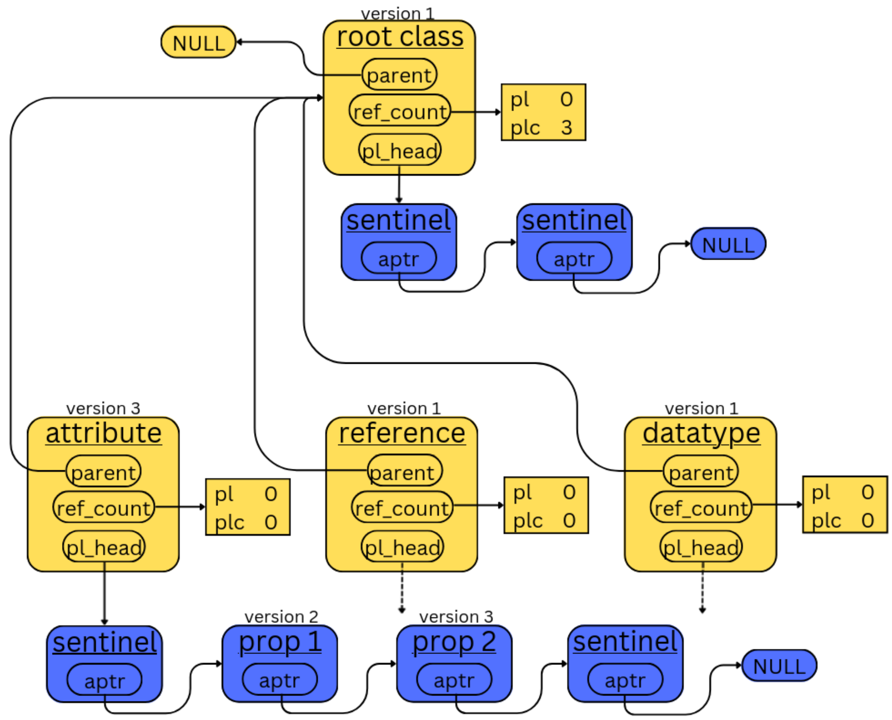
Appendix 6 – Visual Aid for Understanding New Structure Relationship

When a new property list class is created, it is derived from an existing parent class. The properties from the parent class's lock-free singly linked list (LFSLL) are inherited into the new class. The new class also maintains a pointer to the parent class, while the parent class increments its reference count for derived property list classes, plc.

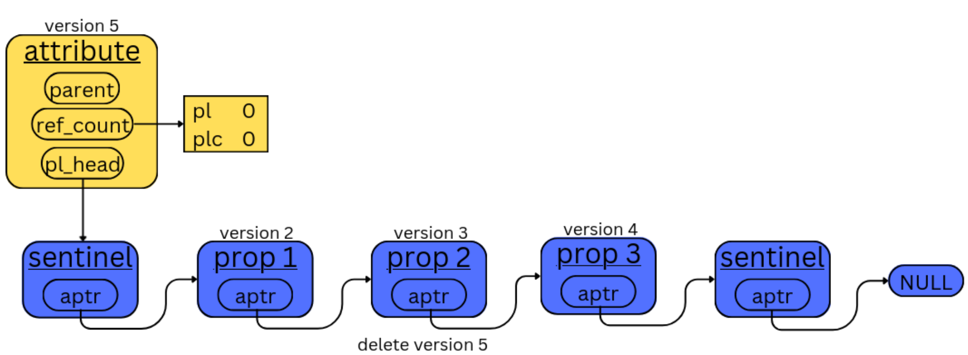


When a property list class is modified–such as updating a property, inserting a new property, or deleting a property in the LFSLL–the property is assigned the class's next version number, and once complete the class updates its current version number to match that next version number. In Image 2, the attribute class is at version 3, reflecting the updates to the versions by inserting property 1 and property 2. When a thread accesses a specific version of a class, it can access any property with a version equal to or older than that specific version of a class, with two exceptions:

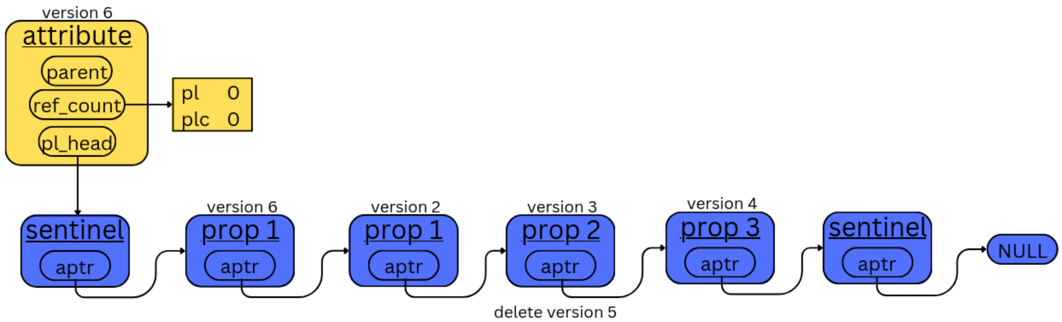
1. If a property has a delete version set equal to or earlier than the accessed version of the class,
2. If a property has a more recent version that is valid within the accessed version of the class.



Properties use delete versions to handle property deletion at the version level. When deleted a property will have its delete version set to the next version, and upon completion the class will update its current version to that new version. This prevents a property from being deleted out from under another thread. For example, Image 3 shows property 2 with a delete version set to delete version 5. If a thread was accessing version 4, and another thread performs the delete of property 2, property 2's delete version is set to the next version (version 5) and the class's version is updated to version 5 upon completion. The thread which was accessing the class at version 4 still has access to property 2 since it was still active in version 4.



When modifying an existing property in a property list class, a new property is created that is a copy of the existing property with the modification. For example, in Image 4 there are two property 1 structures. This is because a modification was made to property 1 while the class was at version 5. A new property 1 structure is created with the modification and it is assigned a create version of the class's next version, and then the class's current version is updated to match.

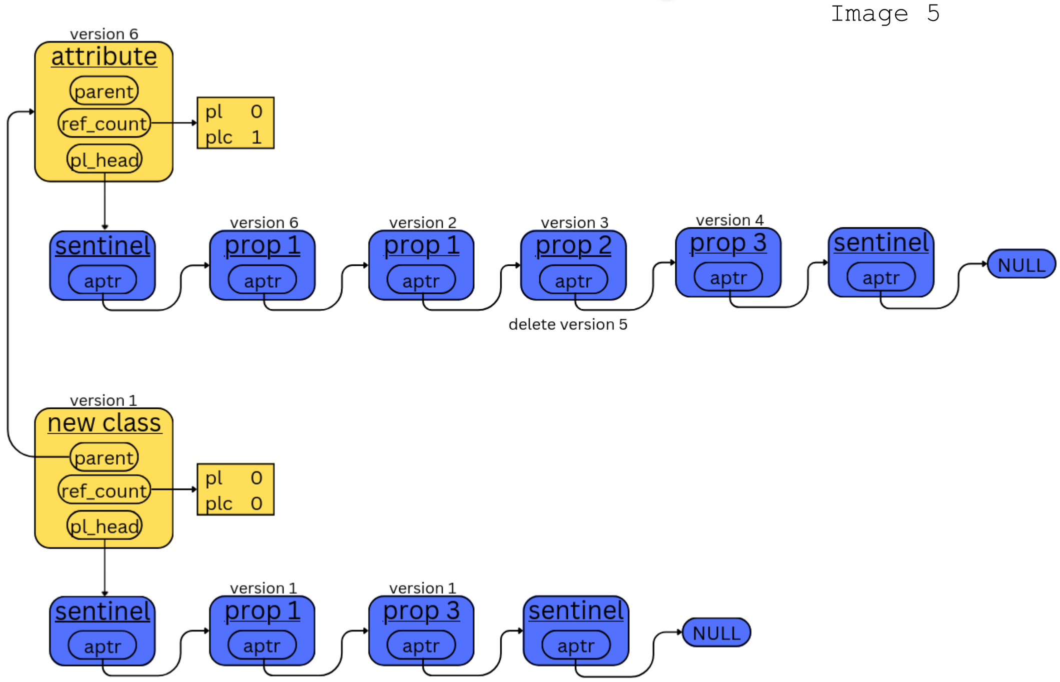


Deriving a new class from another class that has multiple versions, the version from which the new class is derived determines which properties are inherited. For instance, shown in Image 5, if the new class is created from version 5 of the attribute class, two properties will not be inherited. Property 2, because it has a delete version of 5, and version 6 property 1, because it only exists at version 6 or newer. Additionally, the new class and all of its inherited properties are set to version 1 since no modifications have to be made to it because it was just created.

Upon the new class successfully being created, the attribute class will have its derived reference count incremented as well.

The multithread version of property lists contain an array of instances of

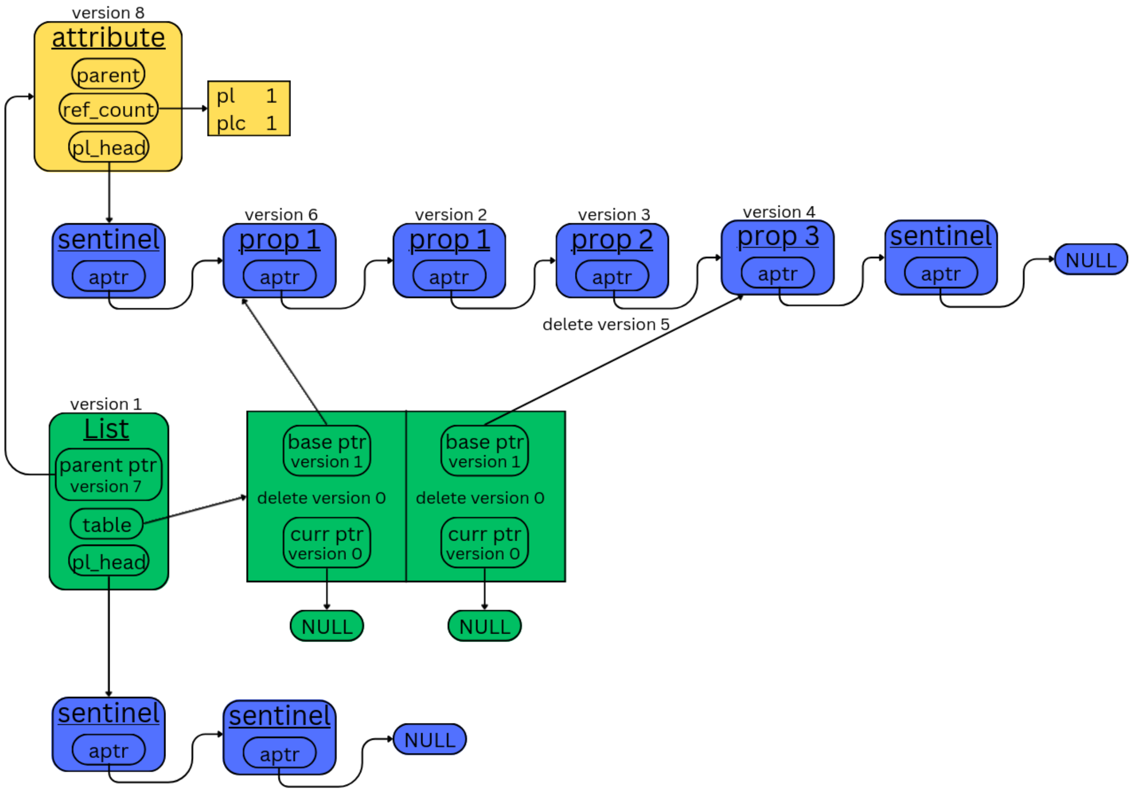
Image 5



H5P_mt_list_table_entry_t (table entries or entries) called a lookup table. Each entry includes the following:

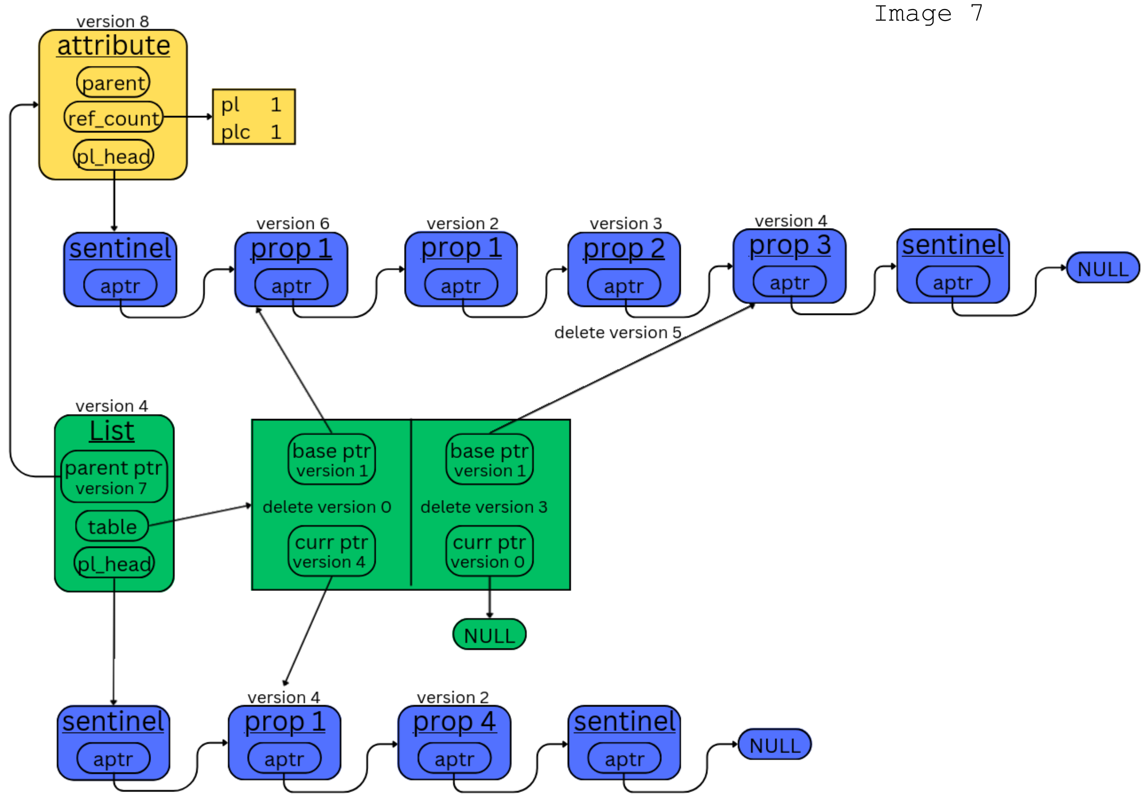
1. The checksum and name of the property the entry's pointers point to.
2. Base pointers, which point to valid properties in the list's parent class's LFSLL, and base versions which are initialized to 1, the version the property list is upon creation and do not change for the life of the property list.
3. Base delete version which is also initialized to zero.
4. Current (curr) pointers, initialized to NULL, and current versions initialized to zero.

Image 6 depicts how a list looks after creation, specifically depicting how the lookup table only creates entries for valid properties. Only the most recent version of property 1 is inherited, because it is the most recent version of that property that is equal to or less than the version the list was derived from, version 7. If it was derived from version 5 or earlier, then property 1 version 2 would be the valid version and the one the list inherits.



Property lists follow the same process as property list classes when creating a new property. A new property structure is created and inserted into the list's LFSLL with the list's next version number assigned as its create version, and the list's current version is then updated to match. However, because of the lookup table there is an extra step when modifying a property in a property list, as shown in Image 7. After inserting the new property structure into the LFSLL the lookup table must be searched to find the entry representing the property being modified. The current pointer of the entry is updated to point to the newly created property and the current version is updated to match the version assigned as the new property's create version. Additionally, when setting a delete version, if the property is still the base version that was inherited from the parent class, the list must find the entry in the lookup table and set the base delete version in the entry. This is because the base pointer points to the property in the parent's LFSLL and the list cannot modify its parent class, so this value is set to represent the base version being deleted from the list.

Image 7



Appendix 7 – Comments

From Jordan Henderson 8/26/24:

Hi John,

here are my notes after reviewing the H5P document:

General

- I'm struggling to understand the search algorithm for a property as it relates to the lkup_tbl and pl_head properties of the H5P_mt_list_t structure, as well as the performance characteristics / algorithmic complexity of the algorithm.

I may need some live discussion to grasp this.

pg. 11/12

- This is definitely a documentation issue for H5Pcopy. I occasionally use it for copying a property list, but didn't realize it can be used to copy a property list class as well.

pg. 28

- For padding out to 128 bits, it seems it might be cleaner to use a union similar to:

```
typedef struct H5P_mt_prop_aptr_t {
   union {
      struct {
         ... fields ...
      }
      struct {
         uint64_t padding[2];
      }
   }
}
```

Either way, I think we should always look to memset to take care of garbage in the structure rather than relying on a programmer setting the field to something.

pg. 28/29

- Again, we should use fixed-width types and possibly a compile-time assertion to ensure the structure is exactly as big as we expect it to be, then use memset to ensure there's no garbage in the structure.

pg. 40

- General note, the somewhat recent H5T refactoring moved towards replacing internal uses of datatype IDs with the pointer to the H5T_t structure for performance reasons. This structure (H5P_mt_list_t) holds the ID of the parent property class (pclass_id) and the ID assigned to the property list (plist_id). If possible, we should see if we can eliminate these and favor direct use of the pointers (pclass_ptr / no current alternative for plist_id) in this and other similar structures since the ID lookups can be expensive.